aws

User Guide

Amazon EventBridge

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon EventBridge: User Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Services or capabilities described in AWS documentation might vary by Region. To see the differences applicable to the AWS European Sovereign Cloud Region, see the AWS European Sovereign Cloud User Guide.

Table of Contents

What Is Amazon EventBridge? 1
Sign up for an AWS account 2
Agent setup guide 3
Prerequisites 3
Claude Code 3
Install aws-core plugin 3
Install Serverless MCP Server 4
Codex 4
Install AWS serverless skill 4
Install Serverless MCP Server 4
Cursor 5
Install AWS serverless skill 5
Install Serverless MCP Server 5
Kiro 5
Install AWS serverless skill 5
Install Serverless MCP Server 6
Install Kiro powers 6
GitHub Copilot 6
Install AWS serverless skill 6
Install Serverless MCP Server 7
Devin Desktop 7
Install AWS serverless skill 7
Install Serverless MCP Server 7
OpenCode 8
Install AWS serverless skill 8
Install Serverless MCP Server 8
Other agents 8
Install AWS serverless skill 9
Install Serverless MCP Server 9
Event bus (Custom) 10
What is the Custom Event Bus? 10
Key features 11
Resources 11
One bus, many accounts 12

How the Custom Event Bus routes and retains events ... 12
How the Custom Event Bus differs from Custom Event Bus - Classic ... 13
Use cases ... 13
Pricing ... 14
Next steps ... 14
Creating a bus ... 14
Creating a bus with the AWS CLI ... 14
Choosing the retention period ... 16
Bus states and what each allows ... 17
Deleting a bus that has subscribers ... 18
Recovering from a failed key change ... 18
Sharing a bus ... 18
Prerequisites for sharing a bus ... 19
Sharing a bus ... 19
Unsharing a bus ... 20
Working with a bus that is shared with you ... 20
Billing and metering ... 20
Write a custom resource policy ... 21
Events that arrive without a publish call ... 22
Publishing events ... 22
Publish APIs ... 23
Publishing raw bytes ... 25
Reading the response ... 25
Supported content types ... 25
Event sources ... 32
Event structure ... 37
Subscribing to events ... 39
Filtering which events reach the target ... 40
Transforming the payload ... 40
Delivering to one target ... 41
Retries and dead-letter queues ... 42
The delivery role ... 42
Example: deliver large orders to a queue ... 43
Troubleshooting: you published and nothing arrived at the target ... 44
Targets ... 45
Filtering ... 56

Input transformations ... 60
Batching ... 66
Retries and dead-letter queues ... 67
Updating a subscriber ... 71
Replaying events ... 78
Ordering and deduplication ... 79
Ordering ... 79
Deduplication at publish time ... 80
Migrating from Custom Event Bus - Classic ... 80
What does not change ... 82
Defaults that differ ... 82
Classic targets without a bespoke equivalent ... 83
Migration sequence ... 84
Access control ... 85
Actions and condition keys ... 85
Revoking ... 93
Permissions by scenario ... 94
Encryption ... 98
Options for Encryption at Rest ... 98
Encrypting Data at Rest using Customer Managed KMS Keys for Custom Event Bus ... 99
Monitoring EventBridge Custom Event Bus interaction with AWS KMS ... 106
Observability ... 108
Metrics ... 108
Publish metrics ... 110
Subscriber logs ... 111
API calls in AWS CloudTrail ... 116
CloudFormation resources ... 116
The resource types ... 116
Example: a bus, a queue, and a subscriber ... 116
Return values ... 118
Properties that replace the resource ... 119
Referring to a bus from another stack ... 120
Names, endpoints, and permissions ... 120
Names to use for each interface ... 121
Why the IAM namespace is shared with Custom Event Bus - Classic ... 122
Event bus (default and Classic) ... 124

How event buses work ... 124
Event bus concepts ... 126
Event buses ... 126
Events ... 127
Event sources ... 128
Rules ... 128
Targets ... 129
Advanced features ... 129
Get started with event buses ... 131
Prerequisites ... 132
Create the rule ... 132
Explore the rule ... 135
Deleting resources ... 137
Template details ... 137
Template ... 138
Tutorials ... 147
Create a sample application ... 149
Archive and replay events ... 154
Download code bindings ... 159
Use input transformer ... 161
Log Auto Scaling group states ... 166
Create rule for AWS API calls via CloudTrail ... 170
Log Amazon EC2 instance states ... 175
Log Amazon S3 object operations ... 179
Send events using schemas ... 184
Schedule automated Amazon EBS snapshots ... 189
Send an email when events happen ... 191
Schedule a Lambda function ... 195
Send events to Datadog ... 196
Send events to Salesforce ... 201
Send events to Zendesk ... 206
Creating an event bus ... 210
Updating an event bus ... 213
Updating a default bus using CloudFormation ... 214
Deleting an event bus ... 215
Generating a CloudFormation template from an event bus ... 216

Considerations when using a generated template ... 217
Permissions for event buses ... 218
Managing event bus permissions ... 218
Send events to a cross-account default bus ... 221
Send events to a cross-account custom bus ... 221
Send events to event bus in the same account ... 222
Send events from a specific rule cross-Region ... 223
Send events from specific Regions ... 224
Deny sending events from specific Regions ... 225
Events ... 226
AWS service events ... 226
Sending events with PutEvents ... 227
Retrying event delivery ... 233
Using dead-letter queues ... 234
Events via CloudTrail ... 239
EventBridge events ... 272
Rules ... 282
Rules that match on event data ... 282
Rules that run on a schedule ... 283
Managed rules ... 283
Creating rules (Enhanced Builder) ... 285
Creating rules (Advanced Builder) ... 290
Disabling or deleting a rule ... 303
Best practices for rules ... 303
Using AWS SAM templates ... 305
Generating a CloudFormation template from a rule ... 307
Targets ... 309
Target types ... 310
Target parameters ... 311
Permissions ... 313
AWS Batch job queues ... 315
CloudWatch Logs groups ... 315
CodeBuild projects ... 316
Amazon ECS tasks ... 316
Incident Manager Response Plans ... 316
Systems Manager Run Command ... 316

Amazon SQS queues ... 318
API Gateway targets ... 319
AWS AppSync targets ... 321
Cross-account services as targets ... 325
Cross-account event buses as targets ... 329
Cross-Region event buses as targets ... 332
Same account event buses as targets ... 333
Input transformation ... 336
Archive and replay ... 349
Archiving events ... 349
Replaying events ... 351
Creating archives ... 353
Updating archives ... 359
Deleting archives ... 360
Creating event replays ... 360
Canceling event replays ... 361
Global endpoints ... 362
Recovery Time & Recovery Point Objectives ... 362
Event replication ... 363
Working with global endpoints by using an AWS SDK ... 364
Available Regions ... 364
Creating a global endpoint ... 365
Best practices ... 367
Template for Route 53 health check ... 368
Logging event buses ... 373
How logging works ... 374
Specifying log level ... 376
Including details in logs ... 378
Truncating logs ... 378
Error reporting in logs ... 378
What gets logged ... 379
Event bus log schema ... 382
Services and tools integration ... 387
AWS CloudFormation ... 388
Kafka connector ... 389
Interface VPC Endpoints ... 390

AWS X-Ray ... 394
Monitoring ... 395
EventBridge metrics ... 397
Dimensions for EventBridge metrics ... 404
Monitoring event delivery ... 404
Troubleshooting ... 406
My rule ran but my Lambda function wasn't invoked ... 407
I just created or modified a rule, but it didn't match a test event ... 409
My rule didn't run at the time I specified in the `ScheduleExpression` ... 409
My rule didn't run at the time that I expected ... 409
My rule matches AWS global service API calls but it didn't run ... 410
The IAM role associated with my rule is being ignored when the rule runs ... 411
My rule has an event pattern that is supposed to match a resource, but no events match . 411
My event's delivery to the target was delayed ... 411
Some events were never delivered to my target ... 412
My rule ran more than once in response to one event ... 412
Preventing infinite loops ... 412
My events are not delivered to the target Amazon SQS queue ... 412
My rule runs, but I don't see any messages published into my Amazon SNS topic ... 413
My Amazon SNS topic still has permissions for EventBridge even after I deleted the rule associated with the Amazon SNS topic ... 415
Which IAM condition keys can I use with EventBridge? ... 415
How can I tell when EventBridge rules are broken? ... 415
Working with AWS SDKs ... 416
Pipes ... 417
How Pipes work ... 417
Pipes concepts ... 418
Pipe ... 419
Source ... 419
Filters ... 419
Enrichment ... 420
Target ... 420
Get started with pipes ... 420
Create the pipe ... 421
Explore the pipe ... 423
Deleting resources ... 426

Template details ... 427
Template ... 428
Permissions for pipes ... 434
DynamoDB permissions ... 435
Kinesis permissions ... 435
Amazon MQ permissions ... 436
Amazon MSK permissions ... 436
Self managed Apache Kafka permissions ... 437
Amazon SQS permissions ... 438
Enrichment and target permissions ... 439
Creating a pipe ... 439
Specifying a source ... 439
Configuring filtering ... 444
Defining enrichment ... 445
Configuring a target ... 446
Configuring pipe settings ... 446
Validating configuration parameters ... 449
Starting or stopping a pipe ... 450
Pipe sources ... 451
DynamoDB stream ... 451
Kinesis stream ... 455
Amazon MQ message broker ... 459
Amazon MSK topic ... 464
Apache Kafka stream ... 473
Amazon SQS queue ... 479
Filtering ... 484
Filtering Amazon SQS messages ... 486
Filtering Kinesis and DynamoDB messages ... 487
Filtering Amazon MSK, self managed Apache Kafka, and Amazon MQ messages ... 488
Differences with Lambda ESM ... 490
Using comparison operators ... 490
Enrichment ... 490
Filtering events using enrichment ... 491
Invoking enrichments ... 491
Targets ... 491
Target parameters ... 492

Permissions ... 494
Invoking targets ... 494
Payload size limits ... 494
AWS Batch job queues ... 495
CloudWatch Logs group ... 495
Amazon ECS task ... 495
Lambda functions and Step Functions workflows ... 495
Timestream for LiveAnalytics table ... 495
EventBridge event bus ... 496
Batching and concurrency ... 496
Batching behavior ... 496
Throughput and concurrency behavior ... 498
Input transformation ... 499
Reserved variables ... 501
Input transform example ... 502
Implicit body data parsing ... 503
Common issues with transforming input ... 504
Logging pipe performance ... 505
How pipe logging works ... 506
Specifying log level ... 507
Including execution data in logs ... 509
Error reporting in log records ... 511
Pipe execution steps ... 512
Log schema reference ... 517
Log and monitor ... 520
Error handling & troubleshooting ... 523
Retry behavior ... 523
Invocation errors and retry behavior ... 524
Amazon SQS retry timing ... 525
DLQ behavior ... 526
Pipe failure states ... 526
Custom encryption failures ... 527
Tutorial: Create a pipe that filters events ... 528
Prerequisites ... 528
Create the pipe ... 530
Confirm the pipe filters events ... 531

Clean up resources ... 533
Template for prerequisites ... 534
Generating a pipe template ... 535
Resources included in pipe templates ... 536
Considerations when using a generated template ... 536
Generating a CloudFormation template from EventBridge Pipes ... 537
Scheduler ... 538
Set up the execution role ... 538
Related resources ... 539
Create a schedule ... 539
Creating a scheduled rule (legacy) ... 544
Create a scheduled rule ... 546
Setting a scheduled rule pattern (legacy) ... 553
Cron expressions ... 553
Rate expressions ... 558
Schemas ... 560
Schema registry API property value masking ... 560
Finding a schema ... 561
Schema registries ... 562
Creating a schema ... 563
Creating a schema using a template ... 563
Creating a schema from event JSON ... 566
Inferring schemas on event buses ... 569
Generating code bindings ... 571
Third-party integrations ... 572
API destinations ... 572
Service-linked role ... 574
Headers in requests ... 574
API destination error codes ... 576
Invocation rate and event delivery ... 577
Region availability ... 577
Create an API destination ... 578
Creating rules with API destination targets ... 579
CloudEvents ... 580
API destination partners ... 582
Connections ... 597

Storing connection authorization ... 599
Connecting to private APIs ... 599
Connection authorization methods ... 609
Connection states ... 610
Creating connections ... 612
Updating connections ... 617
De-authorizing connections ... 619
Deleting connections ... 620
SaaS partner event sources ... 621
Supported SaaS partner integrations ... 621
Receive SaaS events ... 624
Receiving SaaS events through Lambda ... 627
Receiving events from Salesforce ... 635
Event patterns ... 640
Event patterns for event buses ... 641
Event patterns for pipes ... 642
Event pattern syntax ... 643
Matching event values ... 643
Content-based filtering in event patterns ... 644
Considerations when creating event patterns ... 644
Matching on field values ... 646
Field matching ... 646
Value matching ... 647
Matching on null values and empty strings ... 649
Using null values in AWS CloudFormation templates ... 650
Matching on multiple field values ... 651
Comparison operators ... 653
Prefix matching ... 656
Suffix matching ... 656
Anything-but matching ... 657
Numeric matching ... 660
IP address matching ... 661
Exists matching ... 661
Equals-ignore-case matching ... 662
Wildcard matching ... 662
Complex multiple matching ... 664

Complex $or matching ... 665
Best practices ... 666
Avoid writing infinite loops ... 666
Make as precise as possible ... 667
Scope for future updates ... 668
Validate ... 670
Testing an event pattern ... 670
Code examples ... 675
Basics ... 676
Hello EventBridge ... 676
Learn the basics ... 680
Actions ... 742
Scenarios ... 796
Create and trigger a rule ... 797
Send event notifications to EventBridge ... 815
Use scheduled events to invoke a Lambda function ... 817
Security ... 820
Data protection ... 821
Encryption ... 821
Encryption at rest ... 822
KMS key options ... 824
Authorizing key use ... 825
Maintaining key access ... 826
Encrypting event buses ... 827
Encrypting pipe data ... 840
Encrypting archives ... 845
Encrypting connections ... 850
Tag-based policies ... 855
IAM ... 856
Authentication ... 856
Access control ... 858
Managing access ... 859
Using identity-based policies ... 864
Using resource-based policies ... 882
Cross-service confused deputy prevention ... 889
Resource-based policies for schemas ... 892

Permissions reference ... 896
IAM policy conditions ... 899
Using service-linked roles ... 916
CloudTrail logs ... 922
Management events ... 923
Data events ... 923
Management event examples ... 927
Data event examples ... 930
Events for Pipe actions ... 935
Compliance validation ... 937
Resilience ... 938
Infrastructure security ... 939
Security and vulnerability analysis ... 940
Dual-stack endpoints ... 941
Considerations ... 942
Tags ... 943
Managing event bus tags ... 944
Quotas ... 945
Custom Event Bus quotas ... 945
Service Quotas ... 945
Fixed limits ... 947
Custom Event Bus - Classic quotas ... 948
API destinations and connections ... 957
Schema Registry quotas ... 957
Pipes quotas ... 958
Feature availability ... 961
Document History ... 965

What Is Amazon EventBridge?

EventBridge is a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. Event-driven architecture is a style of building loosely-coupled software systems that work together by emitting and responding to events. Event-driven architecture can help you boost agility and build reliable, scalable applications.

EventBridge provides simple and consistent ways to ingest, filter, transform, and deliver events so you can build applications quickly.

EventBridge offers two event buses. The Custom Event Bus routes events to subscribers that consumers create themselves, retains events for a period that you set, and supports ordered delivery, replay, and non-JSON payloads. Custom Event Bus - Classic routes events with rules and targets that the bus owner writes. For new applications, start with the Custom Event Bus. Custom Event Bus - Classic remains available, and this guide describes both.

EventBridge includes two ways to process and deliver events: *event buses* and *pipes*.

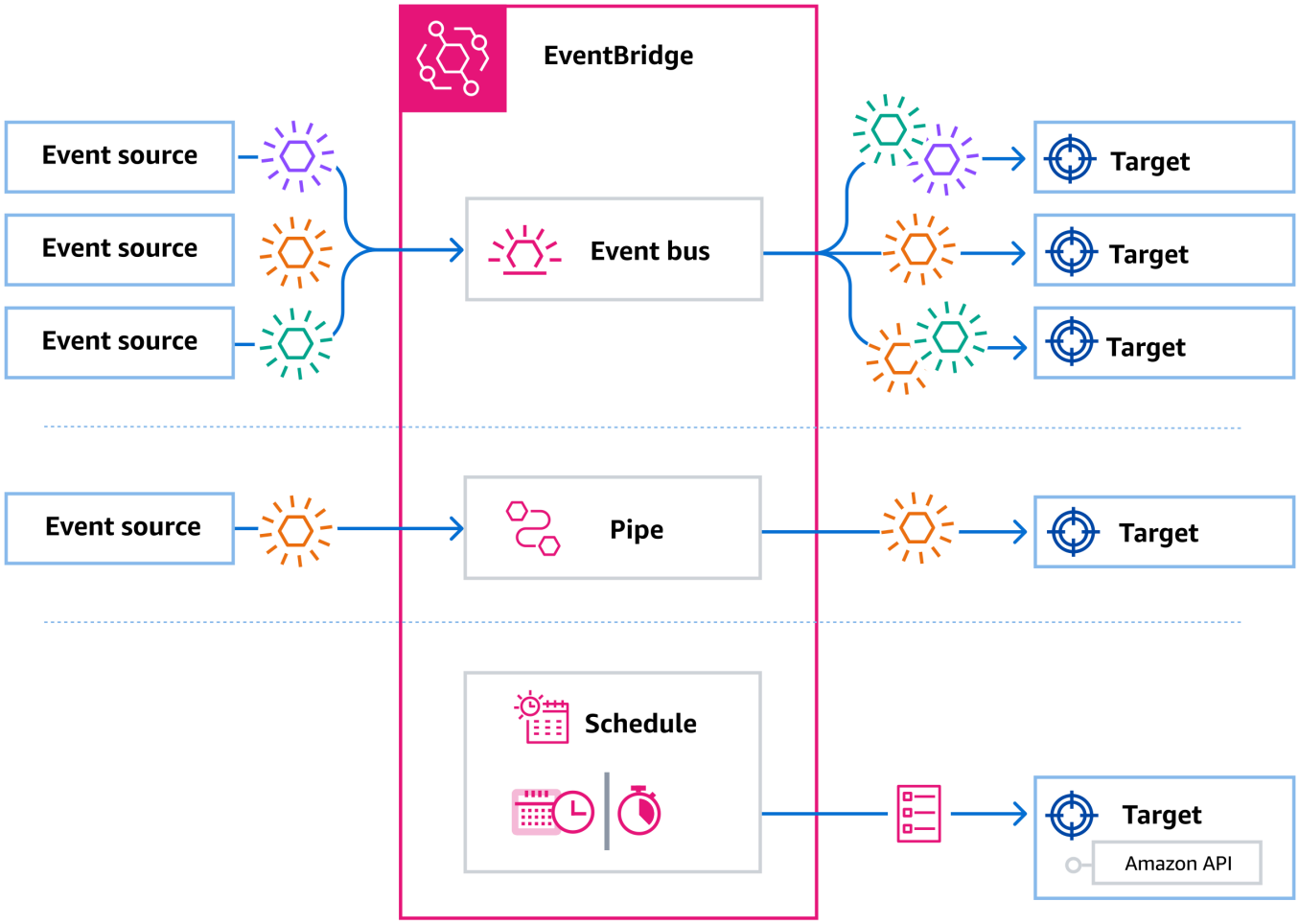
- Event buses are routers that receive events and delivers them to zero or more targets. Use EventBridge to route events from sources such as home-grown applications, AWS services, and third-party software to consumer applications across your organization.

  Event buses are well-suited for routing events from many sources to many targets, with optional transformation of events prior to delivery to a target.
- Pipes EventBridge Pipes is intended for point-to-point integrations; each pipe receives events from a single source for processing and delivery to a single target. Pipes also include support for advanced transformations and enrichment of events prior to delivery to a target.

  Pipes and event buses are often used together. A common use case is to create a pipe with an event bus as its target; the pipe sends events to the event bus, which then sends those events on to multiple targets. For example, you could create a pipe with a DynamoDB stream for a source, and an event bus as the target. The pipe receives events from the DynamoDB stream and sends them to the event bus, which then sends them on to multiple targets according to the rules you've specified on the event bus.

In addition, EventBridge provides EventBridge Scheduler, a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler,

you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.



Sign up for an AWS account

To get started with AWS, you need an AWS account. For information about creating an AWS account, see Getting started with an AWS account in the *AWS Account Management Reference Guide*.

Agent setup guide

AI coding agents can accelerate serverless development by providing intelligent assistance for application scaffolding, deployment configuration, debugging, and infrastructure-as-code generation. By configuring your agent with the right combination of plugins, skills, and MCP Servers, you equip it with expertise required to build and manage serverless applications.

Work with your coding agent

For a quick-start reference, you can drop this link directly into your agent's context: `https://docs.aws.amazon.com/eventbridge/latest/userguide/samples/aws-eventbridge-agent-setup.md`

Choose the installation method that matches your development environment.

Prerequisites

Ensure the following prerequisites are in place for installing aws-core plugin and Serverless MCP Server:

- uv installed on your system.
- (Optional) An AWS account with IAM credentials set up on your local machine. Credentials are required for tools that execute AWS API calls and run scripts, but not for searching documentation or discovering skills. If you do not have credentials configured, see Setting up the AWS MCP Server for detailed instructions.

Claude Code

Install the aws-core plugin, part of the Agent Toolkit for AWS, that bundles the AWS MCP Server configuration and a curated set of agent skills (including the AWS serverless skill).

Install aws-core plugin from Agent Toolkit for AWS

Run the following command to install the plugin:

```
/plugin marketplace add aws/agent-toolkit-for-aws
```

```
/plugin install aws-core@agent-toolkit-for-aws
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

This installs agent skills including the AWS serverless skill, agent hooks, and the AWS MCP Server configuration in one step.

Install Serverless MCP Server

Replace *AWS_PROFILE* with your local AWS profile name.

```
claude mcp add awslabs-aws-serverless-mcp \
  -e AWS_PROFILE=default \
  -e AWS_REGION=us-east-1 \
  -e FASTMCP_LOG_LEVEL=ERROR \
  --scope user \
  -- uvx awslabs.aws-serverless-mcp-server@latest
```

Codex

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Replace *AWS_PROFILE* with your local AWS profile name.

```
codex mcp add awslabs-aws-serverless-mcp \
```

```
  --env AWS_PROFILE=default \
  --env AWS_REGION=us-east-1 \
  --env FASTMCP_LOG_LEVEL=ERROR \
  -- uvx awslabs.aws-serverless-mcp-server@latest
```

Cursor

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the following to `.cursor/mcp.json` under `"mcpServers"`. Replace *`AWS_PROFILE`* with your local AWS profile name.

```
"awslabs.aws-serverless-mcp": {
  "command": "uvx",
  "args": ["awslabs.aws-serverless-mcp-server@latest"],
  "env": {
    "AWS_PROFILE": "default",
    "AWS_REGION": "us-east-1",
    "FASTMCP_LOG_LEVEL": "ERROR"
  }
}
```

Kiro

Install AWS serverless skill in Kiro CLI

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the following to `~/.kiro/settings/mcp.json` under `"mcpServers"`. Replace *`AWS_PROFILE`* with your local AWS profile name.

```
"awslabs.aws-serverless-mcp": {
  "command": "uvx",
  "args": ["awslabs.aws-serverless-mcp-server@latest"],
  "env": {
    "AWS_PROFILE": "default",
    "AWS_REGION": "us-east-1",
    "FASTMCP_LOG_LEVEL": "ERROR"
  },
  "disabled": false
}
```

Install Kiro powers (from Kiro IDE)

Install the following Kiro powers that provide specialized contexts and tools to the Kiro agents on-demand:

- **Serverless Application Model (SAM)** — install via Kiro powers link.

GitHub Copilot

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the following to `.vscode/mcp.json` under `"mcpServers"`. Replace *AWS_PROFILE* with your local AWS profile name.

```
"awslabs.aws-serverless-mcp": {
  "command": "uvx",
  "args": ["awslabs.aws-serverless-mcp-server@latest"],
  "env": {
    "AWS_PROFILE": "default",
    "AWS_REGION": "us-east-1",
    "FASTMCP_LOG_LEVEL": "ERROR"
  }
}
```

Devin Desktop

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the following to `/.codeium/windsurf/mcp_config.json` under `"mcpServers"`. Replace *AWS_PROFILE* with your local AWS profile name.

```
"awslabs.aws-serverless-mcp": {
  "command": "uvx",
  "args": ["awslabs.aws-serverless-mcp-server@latest"],
  "env": {
    "AWS_PROFILE": "default",
    "AWS_REGION": "us-east-1",
```

```
    "FASTMCP_LOG_LEVEL": "ERROR"
  }
}
```

OpenCode

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the following to `/.config/opencode/opencode.jsonc` under `"mcpServers"`. Replace *`AWS_PROFILE`* with your local AWS profile name.

```
"awslabs.aws-serverless-mcp": {
  "command": "uvx",
  "args": ["awslabs.aws-serverless-mcp-server@latest"],
  "env": {
    "AWS_PROFILE": "default",
    "AWS_REGION": "us-east-1",
    "FASTMCP_LOG_LEVEL": "ERROR"
  }
}
```

For all other agents compatible with agent skills and MCP Server configuration

For any other agent that supports the open-source agent skills format and MCP Server configuration, follow these steps:

Install AWS serverless skill

```
npx skills add https://github.com/aws/agent-toolkit-for-aws --skill aws-serverless --
yes --global
```

Then install the Amazon EventBridge skill:

```
npx skills add https://github.com/aws/agent-toolkit-for-aws/tree/main/skills/
specialized-skills/serverless-skills/amazon-eventbridge-event-bus --yes --global
```

Install Serverless MCP Server

Add the Serverless MCP Server to your agent's MCP client configuration file under `"mcpServers"`. Replace *AWS_PROFILE* with your local AWS profile name.

```
{
  "mcpServers": {
    "awslabs.aws-serverless-mcp": {
      "command": "uvx",
      "args": ["awslabs.aws-serverless-mcp-server@latest"],
      "env": {
        "AWS_PROFILE": "default",
        "AWS_REGION": "us-east-1",
        "FASTMCP_LOG_LEVEL": "ERROR"
      }
    }
  }
}
```

Amazon EventBridge event bus (Custom)

The EventBridge Custom Event Bus is an event bus that routes events to subscribers and retains them for a period that you choose. Producers publish events to the bus. Consumers create *subscribers* on the bus. Each subscriber filters the events, optionally transforms them, and delivers each matching event to one target. Because the bus retains events, a subscriber can deliver events in order, start from a point in the past, and pause and resume without losing events.

EventBridge also provides Custom Event Bus - Classic, the event bus that uses rules and targets. The two products share the `events:` IAM namespace and the `events.amazonaws.com` service principal, and both remain available. This chapter describes the Custom Event Bus. For Custom Event Bus - Classic, see *Event bus (default and Classic)*. To move existing rules to subscribers, see the section called "Migrating from Custom Event Bus - Classic".

This chapter contains the following topics:

- the section called "What is the Custom Event Bus?"
- the section called "Creating a bus"
- the section called "Sharing a bus"
- the section called "Publishing events"
- the section called "Subscribing to events"
- the section called "Ordering and deduplication"
- the section called "Migrating from Custom Event Bus - Classic"
- the section called "Access control"
- the section called "Encryption"
- the section called "Observability"
- the section called "CloudFormation resources"
- the section called "Names, endpoints, and permissions"

What is the EventBridge Custom Event Bus?

The EventBridge Custom Event Bus is a serverless event bus that one team runs and many teams and accounts use. You publish events to it with `PutEvents` or `PutRawEvents`. It routes each event to the subscribers whose filters match it, and it keeps a copy of every event for a retention period

that you set on the bus. You share the bus with other accounts through AWS Resource Access Manager (AWS RAM), and each of those accounts attaches its own subscribers. The bus owner writes no routing on their behalf.

In the EventBridge console, the bus type is shown as **Custom Event Bus** for this bus and **Custom Event Bus - Classic** for the buses that use rules and targets. For the CLI, SDK, and IAM names, see the section called “Names, endpoints, and permissions”.

Key features

- Cross-account sharing through AWS RAM, with four managed permissions that let a consumer account publish, subscribe, attach event sources, or all three.
- Retention of every event for 1 to 365 days, set on the bus.
- Ordered (FIFO) delivery within an event group, per subscriber.
- A starting position for each subscriber, so a new subscriber can read retained events and a paused subscriber can resume without loss.
- Deduplication at publish time, by content or by a key that you supply, within a 5-minute window.
- `PutRawEvents` for JSON, Avro, Protobuf, or raw bytes, with schema registry deserialization for filtering.
- Universal targets that call any state-changing AWS API action directly, alongside the bespoke targets.

Resources

You work with three resources.

Event bus

The event bus receives the events you publish with `PutEvents` or `PutRawEvents`, routes them to subscribers, and retains them. You set the retention period, an optional AWS KMS key, and optional cross-account sharing on the bus. For the states a bus moves through, see the section called “Bus states and what each allows”.

Subscriber

A subscriber names the events you want, with filters, and the one target that receives them. You create it with `CreateSubscriber`. It watches one bus, keeps the events that match its filters,

optionally transforms each one, and delivers it to exactly one target. To send an event to several targets, create one subscriber for each target.

Event source

An event source ingests events onto a bus from an AWS service or from a software as a service (SaaS) partner. You create it with `CreateEventSource`, naming the origin and the destination bus, and EventBridge moves the events for you.

Bus and subscriber Amazon Resource Names (ARNs) end in an identifier that EventBridge generates, for example `arn:aws:events:us-east-1:111122223333:event-busv2/orders/EXAMPLE1234567890abcdef`. Store the full ARN rather than the name. If you delete a bus and create another with the same name, the new bus has a different identifier.

One bus, many accounts

To let another account use your bus, share it with AWS RAM and attach one of the managed permissions, for example `AWSRAMEventBridgeEventBusV2SubscribeOnly` to let that account attach subscribers. The account then creates and owns its own subscribers on your bus: it sees only its subscribers when it lists them, and you see all of them. A consumer account cannot change or delete the bus, and you can withdraw any of its subscribers with `RevokeResource`. For the permissions, resource policies, and revocation, see the section called “Sharing a bus”.

How the Custom Event Bus routes and retains events

The bus routes each event to matching subscribers as it arrives. The bus also retains the event for the number of days that you configure. Retention makes two capabilities possible.

- A subscriber can start from a point in the past. That is how you replay retained events, and how a subscriber that you create today reads events that were published yesterday.
- A subscriber can pause and later resume without losing events.

For more information, see the section called “Choosing the retention period” and the section called “Replaying events”.

How the Custom Event Bus differs from Custom Event Bus - Classic

Capability	Custom Event Bus	Custom Event Bus - Classic
Cross-account use	Share the bus with AWS RAM. Each consumer account creates and owns its subscribers on the bus.	Grant other accounts through the bus resource policy, or forward events to a bus in the other account.
Retention	Built into the bus, for a period that you set.	Not retained. To keep events, add an archive.
Ordering	FIFO delivery within an event group, per subscriber.	Not available.
Replay	A subscriber's starting position over retained events.	Replay from an archive.
Publish APIs	`PutEvents` for JSON and `PutRawEvents` for any format, including Avro and Protobuf.	`PutEvents` for JSON.
Deduplication	At publish time, by content or by a key that you supply.	Not available.
Targets per routing resource	One target per subscriber.	Up to five targets per rule.

Use cases

- A platform team runs one `orders` bus and shares it with the accounts of the fulfillment, billing, and analytics teams. Each team attaches its own subscribers; the platform team never edits routing for them.
- A payments service publishes with an `EventGroupId` per account number and a FIFO subscriber delivers each account's events to a queue in order.
- A new analytics consumer is created with a starting position 30 days back and reads a month of retained events before it catches up to live traffic.

Pricing

You pay for the events you publish, the deliveries to subscribers, and retention. For the rates, see Amazon EventBridge pricing.

Next steps

- Learn which names to use in the CLI, SDKs, and IAM: the section called “Names, endpoints, and permissions”.
- Publish your first events: the section called “Publishing events”.
- Create a subscriber that delivers them: the section called “Subscribing to events”.

Creating a bus

You create a bus with `CreateEventBus`, for example `aws eventsv2 create-event-bus --name orders`. The name is the only required parameter. At creation you can also set a description, the number of days the bus retains events, the AWS KMS key that encrypts them, and tags. The call returns at once with the bus in the `CREATING` state; publish to the bus or create a subscriber on it once `DescribeEventBus` reports `ACTIVE`.

Creating a bus with the AWS CLI

The following command creates a bus named `orders` that retains events for 30 days, encrypts them with your AWS KMS key, and carries one tag.

```
aws eventsv2 create-event-bus \
    --name orders \
    --description "Order events from the checkout service" \
    --storage-configuration RetentionPeriodInDays=30 \
    --encryption-configuration KmsKeyIdentifier=arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
    --tags team=orders
```

The following table lists what each parameter accepts.

Parameter	What it accepts
`--name`	Required. 1 to 256 characters: letters, digits, periods, hyphens, and underscores, starting with a letter or digit. The name must be unique in your account in the Region; creating a second bus with the same name fails with `ResourceAlreadyExistsException`.
`--description`	Up to 512 characters.
`--storage-configuration`	`RetentionPeriodInDays`, from 1 to 365. Omit it and the bus retains events for 1 day. You can change the period later with `UpdateEventBus`. See the section called "Choosing the retention period".
`--encryption-configuration`	`KmsKeyIdentifier`: a key ID, key ARN, alias name, or alias ARN of a AWS KMS customer managed key. Omit it and EventBridge encrypts events with an AWS owned key. For the key policy the key needs, see the section called "Encryption".
`--tags`	A map of tag keys to values, for example `team=orders,env=prod`. For tag-based access control, see the section called "Actions and condition keys".

The response returns the bus ARN, which every later call on the bus uses. The ARN ends in an identifier that EventBridge generates, so you cannot construct it from the name.

```
{
    "EventBusArn": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
    "Name": "orders",
    "Description": "Order events from the checkout service",
    "EncryptionConfiguration": {
```

```
        "KmsKeyIdentifier": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    "StorageConfiguration": {
        "RetentionPeriodInDays": 30,
        "RetentionWindowStartTime": "2026-09-21T16:00:00+00:00"
    },
    "State": "CREATING",
    "CreationTime": "2026-09-21T16:00:00+00:00"
}
```

Describe the bus by its ARN until `State` is `ACTIVE`.

```
aws eventsv2 describe-event-bus \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef
```

To create a bus with CloudFormation instead, see the section called “CloudFormation resources”. For the IAM actions that creating and describing a bus require, see the section called “Actions and condition keys”.

Choosing the retention period

A bus retains every event that it accepts for the number of days in `StorageConfiguration.RetentionPeriodInDays`, from 1 to 365; omit it and the bus retains events for 1 day. Retention is what lets a subscriber start from a point in the past, pause and resume without losing events, and read events again after a failure. A longer period costs more storage. You can change the period later with `UpdateEventBus`.

```
aws eventsv2 create-event-bus \
    --name orders \
    --storage-configuration RetentionPeriodInDays=30
```

`DescribeEventBus` returns `RetentionWindowStartTime`, the earliest point that a subscriber can read from. To deliver retained events to a subscriber, see the section called “Replaying events”.

Bus states and what each allows

Creating, updating, and deleting a bus are asynchronous. Every state other than `ACTIVE` refuses some operations, and the three failed states carry a `StateReason` that tells you why. The following table lists the states a bus moves through and what you can do in each.

State	How the bus gets there	What you can do
`CREATING`	`CreateEventBus` returned and the bus is being set up.	Describe and wait. You cannot publish, update, or delete the bus, or create a subscriber or event source on it.
`ACTIVE`	Creation or an update finished.	Everything: publish, create subscribers and event sources, update, share, delete.
`CREATE_FAILED`	Creation failed, which happens when EventBridge cannot use the AWS KMS key you named.	Delete the bus. Nothing else is accepted. Read `StateReason`, fix the key or its policy, and create the bus again.
`UPDATING`	`UpdateEventBus` is changing the encryption configuration. Other updates do not pass through this state.	Publish and deliver as normal. You cannot update or delete the bus until it returns to `ACTIVE`.
`UPDATE_FAILED`	An encryption update failed, again because EventBridge could not use the key.	The bus keeps running with its previous configuration. Retry the update after you fix the cause, or delete the bus.
`DELETING`	`DeleteEventBus` returned.	Wait. You cannot create a subscriber or event source on a bus that is being deleted.

State	How the bus gets there	What you can do
DELETE_FAILED	Deletion found a subscriber or event source that is not revoked.	Delete the bus again after its subscribers and event sources are deleted or revoked. Nothing else is accepted.

Deleting a bus that has subscribers

DeleteEventBus succeeds only when every subscriber and event source on the bus is deleted or revoked. A subscriber that another account created counts. You cannot delete that account's subscriber yourself, so either ask the owner to delete it or revoke it with RevokeResource; see the section called “Revoking”. If a subscriber is created between your check and the delete, the bus lands in DELETE_FAILED; deal with the new subscriber and delete again.

Recovering from a failed key change

Both failed states that a running bus can reach come from the AWS KMS key. When an update to a new key fails, retry UpdateEventBus after you fix the key's policy or grant. If the new key itself is unusable, the retry can name a different key, because EventBridge did not adopt the failed key. Do not disable, delete, or remove EventBridge's access to the previous key until DescribeEventBus shows the bus ACTIVE with the new key; a bus that loses its old key part-way through a change can only be repaired by restoring that key, or deleted. For the key permissions, see the section called “Encryption”.

Sharing a Custom Event Bus with other accounts

To let another account publish to your bus, attach subscribers to it, or attach event sources to it, share the bus with AWS Resource Access Manager (AWS RAM). Then choose one of the four managed permissions. For example, share the orders bus with account 444455556666 using AWSRAMEventBridgeEventBusV2SubscribeOnly, and that account can create subscribers on it. Write a resource policy on the bus instead when you need an explicit Deny, a condition key, or a grant that no managed permission expresses. The other account still needs the same actions in its own identity policies, because both sides must allow.

Prerequisites for sharing a bus

- You are familiar with IAM identity-based and resource-based policies, because sharing a bus requires matching permissions on both the bus and the consumer's identity.
- You must own the bus. A consumer account cannot share a bus onward.
- To share with accounts in your organization without an invitation, enable sharing with AWS Organizations in AWS RAM. To share with an account outside your organization, the account must accept the resource share invitation before it can use the bus.
- The bus and the consumer's subscribers are in the same Region and partition. A consumer publishes to the bus and creates subscribers on it in the bus's Region.

Sharing a bus

We recommend AWS RAM over a resource policy that you write yourself. With AWS RAM, you have one place to see and revoke what you have shared. AWS RAM also works with AWS Organizations and writes the bus's `AWS_RAM` resource policy for you. Create a resource share that names the bus ARN, the consumer accounts or organizational units, and one of the four AWS managed permissions for the bus resource type, each at `arn:aws:ram::aws:permission/`*`name`*.

The following table lists what the consumer account can do with each managed permission.

Managed permission	What the consumer account can do
`AWSRAMEventBridgeEventBusV2 PublishOnly`	Publish to the bus, and describe the bus
`AWSRAMEventBridgeEventBusV2 SubscribeOnly`	Attach subscribers to the bus, update their filters, and describe the bus
`AWSRAMEventBridgeEventBusV2 EventSourceAccess`	Attach and update event sources, and describe the bus
`AWSRAMEventBridgeEventBusV2 FullAccess`	Publish with either API, describe the bus, and create, update, and delete its own subscribers and event sources

```
aws ram create-resource-share \
    --name orders-bus-share \
    --resource-arns arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --principals 444455556666 \
    --permission-arns arn:aws:ram::aws:permission/
AWSRAMEventBridgeEventBusV2SubscribeOnly
```

To grant less than a managed permission, or to add a condition key, author an AWS RAM customer managed permission. AWS RAM rejects one that names a bus owner only action. Read a permission back with the AWS RAM `GetPermission` operation rather than assume its exact action list.

Unsharing a bus

To stop sharing, remove the bus or the principal from the resource share, or delete the share. EventBridge authorizes a subscriber or event source when it is created, so a subscriber the consumer created while the share was active keeps delivering after you unshare. To stop it, revoke it with `RevokeResource`. For more information about revoking a subscriber or an event source, see the section called “Revoking”. The consumer can no longer publish or create new subscribers once the share is gone.

Working with a bus that is shared with you

A bus shared with your account appears in your `ListEventBuses` results with the owner's account in its ARN, and you use it by that ARN, in the bus's Region. What you create on it belongs to you: when you list subscribers on the shared bus you see only your own subscribers, while the bus owner sees all of them. Sharing one bus exposes only that bus; you never see the owner's other buses. You cannot update or delete the bus, change its resource policy, or share it onward, and you cannot act on a subscriber that another account owns. The bus owner can withdraw any of your subscribers or event sources with `RevokeResource`; a revoked resource stops permanently and you can only delete it.

Billing and metering

Sharing a bus has no charge of its own. Publishing, delivery, and retention are billed as described on the Amazon EventBridge pricing page. The bus owner can see delivery volume per consumer account in the `EventsDelivered` and `EgressBytes` metrics with the `SubscriberAccount` dimension. For more information about observability, see the section called “Observability”.

Write a custom resource policy

An event bus is the only Custom Event Bus resource that takes a resource policy; EventBridge rejects any other resource ARN. A bus holds two named policies. The `default` policy is the one you write, in full IAM policy language, including `Deny` statements. `AWS_RAM` is written only by AWS RAM and reflects your resource shares.

EventBridge evaluates both when it authorizes a caller from another account, and an explicit `Deny` in either overrides an `Allow` in the other. When a cross-account caller is denied and the policy you wrote appears to allow them, read the other policy.

The following `default` policy on the `orders` bus lets account `444455556666` attach subscribers and read the bus, and nothing else.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPartnerAccountToSubscribe",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::444455556666:root" },
            "Action": [
                "events:CreateSubscriber",
                "events:DescribeEventBus"
            ],
            "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef"
        }
    ]
}
```

Manage the policies with `PutResourcePolicy`, `GetResourcePolicy`, `DeleteResourcePolicy`, and `ListResourcePolicies`. Each takes the bus ARN in `--resource-arn`. The first three act on `default` when you omit `--policy-name`. You cannot write or delete `AWS_RAM`: naming it fails with `AccessDeniedException`. `GetResourcePolicy` fails with `ResourceNotFoundException` when the named policy does not exist, rather than returning an empty document, so treat that as no policy. A bus that you shared only through AWS RAM has no `default` policy, so `get-resource-policy` without `--policy-name` returns that error; pass `--policy-name AWS_RAM` to read the policy AWS RAM wrote. `ListResourcePolicies` returns policy names and revision IDs only, never documents. To make a write or delete conditional, pass

the policy's current revision in `--expected-revision-id`; on a mismatch the operation fails with `ConflictException`. A `default` policy document can be up to 20 KB; a larger policy fails with `PolicyLengthExceededException`. This quota is adjustable in Service Quotas.

EventBridge rejects a `default` policy that would grant public access, with `PublicPolicyException`, and does not attach it. This check is always on. A statement grants public access when its principal is a wildcard and no condition limits the callers to specific accounts or principals. A condition on event content, such as `events:source`, does not identify the caller. A condition such as `aws:PrincipalOrgID` does. A policy you write is invisible to AWS RAM, so you own tracking it.

Events that arrive without a publish call

Events also arrive on a bus without a direct `PutEvents` or `PutRawEvents` call: from an event source, or from a rule or a subscriber on another bus whose target is this bus. EventBridge authorizes an event source once, when you create it. It checks nothing again as events arrive, so a bus policy cannot filter that traffic afterwards. For more information about event sources, see the section called “Event sources”. A rule or subscriber that you point at a bus yourself is authorized at ingestion, under the role you supplied, once for every event. EventBridge chooses the action per event: `events:PutEvents` for a genuine AWS service or partner event, and `events:PutRawEvents` for an event you originated. You cannot choose the action used. A forwarder carrying both kinds of events through one path must be granted both actions, in the role's identity policy and in the destination bus's resource policy. Granting only one denies half the stream.

Publishing events to a Custom Event Bus

You publish events to a Custom Event Bus with one of two API operations, `PutEvents` or `PutRawEvents`. The operation that you choose fixes the shape of the delivered event, and that shape determines how you write every filter, transformer, and target parameter for the bus. Choose before you write your first subscriber. For the two shapes, see the section called “Event structure”.

The two operations produce one kind of event, and EventBridge treats them identically once the event is on the bus. `PutEvents` is a convenience over `PutRawEvents`: EventBridge takes the entry's `Source`, `DetailType`, `Detail`, and other fields, builds a JSON envelope from them, and stores that envelope as a JSON payload with a content type of `application/eventbridge+json`. From then on, filtering, transformation, ordering, deduplication, retention, and delivery work exactly as they do for a JSON event published with `PutRawEvents`. The only difference

a subscriber sees is where your fields sit: at the top of `Data` for `PutRawEvents`, and under `Data.detail` inside the envelope for `PutEvents`. For Avro, Protobuf, and opaque payloads, see the section called “Supported content types”.

	PutEvents	**PutRawEvents**
Payload field	`Detail`, a JSON string	`Data`, bytes in any format
Required in each entry	`Source`, `DetailType`	`Data`, `SystemMetadata.ContentType`
Content type	Set by EventBridge to `application/eventbridge+json`	Set by you: `application/json`, `application/avro` , `application/protobuf` , or `application/octet-stream` . See the section called “Supported content types”.
Your own metadata	Not available	`Metadata`, a map of up to 100 keys
Ordering and deduplication inputs	`SystemMetadata.EventGroupId` , `SystemMetadata.DeduplicationId`	The same two fields
Use when	Your events are JSON and you want the same envelope that Custom Event Bus - Classic produces	Your payload is binary or non-JSON, you need your own metadata keys, or you want the payload delivered exactly as sent

Publish APIs

Both operations take a batch of 1 to 100 entries per request and return one result per entry, in request order. EventBridge accepts or rejects each entry on its own: a rejected entry carries an

`ErrorCode` and `ErrorMessage` in place of a `SuccessCode`, and the other entries in the same request are unaffected. Batch entries that belong together, and read every result rather than the HTTP status alone; see the section called “Reading the response”. Both calls draw from one publish rate budget for your account, and the bus itself has a per-second ingestion quota that counts every publisher and event source; see the section called “Custom Event Bus quotas”.

Publish a JSON event with PutEvents

```
aws eventsv2 put-events \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --entries '[
        {
            "Source": "com.example.orders",
            "DetailType": "OrderPlaced",
            "Detail": "{\"orderId\":\"1001\",\"total\":42.5}"
        }
    ]'
```

Publish a raw event with PutRawEvents

`Data` is a binary field. In the AWS CLI, supply it as base64 text. The following example publishes the same JSON payload as raw bytes with a content type of `application/json`, and adds a metadata key.

```
aws eventsv2 put-raw-events \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --entries '[
        {
            "Data": "eyJvcmRlcklkIjoiMTAwMSIsInRvdGFsIjo0Mi41fQ==",
            "Metadata": { "tenant": "acme" },
            "SystemMetadata": { "ContentType": "application/json" }
        }
    ]'
```

To publish Avro or Protobuf, name a schema registry in the request with `--schema-registry-configuration`. `RegistryUri` is an AWS Glue Schema Registry ARN or a Confluent Cloud HTTPS URL. A registry is named per request, not on the bus. Events with a content type of `application/octet-stream` need no registry, because EventBridge does not decode them.

Publishing raw bytes

To publish a payload that EventBridge should neither parse nor inspect, call `PutRawEvents` with the bytes Base64-encoded in `Data` and `SystemMetadata.ContentType` set to `application/octet-stream`. EventBridge stores and delivers the bytes exactly as sent, so a `DATA` filter cannot match them; route them with `METADATA` keys you set on the entry or with `SYSTEM_METADATA`. A subscriber's transformer sees the payload as a Base64 string in `$events.Data`. For the delivery choices and the deserialized alternative for Avro and Protobuf, see the section called “Supported content types”; for how the payload appears to a transformer, see the section called “Input transformations”.

Reading the response

Both operations return one result for each entry, in the same order as the request. Each result carries either a `SuccessCode` or an `ErrorCode` and `ErrorMessage`. Check every entry, because a request can succeed while individual entries fail.

`SuccessCode` is `PUBLISHED` or `DEDUPLICATED`. `DEDUPLICATED` means that EventBridge recognized the entry as a duplicate of an event that it already accepted and did not store a second copy. It is a success, not an error. For more information, see the section called “Ordering and deduplication”.

A publish response reports that EventBridge accepted the event. It does not report delivery. To confirm that an event arrived, check the target.

This section contains the following topics:

- the section called “Supported content types”
- the section called “Event sources”
- the section called “Event structure”

Supported content types: JSON, Avro, Protobuf, and raw bytes

`PutRawEvents` accepts any bytes. Each entry's `SystemMetadata.ContentType` tells EventBridge what the bytes are. That value decides whether EventBridge deserializes the payload before it filters and delivers the event. For Avro and Protobuf, EventBridge deserializes the payload against a schema registry that you name on the request. Subscribers then filter, transform, and receive the event as JSON. For opaque bytes, EventBridge delivers the payload untouched.

ContentType	What EventBridge does with the payload	Schema registry
`application/json`	Filters and transforms on the JSON content	Not used
`application/avro`	Deserializes the Avro binary to JSON, then filters and transforms on that JSON	Required
`application/protobuf`	Deserializes the Protobuf binary to JSON, then filters and transforms on that JSON	Required
`application/octet-stream`	Does not deserialize or inspect the payload. Delivers the bytes as sent	Not used

How EventBridge deserializes a payload

When an entry's `ContentType` is `application/avro` or `application/protobuf`, EventBridge performs the following steps for that entry.

1. EventBridge reads the schema identifier that the registry's serializer wrote into the payload bytes.
2. EventBridge fetches the matching schema from the registry you name in `SchemaRegistryConfiguration.RegistryUri`, and deserializes the payload to JSON. If EventBridge cannot find or read the schema, or cannot deserialize the payload against it, the entry fails with a per-entry error. The other entries in the request are unaffected.
3. If a subscriber defines filters, EventBridge evaluates them against the deserialized JSON.
4. EventBridge applies the subscriber's transformation to the deserialized JSON and delivers JSON to the target.

You name the registry on each `PutRawEvents` request, in `SchemaRegistryConfiguration.RegistryUri`. The setting belongs to the request, not to

the bus, and EventBridge ignores it for JSON entries in the same request. EventBridge reads the registry with the caller's credentials, so the identity that publishes needs read access to the registry.

After deserializing, the event behaves like any JSON event published with `PutRawEvents`. A filter with a scope of `DATA` matches fields of the deserialized record by their schema names. A JSONata transformer addresses them under `$events.Data`, and a target receives JSON. EventBridge also sets three keys in the delivered `SystemMetadata`.

- `aws:SchemaId`: the identifier of the schema EventBridge deserialized with; a UUID for AWS Glue, an integer for Confluent.
- `aws:RegistryType`: the registry that decoded the event, `Glue` or `Confluent`. Events published with `PutEvents` never carry it.
- `ContentType`: the content type of the published entry, for example `application/avro`.

Choosing how EventBridge delivers your payload

`ContentType` selects one of two delivery paths.

- **Deserialize to JSON.** Set `ContentType` to `application/avro` or `application/protobuf`. EventBridge deserializes the payload against your schema registry, and subscribers filter on `DATA`, transform with JSONata, and receive JSON. Use this path when you want content-based routing and reshaping of schema-encoded events.
- **Deliver bytes untouched.** Set `ContentType` to `application/octet-stream`. EventBridge does not deserialize or inspect the payload, no schema registry is involved, and the subscriber receives the exact bytes you sent. Use this path when you deserialize the payload yourself at the target, or when the content is a format EventBridge does not deserialize.

Deserializing with the AWS Glue Schema Registry

Set `RegistryUri` to the registry ARN, of the form `arn:aws:glue:`*`region`*`:`*`account-id`*`:registry/`*`name`*. Before publishing, encode the payload with the AWS Glue Schema Registry serializer for your language. The serializer embeds the schema reference that EventBridge uses to resolve and deserialize the event.

EventBridge reads the schema as the publishing caller, so grant the publishing identity read access to the registry and its schemas. The following policy grants the minimum permission, scoped to one registry.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "glue:GetSchemaVersion",
            "Resource": [
                "arn:aws:glue:us-east-1:111122223333:registry/orders",
                "arn:aws:glue:us-east-1:111122223333:schema/orders/*"
            ]
        }
    ]
}
```

Publish the schema-encoded bytes with the Glue registry ARN in `RegistryUri`.

```
aws eventsv2 put-raw-events \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --schema-registry-configuration RegistryUri=arn:aws:glue:us-
east-1:111122223333:registry/orders \
    --entries '[
        {
            "Data": "base64-encoded-avro-bytes",
            "SystemMetadata": { "ContentType": "application/avro", "EventGroupId":
 "order-1001" }
        }
    ]'
```

A subscriber then filters on the deserialized record. If the Avro schema has a field named `orderId`, the following filter matches one order.

```
{ "Scope": "DATA", "Pattern": "{\"orderId\":[\"1001\"]}" }
```

Deserializing with a Confluent Schema Registry

Set `RegistryUri` to the registry's HTTPS URL. An HTTPS registry also requires `SchemaRegistryConfiguration.ConfluentPublicRegistryConfiguration.ConnectionArn`, the ARN of an EventBridge connection that holds the registry's API key or OAuth credentials. The connection must belong to the calling account; a cross-account connection is rejected. Create the connection before the first publish. Before publishing, encode the payload with the Confluent

serializer for your language. The serializer embeds the schema reference that EventBridge uses to resolve and deserialize the event.

EventBridge reads the registry credentials through the connection you name. Grant the publishing identity permission to read that connection's credentials.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "events:RetrieveConnectionCredentials",
            "Resource": "arn:aws:events:us-east-1:111122223333:connection/confluent-
registry/*"
        },
        {
            "Effect": "Allow",
            "Action": ["secretsmanager:GetSecretValue",
 "secretsmanager:DescribeSecret"],
            "Resource": "arn:aws:secretsmanager:us-east-1:111122223333:secret:events!
connection/confluent-registry/*",
            "Condition": { "Bool": { "aws:ViaAWSService": "true" } }
        }
    ]
}
```

If the connection secret is encrypted with a customer managed AWS KMS key, also grant `kms:Decrypt` on that key.

Publish the schema-encoded bytes with the Confluent URL and connection ARN.

```
aws eventsv2 put-raw-events \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --schema-registry-configuration '{
        "RegistryUri": "https://psrc-example.us-east-1.aws.confluent.cloud",
        "ConfluentPublicRegistryConfiguration": {
            "ConnectionArn": "arn:aws:events:us-east-1:111122223333:connection/
confluent-registry/EXAMPLE-1234-5678-abcd"
        }
    }' \
    --entries '[
        {
```

```
            "Data": "base64-encoded-protobuf-bytes",
            "SystemMetadata": { "ContentType": "application/protobuf" }
        }
    ]'
```

Connection authentication options

Configure the connection with the authentication type the Confluent registry expects. EventBridge reads only the authentication headers the connection produces; it never sees the raw secret, and it refreshes OAuth tokens automatically.

API key. Confluent Schema Registry typically uses HTTP Basic authentication with the schema registry API key as the user name and the API secret as the password.

```
aws events create-connection \
    --name confluent-registry \
    --authorization-type API_KEY \
    --auth-parameters '{
        "ApiKeyAuthParameters": {
            "ApiKeyName": "Authorization",
            "ApiKeyValue": "Basic base64(srApiKey:srApiSecret)"
        }
    }'
```

OAuth client credentials. EventBridge fetches and refreshes the token. You must include `grant_type=client_credentials` as a body parameter; EventBridge does not add it for you.

```
aws events create-connection \
    --name confluent-registry-oauth \
    --authorization-type OAUTH_CLIENT_CREDENTIALS \
    --auth-parameters '{
        "OAuthParameters": {
            "AuthorizationEndpoint": "https://idp-domain/oauth/token",
            "HttpMethod": "POST",
            "ClientParameters": {
                "ClientID": "oauth-client-id",
                "ClientSecret": "oauth-client-secret"
            },
            "OAuthHttpParameters": {
                "BodyParameters": [
                    { "Key": "grant_type", "Value": "client_credentials",
 "IsValueSecret": false }
```

```
                ]
            }
        }
    }'
```

EventBridge stores either credential in a managed Secrets Manager secret (`events!connection/...`), which the connection reads on your behalf.

Delivering bytes without deserializing

To have EventBridge deliver a payload exactly as sent, set the entry's `SystemMetadata.ContentType` to `application/octet-stream`. This is a field in each entry, not an HTTP header. EventBridge does not deserialize or inspect the bytes, and no schema registry is involved. Any format works, including Avro or Protobuf that you deserialize yourself at the target.

Because the content is opaque, a `DATA` filter cannot match it. Route such events with filters on `METADATA`, using keys you set in the entry's `Metadata` map, or on `SYSTEM_METADATA`. Content-based deduplication still works and hashes the bytes exactly as you sent them. In a transformer, the payload appears as a Base64 string; see the section called “Input transformations”.

Filtering and transforming deserialized events

Once EventBridge deserializes an event to JSON, the payload flows through the normal pipeline. A filter with a scope of `DATA` matches the deserialized fields by their schema names, and a JSONata transformer addresses them under `$events.Data`. An event published as Avro or Protobuf gets the same content-based routing and per-subscriber reshaping as an event published as plain JSON. Write filter patterns against the deserialized JSON, keeping two conversion rules in mind.

- **Avro** deserializes to standard JSON. The conversion is one-way: the JSON cannot be converted back to an Avro object.
- **Protobuf** deserializes to JSON with the field names defined in your schema, rather than converting them to camel case as Protobuf tooling typically does. Match on the schema field names.

Deduplicating events across formats

EventBridge supports two deduplication modes: ID-based, where you supply a `DeduplicationId` on the entry, and content-based, where EventBridge computes a SHA-256 hash of the event

content. For deserialized events, content-based deduplication hashes the original Avro or Protobuf wire bytes, not the deserialized JSON. Two events are duplicates only when their original payloads are byte-identical. EventBridge therefore does not deduplicate the same logical event published in different formats. Avro bytes, Protobuf bytes, and raw JSON differ on the wire even when they deserialize to the same data. See the section called “Ordering and deduplication”.

Errors

- An Avro or Protobuf entry published without `SchemaRegistryConfiguration` fails as a per-entry error with an `ErrorCode` of `VALIDATION_ERROR` and a message that states `SchemaRegistryConfiguration is required`. The other entries in the request are unaffected.
- A `ConnectionArn` that does not name a usable connection, or that belongs to another account, fails the entry with `VALIDATION_ERROR` and a message that names the registry.
- A schema that cannot be found or read, or a payload that cannot be deserialized against it, fails that entry with `VALIDATION_ERROR`. This error is not retryable; fix the request or the caller's registry access before republishing.
- A registry that cannot be reached fails the request with `SchemaRegistryUnavailableException`. On the Confluent OAuth path, this can occur while the registry access token is being refreshed. This error is transient; retry the request with backoff.

For per-entry results and retry guidance in general, see the section called “Publishing events”.

Event sources for a Custom Event Bus

An event source ingests events from an AWS service or a SaaS partner onto a Custom Event Bus. You create it with `CreateEventSource`, naming a destination bus and one origin, for example `aws.s3`, and EventBridge delivers events from the origin to the bus. Your own applications publish to the bus directly, so you need an event source only for events you do not produce yourself.

Ingested events arrive on the destination bus in the EventBridge envelope, with content type `application/eventbridge+json`. The event data holds the envelope fields: `source`, `detail-type`, `account`, `time`, `region`, `resources`, and `detail`. EventBridge also sets the `aws:Source` and `aws:DetailType` system metadata fields from the envelope's `source` and `detail-type`. See the section called “Event structure”.

How the events are moved

When you create an event source, EventBridge creates a managed rule and a target in your account, on your Custom Event Bus - Classic, and they carry events from the origin to your Custom Event Bus. Both appear in your account with `ManagedBy` set to `events.amazonaws.com`. You do not manage them; deleting or revoking the event source removes them. The events therefore still appear in the Custom Event Bus - Classic's metrics.

AWS service event sources

An AWS service event source ingests the events that one AWS service, such as `aws.s3` or `aws.ec2`, emits for your account. By default every event from that service arrives on the bus; to narrow that down, add a `Pattern`. The following example creates an event source that admits only Amazon S3 Object Created events.

```
aws eventsv2 create-event-source \
    --name s3-object-created \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --configuration '{"AwsServiceEventsConfiguration":
{"AwsService":"aws.s3","Pattern":"{\"detail-type\":[\"Object Created\"]}"}}'
```

The event source's pattern decides what crosses onto the bus. A subscriber's filter decides what that subscriber receives from the events flowing through the bus; see the section called “Filtering”. Write the pattern in EventBridge event pattern syntax (*Event patterns*), and EventBridge evaluates it against the envelope. Leave the `source` key and the top-level `account` and `region` keys out of the pattern: EventBridge adds those conditions itself, from your account and Region and the service you selected, and rejects a pattern that supplies them.

Partner event sources

A partner event source ingests the events a SaaS partner sends to your account. Before you create one, the partner must offer a partner event source to your account (see the section called “SaaS partner event sources”), and that source must be in the `PENDING` state. The following example creates an event source for it.

```
aws eventsv2 create-event-source \
    --name example-partner-events \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
```

```
    --configuration '{"PartnerEventsConfiguration":
{"PartnerEventSourceArn":"arn:aws:events:us-east-1::event-source/aws.partner/
example.com/12345"}}'
```

When you create the event source, EventBridge creates a managed partner event bus (a Custom Event Bus - Classic) in your account to activate the partner source. When you delete the event source, EventBridge deletes that bus and returns the partner source to `PENDING`. To encrypt that bus with a customer managed key, set `PartnerBusKmsKeyIdentifier` to a key in your account and Region; otherwise EventBridge uses an AWS owned key.

Dead-letter queue

To capture events that fail to reach the bus, set `OnFailureConfiguration.Arn` to a standard Amazon SQS queue in your account and Region; a FIFO queue is rejected. Messages arrive on the queue from the `events.amazonaws.com` service principal, not from a role you supply, so grant that principal `sqs:SendMessage` in the queue's resource policy.

```
{
    "Effect": "Allow",
    "Principal": { "Service": "events.amazonaws.com" },
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:us-east-1:111122223333:my-event-source-dlq",
    "Condition": { "StringEquals": { "aws:SourceAccount": "111122223333" } }
}
```

A partner event source sends to the queue under two source ARNs: the managed rule, for delivery failures, and the managed partner event bus, for encryption failures. A policy that conditions on `aws:SourceArn` must allow both.

Permissions

To create an event source on a bus in another account, the bus owner must grant you `events:CreateEventSource`, through an AWS RAM share (`AWSRAMEventBridgeEventBusV2EventSourceAccess` or `FullAccess`) or with a `PutResourcePolicy` statement on the bus. The `events:source` condition key carries the AWS service name or the partner event source name, so a bus owner can allow or deny specific origins. See the section called “Sharing a bus”.

To encrypt the managed partner event bus, your identity needs `kms:DescribeKey` and `kms:Decrypt` on the key you name in `PartnerBusKmsKeyIdentifier`; without them, `CreateEventSource` fails with `AccessDeniedException`.

```
{
    "Effect": "Allow",
    "Action": ["kms:DescribeKey", "kms:Decrypt"],
    "Resource": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
}
```

If another account owns the destination bus and encrypts it with a customer managed key, that key's policy must also admit your account; otherwise create, update, and describe of the event source fail.

Subscribing to ingested events

To match AWS service or partner events on a subscriber, filter on the `aws:Source` system metadata field. EventBridge writes this field and no publish call can set a value that begins with `aws.`, so a match on its value is the only match that proves the origin of an event. For AWS service events, match the service name; for partner events, match the partner event source name.

```
{ "Scope": "SYSTEM_METADATA", "Pattern": "{\"aws:Source\":[\"aws.s3\"]}" }

{ "Scope": "SYSTEM_METADATA", "Pattern": "{\"aws:Source\":[{\"prefix\":\"aws.partner/
example.com\"}]}" }
```

To filter on the event content, add a `DATA` filter. The envelope carries the content under `detail`.

```
{ "Scope": "DATA", "Pattern": "{\"detail\":{\"bucket\":{\"name\":[\"amzn-s3-demo-bucket
\"]}}}" }
```

The event data also carries `source` and `detail-type`, and any publisher can write those values into a payload, so they do not prove the origin. Match `aws:Source` by value, not by presence: a direct `PutEvents` call also sets `aws:Source`, from its own `Source`, which never starts with `aws.`.

Permanently stopping an event source

The bus owner can permanently stop an event source from ingesting by calling `RevokeResource` with its ARN; see the section called “Revoking”. A revoked event source cannot be restored.

Revocation covers the one resource you name, so pair it with a bus resource policy that stops the same account from creating another. To block the account from creating any event source on the bus:

```
{
    "Effect": "Deny",
    "Principal": { "AWS": "arn:aws:iam::444455556666:root" },
    "Action": "events:CreateEventSource",
    "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef"
}
```

To block one origin while allowing others, condition the deny on `events:source`.

```
{
    "Effect": "Deny",
    "Principal": { "AWS": "arn:aws:iam::444455556666:root" },
    "Action": "events:CreateEventSource",
    "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
    "Condition": { "StringEquals": { "events:source": "aws.partner/
example.com/12345" } }
}
```

Confirming that events cross onto the bus

Two sets of metrics show whether ingested events reach the Custom Event Bus. The managed rule reports the usual rule metrics in your account, under `AWS/Events`. The destination bus's publish metrics carry an `EventSource` dimension, which gives a per-event-source breakdown of inbound traffic; see the section called “Observability”.

When a subscriber carries events from one Custom Event Bus to another across accounts, AWS service and partner events keep their `aws:Source` and `aws:DetailType` as long as the event's structure and content stay intact. EventBridge authorizes their arrival on the destination bus with `events:PutEvents`, so the destination bus's resource policy must grant that action to the forwarding role. Events your applications published take `events:PutRawEvents`, so a stream that mixes both needs both actions; see the section called “Event bus”.

Event structure: data, metadata, and system metadata

Filters, transformers, and target parameters share one addressing model. EventBridge delivers every event under three top-level names: `Data` holds the payload, `Metadata` holds the keys that you supplied with `PutRawEvents`, and `SystemMetadata` holds the fields that EventBridge assigns. Where your own fields sit inside `Data` depends on the publish API. Both APIs produce the same kind of event, and EventBridge filters and transforms both the same way. The difference is only what sits inside `Data`.

Part	Published with PutRawEvents	Published with PutEvents
`Data`	Your payload, exactly as sent	An envelope with `source`, `detail-type` , `time`, and other fields. Your payload is the value of `detail`.
`Metadata`	Your metadata keys	Not present
`SystemMetadata`	Fields assigned by EventBridge	Fields assigned by EventBridge

A JSONata expression sees the event as `$events`. The same order identifier is at a different path for each API.

Published with `PutRawEvents`:

```
$events.Data.orderId
```

Published with `PutEvents`:

```
$events.Data.detail.orderId
```

The same difference applies to a filter with a scope of `DATA`. The pattern is written against the contents of `Data`.

Published with `PutRawEvents`:

```
{ "total": [ { "numeric": [ ">", 500 ] } ] }
```

Published with PutEvents:

```
{ "detail": { "total": [ { "numeric": [ ">", 500 ] } ] } }
```

Important

An expression or pattern written for the wrong shape does not fail. It resolves to nothing, the filter matches no events, and the transformer produces no output. A subscriber whose filter matches nothing looks identical to a subscriber that is not working. Before you rely on a subscriber, publish one event that should match and one that should not, and confirm both outcomes at the target. If producers publish with both APIs to the same bus, give each kind of traffic its own subscribers.

SystemMetadata fields

SystemMetadata is the same for both APIs and is the only part of the event that EventBridge itself writes. A filter with a scope of SYSTEM_METADATA matches against it, and a JSONata expression reads it as $events.SystemMetadata. Every value is a string. EventBridge omits a field that has no value, so a JSONata reference to it evaluates as undefined; only aws:DeliveryType is always present. The following table lists the fields.

Field	Value
aws:EventId	The identifier EventBridge assigned, returned to the publisher as EventId
aws:Source , aws:DetailType	The entry's Source and DetailType for PutEvents ; the originating service or partner and its event type for an event source. No publish call can set a source that begins with aws., so {"aws:Source":["aws.s3"]} proves that Amazon S3 produced the event

Field	Value
aws:DeliveryType	LIVE for an event delivered as it arrived, REPLAY for one read from retained history. See the section called “Replaying events”
EventGroupId , DeduplicationId	As supplied at publish time. See the section called “Ordering and deduplication”
ContentType	The content type as published: application/eventbridge+json for PutEvents ; for PutRawEvents , the entry's value, so Avro and Protobuf data that EventBridge decoded to JSON still shows application/avro or application/protobuf
aws:IngestionTime	When the bus accepted the event
aws:SequenceNumber	The event's position on the bus; increases with ingestion order
aws:SchemaId , aws:RegistryType	The schema and registry used to deserialize an Avro or Protobuf payload; absent otherwise

Metadata is present only on events published with PutRawEvents that set it; for every other event it is empty. For application/octet-stream, Data is a Base64 string. In a Transformer expression $events is one event; only in a universal target's Input is it an array. See the section called “Input transformations”.

Subscribing to events on a Custom Event Bus

A subscriber is one resource that says which events you want and where to send them. You create it with CreateSubscriber, naming the bus, the filters, and one target. It watches that bus, keeps the events that match its filters, optionally transforms each one, and delivers it to exactly one target. To fan an event out to several targets, create one subscriber for each target on the same bus.

EventBridge applies one subscriber create or delete at a time on a given bus, whichever account calls. A second `CreateSubscriber` or `DeleteSubscriber` on the same bus while one is in progress fails with `ConcurrentModificationException`; retry it. The create and delete rates in the section called "Custom Event Bus quotas" therefore apply per bus.

Filtering which events reach the target

`FilterConfiguration` holds a list of `Filters`. Each filter has a `Pattern`, in EventBridge event pattern syntax, and one `Scope`. You can have at most one filter per scope, and EventBridge delivers an event only if it matches every filter in the list. A subscriber without a `FilterConfiguration` receives every event on the bus.

- `DATA`: matches the payload. Write the pattern against the shape described in the section called "Event structure".
- `METADATA`: matches the key-value pairs a producer set with `PutRawEvents`, exact match only.
- `SYSTEM_METADATA`: matches the fields EventBridge sets, such as `aws:Source` or `EventGroupId`.

For the pattern syntax, the scope rules, and examples, see the section called "Filtering".

Transforming the payload

`Transformer.Type` selects what the target receives. A universal target does not use it; see the section called "Universal targets".

- `RAW`, the default. The payload alone.
- `WITH_METADATA`. The full event: `Data`, `Metadata`, and `SystemMetadata`.
- `JSONATA`. The result of the JSONata expression in `JsonataConfiguration.Expression`, wrapped in `{% %}` delimiters, with the event available as `$events`.

EventBridge checks the syntax of an expression when you create the subscriber and evaluates it at delivery time. An expression that throws, or that produces no value, fails the delivery of that event. Test expressions against real events before you depend on them. For what `$events` holds, the extra functions, and the limits, see the section called "Input transformations".

Delivering to one target

`InvokeConfiguration` carries the `TargetArn`, the `RoleArn` that EventBridge assumes to invoke the target, and at most one parameter block that matches the target type. Most parameter values accept a JSONata expression, so a value can come from the event. The following table lists the bespoke target types and their parameter block; each has its own page under the section called "Targets".

Target	Parameter block
Amazon SQS queue	`SqsParameters`
Lambda function	`LambdaParameters`
Kinesis stream	`KinesisParameters`
Step Functions state machine	`StepFunctionsParameters`
Amazon SNS topic	`SnsParameters`
HTTP endpoint or API destination	`HttpParameters`
Firehose stream	`FirehoseParameters`

An Custom Event Bus or a Custom Event Bus - Classic can also be the target: set the bus ARN as `TargetArn`. For a FIFO target such as an Amazon SQS FIFO queue, set the target's own deduplication identifier in the parameter block, for example `SqsParameters.MessageDeduplicationId`, to a per-message value derived from the event with a JSONata expression. A constant makes the target discard every message after the first.

A *universal target* invokes an AWS API action directly, with no function in between. Its ARN has the form `arn:aws:events:::aws-sdk:`*`service`*`:`*`apiAction`*, for example `arn:aws:events:::aws-sdk:dynamodb:putItem`, and its `UniversalTargetParameters.Input` holds the request as JSON or as a JSONata expression that produces it. For the naming rules, batching, and failure behavior, see the section called "Universal targets".

When you create the subscriber, EventBridge resolves the target type from the ARN and checks three things: that the parameter block matches the target type, that the ARN belongs to your

account, and that every expression parses. Whether the target exists, whether the role can reach it, and whether the target's API accepts the values are checked at delivery, so confirm the first delivery at the target.

Retries and dead-letter queues

`RetryPolicy` sets how long EventBridge retries a failed delivery, by default 5 attempts within 300 seconds, and `OnFailureConfiguration.Arn` names the Amazon SQS queue that receives a record for each event that exhausted its retries. Set both when you create the subscriber. For the limits, the record format, and how to redrive, see the section called “Retries and dead-letter queues”.

The delivery role

The role in `RoleArn` must belong to your account and trust the `events.amazonaws.com` service principal. Your caller needs `iam:PassRole` on the role to create the subscriber, or EventBridge denies the request before it checks the target ARN. The role's permissions policy needs two grants.

- The action the target requires, for example `sqs:SendMessage` on a queue, `lambda:InvokeFunction` on a function, `sns:Publish` on a topic, or `kinesis:PutRecords` on a stream.
- `sqs:SendMessage` on the dead-letter queue in `OnFailureConfiguration.Arn`. Without it, a failed delivery has nowhere to record itself, and those events are lost with no signal.

The following trust policy allows EventBridge to assume the role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "events.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

The following permissions policy delivers to an Amazon SQS queue and dead-letters to a second queue.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "InvokeTheTarget",
            "Effect": "Allow",
            "Action": "sqs:SendMessage",
            "Resource": "arn:aws:sqs:us-east-1:111122223333:large-orders"
        },
        {
            "Sid": "WriteToTheDeadLetterQueue",
            "Effect": "Allow",
            "Action": "sqs:SendMessage",
            "Resource": "arn:aws:sqs:us-east-1:111122223333:large-orders-dlq"
        }
    ]
}
```

Example: deliver large orders to a queue

The following command creates a subscriber that delivers orders over 500 to an Amazon SQS queue, for events published with `PutEvents`, and sends failures to a dead-letter queue.

```
aws eventsv2 create-subscriber \
    --name large-orders \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --filter-configuration '{
        "Filters": [
            { "Scope": "DATA", "Pattern": "{\"detail\":{\"total\":[{\"numeric\":[\">
\",500]}]}}" }
        ]
    }' \
    --invoke-configuration '{
        "TargetArn": "arn:aws:sqs:us-east-1:111122223333:large-orders",
        "RoleArn": "arn:aws:iam::111122223333:role/EventBusDeliveryRole"
    }' \
    --on-failure-configuration '{ "Arn": "arn:aws:sqs:us-east-1:111122223333:large-
orders-dlq" }'
```

Troubleshooting: you published and nothing arrived at the target

A publish response says only that the bus accepted and stored the event; it never confirms delivery. Work through the following checks in order.

Symptom

The publish call returned HTTP 200 with no failed entries, and the target has received nothing after several minutes.

Cause

One of five things, in order of frequency: delivery is still being retried; the subscriber is `STOPPED` or revoked; the subscriber's `StartingPosition` is `LATEST` and the event was published before the subscriber existed; the filter is written for the other publish API and matches nothing; or the delivery role cannot reach the target or the dead-letter queue. `TargetInvocations` tells you which half of the problem you have. Datapoints on `TargetInvocations` with none on `EventsDelivered` mean that deliveries are attempted and fail. No datapoints on `TargetInvocations` while `FilterEvaluated` has them means that no delivery is attempted; check that the subscriber is `RUNNING`, and check `EventTransformationFailures`.

Solution

1. Read the subscriber. `State` must be `RUNNING` and `Revoked` must be absent; it appears only when true. Note `StartingPosition` and the filters.

   ```
   aws eventsv2 describe-subscriber \
       --subscriber-arn arn:aws:events:us-east-1:111122223333:subscriber/large-
   orders/EXAMPLE1234567890abcdef
   ```

2. Read the subscriber's metrics in `AWS/EventsV2`. `FilterEvaluated` at zero means no event reached this subscriber: the producer published to another bus, or before the subscriber existed with `LATEST`. `FilterEvaluated` above zero with `FilterMatched` at zero means the filter matched nothing; see the section called “Event structure”. When `FilterMatched` has datapoints and `EventsDelivered` does not, deliveries are failing, and which metric records the outcome depends on the subscriber: `EventsDropped` when no dead-letter queue is configured, `OnFailureDestinationDelivered` when the event reached the queue, and `OnFailureDestinationFailed` when it did not.
3. Turn on logs and reproduce. Set `LogConfiguration.Level` to `INFO` and `IncludePayload` to `FULL`, wire a log delivery, publish again, and read the

`EVENT_DELIVERY_ATTEMPT` records. Each carries the target's own error code and message, and `details.target_input`, which is what EventBridge sent rather than what you meant to send. See the section called “Observability”.

4. If the filter is the suspect, add a diagnostic subscriber on the same bus with no filter, `Transformer.Type` `WITH_METADATA`, and a queue you own as the target. It delivers each event exactly as EventBridge sees it, as `Data`, `Metadata`, and `SystemMetadata`. Then simplify your filter one condition at a time; the last condition you remove is the cause.

Verification

Publish one event that should match and one that should not. The first arrives at the target and `EventsDelivered` increments by one; the second does not arrive and `FilterEvaluated` increments while `FilterMatched` does not.

> **Note**
>
> A subscriber's metrics are available to the bus owner and to the subscriber's owner; its logs are delivered only to the subscriber's owner.

Targets for a Custom Event Bus subscriber

A subscriber delivers to one target. Set `InvokeConfiguration.TargetArn` to the target's ARN, add the parameter block for that target type, for example `SqsParameters` for a queue, and give the delivery role the action the target needs, for example `sqs:SendMessage`. Every parameter value can be a fixed string or a JSONata expression in `{% %}` that reads the event, so a queue's `MessageGroupId` can be `{% $events.SystemMetadata.EventGroupId %}`.

The following pages describe each target type. For an AWS API action that has no bespoke target, use a universal target.

- the section called “Amazon SQS queue”
- the section called “Lambda function”
- the section called “Amazon SNS topic”
- the section called “Kinesis stream”
- the section called “Firehose stream”
- the section called “Step Functions state machine”

- the section called “HTTP and API destinations”
- the section called “Event bus”
- the section called “Universal targets”

Amazon SQS queue target

To deliver to an Amazon SQS queue, set `TargetArn` to the queue ARN, for example `arn:aws:sqs:us-east-1:111122223333:large-orders`, and add `SqsParameters`. For a FIFO queue, set `MessageGroupId`; the usual value is the event's group, `{% $events.SystemMetadata.EventGroupId %}`.

Parameters

Parameter	What it sets
`MessageGroupId`	The queue message group. Required for a FIFO queue
`MessageDeduplicationId`	The queue deduplication identifier for a FIFO queue without content-based deduplication. Derive it from the event; a constant makes the queue discard every message after the first
`DelaySeconds`	Delivery delay for the message
`MessageAttributes`	Message attributes to attach
`MessageSystemAttributes`	System attributes, such as the AWS X-Ray trace header

Delivery role

The role needs `sqs:SendMessage` on the queue. If the queue is encrypted with a customer managed key, the role also needs `kms:GenerateDataKey` and `kms:Decrypt` on that key.

Batching and payload

EventBridge calls `SendMessageBatch`, so one target call carries up to 10 events, and one queue message carries one event. `Transformer` decides the body of each message. See the section called “Batching”.

Lambda function target

To deliver to a Lambda function, set `TargetArn` to the function ARN, optionally with an alias or version, and add `LambdaParameters`. The shape of the payload depends on `BatchConfiguration.MaxBatchSize`. With `MaxBatchSize` set to 1, the function receives the event as a single JSON object. With any other value, or with no `BatchConfiguration`, the function receives a JSON array holding the events in the batch, one element per event, even when the batch holds one event.

Parameters

Parameter	What it sets
`InvocationType`	`EVENT` invokes asynchronously and treats a successful `Invoke` as delivered. `REQUEST_RESPONSE` waits for the function to return and treats a function error as a failed delivery. After a successful `EVENT` invoke, delivery to the function's code is governed by Lambda's asynchronous queue, which is at-least-once. For ordered processing on a FIFO subscriber, use `REQUEST_RESPONSE` .
`Qualifier`	A function version or alias, if not part of the ARN
`DurableExecutionName`	The name of a durable execution, for functions that use durable execution
`TenantId`	The tenant identifier passed to a multi-tenant function

Parameter	What it sets
`InvocationTimeoutSeconds`	How long EventBridge waits for the call, 1 to 30 seconds, default 30

Delivery role

The role needs `lambda:InvokeFunction` on the function, including its alias or version if you name one.

Batching and payload

One `Invoke` carries the whole batch as a JSON array. Set `BatchConfiguration.MaxBatchSize` to 1 if the function expects one event per invocation; the function then receives that event as a single JSON object rather than a one-element array. See the section called “Batching”.

Amazon SNS topic target

To deliver to an Amazon SNS topic, set `TargetArn` to the topic ARN and add `SnsParameters`. For a FIFO topic, set `MessageGroupId`, usually `{% $events.SystemMetadata.EventGroupId %}`.

Parameters

Parameter	What it sets
`MessageGroupId`	The topic message group. Required for a FIFO topic
`MessageDeduplicationId`	The deduplication identifier for a FIFO topic without content-based deduplication. Derive it from the event
`Subject`	The message subject, for email subscriptions
`MessageStructure`	Set to `json` to send a different message per protocol
`MessageAttributes`	Message attributes to attach

Delivery role

The role needs `sns:Publish` on the topic. If the topic is encrypted with a customer managed key, the role also needs `kms:GenerateDataKey` and `kms:Decrypt` on that key.

Batching and payload

EventBridge calls `PublishBatch`, so one target call carries up to 10 events, and each topic message carries one event. See the section called “Batching”.

Kinesis stream target

To deliver to a Kinesis data stream, set `TargetArn` to the stream ARN and add `KinesisParameters` with a `PartitionKey`. To keep an event group on one shard, use `{% $events.SystemMetadata.EventGroupId %}`.

Parameters

Parameter	What it sets
`PartitionKey`	The partition key for each record. Required
`ExplicitHashKey`	A hash key that overrides the shard chosen from the partition key

Delivery role

The role needs `kinesis:PutRecords` on the stream.

Batching and payload

EventBridge calls `PutRecords`, so one target call carries up to 500 events, one record each. See the section called “Batching”.

Amazon Data Firehose stream target

To deliver to an Amazon Data Firehose stream, set `TargetArn` to the stream ARN and add an empty `FirehoseParameters` block, `"FirehoseParameters": {}`, so that EventBridge knows the target type you intend.

Parameters

This target has no parameters; the payload is the whole request.

Delivery role

The role needs `firehose:PutRecordBatch` on the stream.

Batching and payload

EventBridge calls `PutRecordBatch`, so one target call carries up to 500 events, one record each. Firehose does not add a delimiter between records; if your destination expects one record per line, end each event with a newline in the `Transformer`. See the section called “Batching”.

Step Functions state machine target

To start a state machine, set `TargetArn` to the state machine ARN and add `StepFunctionsParameters`. EventBridge starts one execution per batch, and the execution input is a JSON array of the batch's events, so a `Name` expression must not depend on a single event's fields.

Parameters

Parameter	What it sets
`InvocationType`	`EVENT` calls `StartExecution` . `REQUEST_RESPONSE` calls `StartSyncExecution` , which only an Express state machine supports, and treats a failed execution as a failed delivery
`Name`	The execution name. Must be unique per execution; omit it to let Step Functions generate one
`TraceHeader`	The AWS X-Ray trace header to propagate
`InvocationTimeoutSeconds`	How long EventBridge waits for the call, 1 to 30 seconds, default 30

Delivery role

The role needs `states:StartExecution` on the state machine, or `states:StartSyncExecution` for `REQUEST_RESPONSE`.

Batching and payload

One execution per batch. Set `BatchConfiguration.MaxBatchSize` to 1 for one execution per event. See the section called “Batching”.

HTTP targets: API Gateway and API destinations

To call an HTTP endpoint, set `TargetArn` either to an API Gateway execute-api ARN, such as `arn:aws:execute-api:us-east-1:111122223333:a1b2c3d4e5/prod/POST/orders`, or to an EventBridge API destination ARN for any other HTTPS endpoint with its connection's credentials, and add `HttpParameters`. The request body is the transformed event.

Parameters

Parameter	What it sets
`PathParameterValues`	Values for path variables in the endpoint, in order
`HeaderParameters`	HTTP headers to add
`QueryStringParameters`	Query string parameters to add
`InvocationTimeoutSeconds`	How long EventBridge waits for the response, 1 to 30 seconds, default 30

Delivery role

For an API Gateway endpoint, the role needs `execute-api:Invoke` on the method's ARN. For an API destination, the role needs `events:InvokeApiDestination` on the API destination.

Batching and payload

One HTTP request carries the batch as a JSON array unless `MaxBatchSize` is 1. A response outside the 2xx range is a failed delivery. See the section called “Batching”.

Event bus target: bus to bus

To forward events to another event bus, set `TargetArn` to that bus's ARN. The destination can be another Custom Event Bus or a Custom Event Bus - Classic, in your account or another account, in the same Region or a different one. No parameter block is needed. A Custom Event Bus - Classic rule can likewise name a Custom Event Bus as its target, so events can move in either direction during a migration.

Parameters

This target has no parameters; the payload is the whole request.

Delivery role

The role needs `events:PutEvents` on the destination bus for events that carry a `PutEvents` envelope, and `events:PutRawEvents` for events published with `PutRawEvents`. EventBridge chooses the action per event and you cannot override it, so grant both when the source bus carries both kinds. A destination bus in another account must also allow your role in its resource policy or through an AWS RAM share; see the section called “Sharing a bus”.

Batching and payload

Events arrive on the destination bus as new events: they get a new event identifier and ingestion time, and the destination bus's subscribers filter them as usual. Publishing to the destination counts against that bus's throughput quota.

Loop detection

A chain of buses, rules, subscribers, and functions can send an event back to a bus it already passed through, and multiply it on each pass. EventBridge detects such loops through a Custom Event Bus on a best-effort basis: if an event, or an event derived from it, already passed through the target Custom Event Bus, EventBridge does not deliver it again, and if the subscriber has a dead-letter queue, the record carries the error code `LOOP_DETECTED`. Do not design an application that depends on an event passing through the same bus more than once, and do not rely on EventBridge to stop a loop for you. Rules on a Custom Event Bus - Classic keep their existing behavior; see the section called “Preventing infinite loops”.

Universal targets for a Custom Event Bus

A *universal target* invokes an AWS API action directly from a subscriber, with no function in between. Instead of naming a queue, topic, or function as the target, you name the service and

the API action, and you write the API request that EventBridge sends, either as fixed JSON or as a JSONata expression that builds the request from the event. A subscriber can write an item to a DynamoDB table, put an object in an Amazon S3 bucket, publish to an Amazon SNS topic, or start a Step Functions execution without a Lambda function in the path.

The target ARN

A universal target's `TargetArn` has a fixed form with no Region and no account:

```
arn:aws:events:::aws-sdk:service:apiAction
```

- *`service`* is the AWS SDK service identifier, which is not always the service's familiar name. For example, Step Functions is `sfn`, not `states`. Examples that work: `sqs`, `sns`, `s3`, `dynamodb`, `bedrockagentcore`.
- *`apiAction`* is the API action name in camelCase with a lowercase first letter: `putItem`, not `PutItem`. EventBridge looks the name up exactly as you wrote it in its index of actions, which is in camelCase, so `PutItem` is rejected by `CreateSubscriber` with the message *The api PutItem is not valid for the service dynamodb*. The same message means the action does not exist for that service.

For example, `arn:aws:events:::aws-sdk:dynamodb:putItem` invokes DynamoDB `PutItem`, and `arn:aws:events:::aws-sdk:sns:publish` invokes Amazon SNS `Publish`.

Actions that are not supported

Two kinds of action cannot be the target of a subscriber, and `CreateSubscriber` rejects them with a message that says the API, or the whole service, is not supported.

- Actions that read rather than change state, such as those whose names begin with `get`, `list`, or `describe`. An event carries something that happened, so a target action must change state. EventBridge keeps a list of blocked action names and services and can add to it, for example when an action does not behave as a target should, so an action that was accepted once is not guaranteed forever; treat the create-time message as the authority.
- The EventBridge publish actions, `eventbridge:putEvents` and `eventbridge:putPartnerEvents`. To deliver events to another event bus, set the bus ARN as the `TargetArn`; a Custom Event Bus and a Custom Event Bus - Classic are both supported as targets in their own right. See the section called “Event bus”.

Parameters

A universal target requires `InvokeConfiguration.UniversalTargetParameters`. `CreateSubscriber` rejects a subscriber that omits it, or that supplies a different parameter block such as `SqsParameters`. The message says `UniversalTargetParameters` is expected. Shape the request through `Input`; a universal-target subscriber does not use the subscriber's `Transformer`, so put any JSONata transformation in `Input`.

`Input` (required)

The API request, as a JSON string or as a JSONata expression wrapped in `{% %}` that produces the request. Use the member names from the target API's own reference; for Amazon SQS `SendMessage` that is `QueueUrl` and `MessageBody`. Up to 262,144 characters.

A fixed JSON `Input` is checked when you create the subscriber against the fields the API requires, and a missing field is rejected by name. A JSONata `Input` is checked for syntax only, because its result depends on the event, so a request that the target API rejects surfaces at delivery time instead.

`InvocationTimeoutSeconds`

How long EventBridge waits for one API call, from 1 to 30 seconds. The default is 30. This must be a literal integer; a JSONata expression is not accepted here.

The delivery role in `RoleArn` needs the IAM action for the API you invoke, such as `dynamodb:PutItem`, and `sqs:SendMessage` on the dead-letter queue. For the role's trust policy and the rest of its permissions, see the section called “Access control”.

Batching and the `$events` array

EventBridge makes one API call per batch, not one per event. `BatchConfiguration` sets the batch: `MaxBatchSize` from 1 to 500, and `MaxBatchWindowInSeconds` from 0 to 300. A batch closes when it reaches the size or when the window elapses, whichever comes first.

In a JSONata `Input`, `$events` is the batch: an array of the events in it. With a batch size of 1, `$events.Data.orderId` yields the one event's value. With a larger batch, address the array explicitly, for example with `$map` and `$join`, or the expression produces an array where the API expects a string and the delivery fails.

Set `MaxBatchSize` to 1 and `MaxBatchWindowInSeconds` to 0 to make one API call for each event. Do this whenever `Input` is fixed JSON, because a fixed request is identical for every event in the batch and the batch produces a single call.

Example: one DynamoDB item per event

The following subscriber writes one item to a DynamoDB table for each event published with `PutRawEvents`, taking the partition key from the event's `orderId` field. The delivery role needs `dynamodb:PutItem` on the table.

```
aws eventsv2 create-subscriber \
    --name orders-to-dynamodb \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --invoke-configuration '{
        "TargetArn": "arn:aws:events:::aws-sdk:dynamodb:putItem",
        "RoleArn": "arn:aws:iam::111122223333:role/EventBusDeliveryRole",
        "UniversalTargetParameters": {
            "Input": "{% { \"TableName\": \"orders\", \"Item\": { \"pk\": { \"S\":
 $events.Data.orderId }, \"total\": { \"N\": $string($events.Data.total) } } } %}"
        }
    }' \
    --batch-configuration MaxBatchSize=1,MaxBatchWindowInSeconds=0 \
    --on-failure-configuration '{ "Arn": "arn:aws:sqs:us-east-1:111122223333:orders-to-
dynamodb-dlq" }'
```

For an event published with `PutEvents`, the same fields are under `detail`: `$events.Data.detail.orderId`. See the section called “Event structure”.

When the API call fails

When you create the subscriber, EventBridge checks the ARN's shape, that the action exists and is not blocked, and that a fixed `Input` carries the action's required fields and, for a JSONata `Input`, that the expression parses. It does not check that the resource named in the request exists or that the delivery role can call the action. Those failures happen at delivery, and every delivery failure is retried under the subscriber's `RetryPolicy`, by default up to 5 attempts within 300 seconds. When a limit is reached, EventBridge writes a record to the dead-letter queue whose `errorCode` names the last failure. The codes a universal target produces most often:

- `CUSTOMER_VALIDATION`: the target API rejected the request, for example a missing required field or a value out of range. The record carries the target API's own error message.

- ACCESS_DENIED: EventBridge could not assume the delivery role, or the role is not allowed to call the action on the resource. The record says which, in an EventBridge message that names the role or the resource.
- RESOURCE_NOT_FOUND: the table, topic, or other resource named in the request does not exist.
- INPUT_TRANSFORMATION_FAILURE: the JSONata expression in Input threw an error or produced no value.
- THROTTLING: the target API throttled the call. REQUEST_TOO_LARGE: the request exceeded the action's size limit; lower MaxBatchSize. EXECUTION_TIMEOUT: the call did not return within the timeout.

For the full list of error codes, the message each carries, and how to redrive, see the section called "Retries and dead-letter queues". Turn on subscriber logs to see each attempt and the exact request sent; see the section called "Observability".

Filtering events for a subscriber

A filter selects which events a subscriber receives. Attach a FilterConfiguration to the subscriber with one or more filters, each naming a Scope and a Pattern in EventBridge event pattern syntax. For example, a DATA filter with the pattern {"detail":{"amount":[{"numeric":[">",500]}]}} delivers only orders over 500 published with PutEvents. This page describes the filter model; for the pattern operators, see *Event patterns*. Subscriber filters do not support wildcard matching.

How filtering works

Each filter has two parts: a Scope, which is the part of the event the filter examines (DATA, METADATA, or SYSTEM_METADATA; see the section called "Event structure"), and a Pattern, an event pattern supplied as a JSON string. EventBridge combines the filters in a configuration with AND: an event must match every filter before the subscriber receives it. A subscriber with no FilterConfiguration receives all events.

The three scopes

DATA

Examines the event data. For PutRawEvents, the data is the payload you sent. For PutEvents, the data is the EventBridge envelope, and your Detail is nested under the detail key.

DATA filters support every event pattern operator except wildcards. EventBridge deserializes application/avro and application/protobuf events to JSON, and a DATA filter examines that JSON like any JSON event. An application/octet-stream event has no data a filter can read, so a subscriber with a DATA filter never matches one; METADATA and SYSTEM_METADATA filters still work on it.

METADATA

Examines the key-value pairs the publisher set in Metadata with PutRawEvents. METADATA filters match exact values only: each value in the pattern must be a JSON array of exact values, and operators such as prefix, numeric, exists, and $or are not supported. To use those operators, filter on DATA or SYSTEM_METADATA. A PutEvents event has no publisher metadata, so a METADATA filter never matches one.

SYSTEM_METADATA

Examines the fields EventBridge sets on the event, listed in the following table. SYSTEM_METADATA filters support every event pattern operator except wildcards. Fields that EventBridge generates carry the aws: prefix.

Field	Description	Present on
ContentType	The content type of the event data	Every event
DeduplicationId	The deduplication identifier	Every event
EventGroupId	The event group	Every event
aws:EventId	The identifier EventBridge assigned	Every event
aws:IngestionTime	When EventBridge received the event	Every event
aws:SequenceNumber	The sequence number EventBridge assigned	Every event
aws:DeliveryType	LIVE or REPLAY	Every delivered event

Field	Description	Present on
`aws:SchemaId`	The schema identifier from the registry: a version UUID for Glue, an integer for Confluent	Avro and Protobuf events
`aws:RegistryType`	`Glue` or `Confluent`	Avro and Protobuf events
`aws:Source`	The source of the event	`PutEvents` , AWS service, and SaaS partner events
`aws:DetailType`	The detail type of the event	`PutEvents` , AWS service, and SaaS partner events

Note

To filter on AWS service events or SaaS partner events, use `aws:Source` and `aws:DetailType`. EventBridge writes these fields and no publish call can set a source that begins with `aws.`, so they identify the verified origin of the event. Do not use the `source` and `detail-type` fields in the event data for this purpose; any publisher can write those. See the section called “Event sources”.

Example: filter an event published with PutRawEvents

`PutRawEvents` publishes a payload with a content type. A `DATA` filter matches the payload directly, and a `METADATA` filter matches the publisher's metadata map. The following configuration has one filter per scope; the subscriber receives an event only when all three match: the payload's `amount` is over 500, the metadata key `region` is EU, and the content type is JSON.

```
{
    "Filters": [
        { "Scope": "DATA", "Pattern": "{\"amount\":[{\"numeric\":[\">\",500]}]}" },
        { "Scope": "METADATA", "Pattern": "{\"region\":[\"EU\"]}" },
        { "Scope": "SYSTEM_METADATA", "Pattern": "{\"ContentType\":[\"application/json
\"]}" }
    ]
```

```
}
```

Example: filter an event published with PutEvents

`PutEvents` publishes an event in the EventBridge envelope, built from the entry's `Source`, `DetailType`, `Resources`, `Time`, and `Detail`. A `DATA` filter matches this envelope, with your `Detail` nested under `detail`. The event's `ContentType` is `application/eventbridge+json`, `aws:Source` and `aws:DetailType` come from the entry's `Source` and `DetailType`, and the envelope also exposes `source` and `detail-type` in the `DATA` scope. The following configuration matches when the `amount` in the detail is over 500 and the source is `com.myapp.storage`.

```
{
    "Filters": [
        { "Scope": "DATA", "Pattern": "{\"detail\":{\"amount\":[{\"numeric\":[\">
\",500]}]}}" },
        { "Scope": "SYSTEM_METADATA", "Pattern": "{\"aws:Source\":[\"com.myapp.storage
\"]}" }
    ]
}
```

Example: filter events from an event source

Events that an event source ingests arrive in the EventBridge envelope, with the originating service or partner in `aws:Source`. To receive only Amazon S3 object-created events for one bucket, match the service in `SYSTEM_METADATA` and the bucket in `DATA`. For a partner, match the partner event source name, for example with a `prefix` on `aws.partner/example.com`.

```
{
    "Filters": [
        { "Scope": "SYSTEM_METADATA", "Pattern": "{\"aws:Source\":[\"aws.s3\"]}" },
        { "Scope": "DATA", "Pattern": "{\"detail-type\":[\"Object Created\"],\"detail
\":{\"bucket\":{\"name\":[\"amzn-s3-demo-bucket\"]}}}" }
    ]
}
```

Behavior and limits

- **AND combination**: an event must match every filter in the configuration.
- **No filter**: a subscriber with no filter configuration receives all events.

- **One filter per scope**: a configuration holds at most one filter for each scope, so at most three filters. EventBridge rejects a configuration with two filters of the same scope.
- **Wildcards**: the `wildcard` operator and `anything-but` with a wildcard are rejected, with the message *Filter pattern must not contain wildcard matchers (wildcard, anything-but wildcard)*.
- **Pattern combinations with `$or`**: EventBridge rejects a filter whose `$or` usage produces more than 1,000 pattern combinations. Multiply the number of arguments in each `$or` array: one array of three gives three combinations, a second array of two gives six.
- **Size**: all filters in a configuration share one limit of 4,096 bytes across the three scopes.
- **Pattern shape**: a `DATA` pattern can have an array at its root; `METADATA` and `SYSTEM_METADATA` patterns must be JSON objects. A pattern that is not valid JSON is rejected when you create or update the subscriber.
- **Changes**: a subscriber's filters can change 24 times in any rolling 24 hours; see the section called "Updating a subscriber".

A pattern written for the wrong shape does not fail; it matches nothing. Before you rely on a subscriber, publish one event that should match and one that should not, and check both at the target.

Transforming events with JSONata

To change what a target receives, set `Transformer` on `CreateSubscriber` or `UpdateSubscriber`. `Transformer.Type` `RAW`, the default, delivers the published data alone; `WITH_METADATA` delivers the data with its metadata; and `JSONATA` delivers the result of an expression, for example `{% { "orderId": $events.Data.orderId, "priority": $events.Data.priority } %}`. A subscriber whose target is a Custom Event Bus - Classic supports only `RAW`. A universal target does not use `Transformer`; it shapes its request in `UniversalTargetParameters.Input`, which takes the same expressions.

Transformer type	Use when	Target payload
`RAW`	The target needs the published event data	The published data, unchanged. The default.
`WITH_METADATA`	The target needs the data and its metadata	A JSON envelope with `Data`, `Metadata`, and `SystemMetadata`

Transformer type	Use when	Target payload
JSONATA	The target needs selected values or a new shape	The result of one JSONata expression per event

What $events holds

WITH_METADATA, JSONATA, and every target parameter expression read the same input object, which a JSONata expression sees as $events. It has three parts.

```
{
    "Data": { "orderId": "12345", "priority": "high" },
    "Metadata": { "tenant": "acme" },
    "SystemMetadata": {
        "ContentType": "application/json",
        "aws:EventId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "aws:IngestionTime": "2026-09-23T18:00:00Z",
        "aws:DeliveryType": "LIVE"
    }
}
```

- Data is the published data and keeps its JSON type: an object, array, string, number, Boolean, or null. For PutRawEvents it is the entry's Data value, so an order identifier is at $events.Data.orderId. For PutEvents it is the whole EventBridge envelope, with version, id, detail-type, source, account, time, region, resources, and detail, so the same identifier is at $events.Data.detail.orderId. For application/octet-stream it is a Base64 string; see the section called “Transforming binary data”.
- Metadata is the string-to-string map the producer set with PutRawEvents. PutEvents takes no metadata, so for its events the map is empty.
- SystemMetadata holds the fields EventBridge assigns. Every value is a string. EventBridge omits an optional field that has no value, and an omitted field evaluates as undefined in JSONata; only aws:DeliveryType is always present. ContentType keeps the published value, so Avro and Protobuf data that EventBridge decoded to JSON still shows application/avro or application/protobuf here, and a PutEvents event shows application/eventbridge+json. For every field, see the section called “SystemMetadata fields”.

Quote a field name that contains a colon, as in $events.SystemMetadata."aws:EventId".

Building a target payload with JSONata

Set `Transformer.Type` to `JSONATA` and put one complete expression, enclosed in `{%` and `%}`, in `JsonataConfiguration.Expression`. For a `PutRawEvents` entry whose data is `{"orderId": "12345", "priority": "high", "items": [...]}`, the following transformer delivers only the two named fields.

```
{
    "Transformer": {
        "Type": "JSONATA",
        "JsonataConfiguration": {
            "Expression": "{% {\"orderId\": $events.Data.orderId, \"priority\":
 $events.Data.priority} %}"
        }
    }
}
```

The target receives `{"orderId": "12345", "priority": "high"}`. EventBridge evaluates the expression once for each event, before batching. The result's type decides what the target receives.

Expression result	Target payload
String	The string, without JSON quotation marks
Object or array	The serialized JSON object or array. A `null` nested inside it stays JSON `null`.
Number or Boolean	The serialized JSON value
`null` or `undefined`	A transformation failure; see the section called "Limits and failures"

Transforming binary data

When `SystemMetadata.ContentType` is `application/octet-stream`, EventBridge stores the published bytes without parsing them and exposes them to an expression as a Base64 string in `$events.Data`. The bytes of `hello` arrive as `$events.Data = "aGVsbG8="`. EventBridge does not parse that string as JSON even when the decoded bytes are JSON text, so check `ContentType`

before you interpret `Data`, and use `$base64decode($events.Data)` only when the original bytes are text.

```
{% {
    "contentType": $events.SystemMetadata.ContentType,
    "base64Data": $events.Data
} %}
```

The transformer type decides how the bytes reach the target. Only `RAW` can forward the original byte array.

Transformer type	What the target receives
`RAW`	The original bytes, for Kinesis, Firehose, API Gateway, and Custom Event Bus targets. Base64 text, for Amazon SQS and Amazon SNS targets. A JSON string containing the Base64 text, for Lambda, Step Functions, Custom Event Bus - Classic, and API destinati on targets; a batched Lambda or Step Functions invocation receives an array of those strings.
`WITH_METADATA`	The envelope as UTF-8 JSON, with `Data` holding the Base64 text
`JSONATA`	The expression result as UTF-8 text or JSON; the expression reads the Base64 text

A transformer cannot set `ContentType`. When the target is another Custom Event Bus, EventBridge copies the published value, except that Avro and Protobuf data is delivered as JSON and so arrives as `application/json`.

Using JSONata in target parameters

A target parameter in `InvokeConfiguration`, such as `SqsParameters.MessageGroupId`, holds either a literal value or one complete JSONata expression enclosed in `{%` and `%}`. EventBridge does not evaluate an expression embedded inside a longer string. Target parameter expressions

read the original input object, not the payload that `Transformer` produced. The following Amazon SQS parameters take the FIFO values from each event.

```
{
    "InvokeConfiguration": {
        "TargetArn": "arn:aws:sqs:us-east-1:111122223333:orders.fifo",
        "RoleArn": "arn:aws:iam::111122223333:role/SubscriberTargetRole",
        "SqsParameters": {
            "MessageGroupId": "{% $events.Data.customerId %}",
            "MessageDeduplicationId": "{% $events.SystemMetadata.\"aws:EventId\" %}"
        }
    }
}
```

What `$events` holds depends on the parameter. Parameters that EventBridge resolves per event see one input object. Parameters that it resolves once per target invocation see an array of input objects, one per event in the batch, so the first event is `$events[0].Data`.

Parameters	$events holds
`SqsParameters` , `SnsParameters` , `KinesisParameters` , `HttpParameters` , and `EventBusV2Parameters.Metadata`	One input object, per event
`LambdaParameters` , `StepFunctionsParameters` , and `EventBusV2Parameters.DeduplicationConfiguration`	An array of input objects, per invocation
`UniversalTargetParameters.Input`	An array of input objects, per invocation

A target parameter expression must return a string, unless the field defines another type; `UniversalTargetParameters.Input` can return any JSON value that forms the API request. In a map-valued parameter, both the keys and the string values can be expressions: `SqsParameters.MessageAttributes.`*`name`*`.StringValue`, `SqsParameters.MessageSystemAttributes.`*`name`*`.StringValue`, `SnsParameters.MessageAttributes.`*`name`*`.StringValue`, `HttpParameters.QueryStringParameters.`*`key`*, and

`EventBusV2Parameters.Metadata.key`. The `BinaryValue` of an Amazon SQS or Amazon SNS message attribute is a literal Base64 value that EventBridge never evaluates, so `"BinaryValue": "AQID"` reaches the target as the bytes `0x01 0x02 0x03`; use `StringValue` for an expression.

For an API Gateway target, each `HttpParameters.HeaderParameters` key must be a literal; an API destination target also permits expressions in header keys. EventBridge rejects a header that would override request signing: `Authorization`, `Host`, and any header that begins with `X-Amz`. Every `$now()` and `$millis()` call within one resolution of a subscriber's parameters returns the same timestamp.

Functions beyond standard JSONata

In addition to the standard JSONata library, six functions are available.

Function	Result
`$hash(input, algorithm)`	Lowercase hex digest. `algorithm` is case-sensitive: `MD5`, `SHA-1`, `SHA-256`, `SHA-384`, or `SHA-512`
`$uuid()`	A version 4 UUID
`$parse(jsonString)`	The parsed JSON; an error if the string is not valid JSON
`$partition(array, chunkSize)`	The array split into chunks
`$range(start, end, delta)`	A numeric range; all three arguments are required, and the result is capped at 10,000,000 elements
`$random(seed)`	A number in [0, 1); the same seed gives the same number

The `$eval` function is not available.

Limits and failures

A `Transformer` expression and each target parameter expression can be up to 8,192 characters; a universal target's `Input` expression up to 262,144. EventBridge checks the syntax when you create or update the subscriber and rejects an invalid expression with `InvalidInputException`. At delivery, an expression can run for one second, use a stack depth of 100, and allocate 10 MiB.

An expression that throws, exceeds those limits, or produces `null` or `undefined` fails the delivery of that event. EventBridge retries under the subscriber's `RetryPolicy`, then sends the event to the on-failure destination if one is configured. The `EventTransformationFailures` metric counts each failure, an `EVENT_TRANSFORMATION_FAILURE` log record carries the error, and the dead-letter record's `errorCode` is `INPUT_TRANSFORMATION_FAILURE`. See the section called “Observability” and the section called “Retries and dead-letter queues”.

Batching deliveries to a target

EventBridge groups events into one target call when it can, so a queue receives up to 10 events per `SendMessageBatch` and a function receives an array per `Invoke`. To bound a batch, set `BatchConfiguration` on the subscriber: `MaxBatchSize` from 1 to 500, and `MaxBatchWindowInSeconds` from 0 to 300 to wait for more events before calling. For one event per call, set `MaxBatchSize` to 1.

How a batch is closed

A batch closes when the first of three limits is reached: the count in `MaxBatchSize`, the wait in `MaxBatchWindowInSeconds`, or the target's own payload size limit, for example 256 KB for an Amazon SQS request. Large events therefore arrive in smaller batches than `MaxBatchSize`. You can set the count and the window; the byte limit is the target's and cannot be changed.

- For a bespoke target, `BatchConfiguration` is optional. Without it, EventBridge batches up to the largest size the target accepts, with a window of 0 seconds.
- For a universal target, `BatchConfiguration` is required, with both fields set, because EventBridge cannot know how many events the action's request can carry.

Batching and ordering

Within a batch, events keep their publish order. A batch from a FIFO subscriber can hold events from several event groups; each group's events are in order relative to one another. A target that

processes a batch in parallel, such as a function iterating over the array with concurrency, loses that order.

Seeing the batch size

The `TargetInvocations` metric counts target calls and `EventDeliveryAttempts` counts events, so their ratio is the average batch size. See the section called “Observability”.

Retry policies and dead-letter queues

When a target call fails, EventBridge retries it until one of the two limits in `RetryPolicy` is reached, by default 5 attempts or 300 seconds, then writes a record to the Amazon SQS queue in `DeadLetterConfig.Arn`. Set both when you create the subscriber: without a dead-letter queue, an event that exhausts its retries is dropped, and the only trace is the `EventsDropped` metric.

Retry limits

Delivery stops at whichever limit is reached first.

Limit	Range	Default
`MaxRetryAttempts`	0 to 185	5
`MaxEventAgeInSeconds`	60 to 86,400	300

The defaults are shorter than a Custom Event Bus - Classic target's 24 hours and 185 attempts. A migrated design that relied on a day of retries must set `MaxEventAgeInSeconds` to 86,400 explicitly; otherwise events that fail during a target outage longer than 5 minutes go to the dead-letter queue instead of being delivered late. For a FIFO subscriber, a failing event blocks the later events in its group for as long as it is retried, so a long retry window trades order for latency.

The dead-letter record

The record is a JSON document, not the event. The following example shows one record for two events that failed together in one batch.

```
{
```

```
    "version": "1.0",
    "busArn": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
    "subscriberArn": "arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLEabcdef1234567890",
    "targetArn": "arn:aws:sqs:us-east-1:111122223333:large-orders",
    "errorCode": "ACCESS_DENIED",
    "errorMessage": "The delivery role is not authorized to perform sqs:SendMessage on
 the target.",
    "exhaustedRetryCondition": "MaximumEventAgeInSeconds",
    "retryAttempts": 4,
    "failedMessages": [
        { "eventId": "a1b2c3d4-...", "eventGroupId": "order-1001", "deduplicationId":
 null, "timestamp": "2026-09-16T18:40:12Z", "targetRequestId": "..." },
        { "eventId": "e5f6a7b8-...", "eventGroupId": "order-1001", "deduplicationId":
 null, "timestamp": "2026-09-16T18:40:13Z", "targetRequestId": "..." }
    ]
}
```

Every failure that reaches the queue was retried first: there is no delivery failure that skips `RetryPolicy`. Failures inside EventBridge itself are retried by EventBridge without limit and never produce a record. The following table lists the codes a record can carry. `errorMessage` is truncated to 1,024 characters.

errorCode	**What failed**	**errorMessage**
`CUSTOMER_VALIDATION`	The target API rejected the request (a 4xx response): a bad value, a missing field, a malformed body	The target API's own message
`ACCESS_DENIED`	EventBridge could not assume the delivery role, the role is not allowed to call the target, or the target's AWS KMS key denied access	*EventBridge could not assume `role`. Check that the role exists and that its trust policy allows the EventBridge service principal to assume it.* or *EventBridge could not access `resource`. Check that it exists and that the subscribe*

errorCode	What failed	errorMessage
		r's delivery role is allowed to access it.
RESOURCE_NOT_FOUND	The target, or its AWS KMS key, does not exist	The *could not access* message above, or the generic message
INPUT_TRANSFORMATION_FAILURE	The `Transformer` or universal-target `Input` expression threw or produced no value	The expression error
THROTTLING	The target throttled the call	The generic message
EXECUTION_TIMEOUT	The target did not respond within `InvocationTimeoutSeconds`	The generic message
REQUEST_TOO_LARGE	The request exceeded the target's size limit; lower `MaxBatchSize`	The generic message
PARTIAL_BATCH_FAILURE	The target accepted part of a batch and rejected the rest; the record's `failedMessages` lists only the rejected events	The generic message
RESOURCE_CONFLICT	The target reported a conflict, for example a name already in use	The generic message
KMS_INVALID_STATE	The target's AWS KMS key is disabled or pending deletion	The generic message

errorCode	What failed	errorMessage
SFN_TYPE_NOT_SUPPORTED	REQUEST_RESPONSE was used with a Standard state machine, which does not support synchronous execution	The generic message
COMPUTE_EXECUTION_ERROR	A REQUEST_RESPONSE function or execution ran and reported an error	The generic message
INVOCATION_DEPENDENCY_INTERNAL	The target returned a server error (a 5xx response)	The generic message
LOOP_DETECTED	The event, or an event derived from it, already passed through the target bus; see the section called "Loop detection"	The generic message
INTERNAL_SERVER_ERROR	A failure inside EventBridge while making the call that still exhausted the retries	The generic message

The generic message is *The delivery failed. Use this record's error code and request id when investigating, and contact AWS Support if the failure persists.* EventBridge uses it wherever passing the underlying message through could expose another service's internals, so for those codes the `errorCode` and `targetRequestId` are the facts to work from, together with the `EVENT_DELIVERY_ATTEMPT` log records, which carry the request that was sent. The same values arrive as Amazon SQS message attributes (`ERROR_CODE`, `ERROR_MESSAGE`, `SUBSCRIBER_ARN`, `TARGET_ARN`, `BUS_ARN`, `EXHAUSTED_RETRY_CONDITION`, `RETRY_ATTEMPTS`), so a consumer can route on them without parsing the body. For a FIFO subscriber, use a FIFO queue as the dead-letter queue; EventBridge sets each record's `MessageGroupId` from the events' `EventGroupId`, so records for one group stay in order.

The delivery role needs `sqs:SendMessage` on the queue. Without it, the `OnFailureDestinationFailed` metric counts each record that could not be written, and those events are lost.

Redriving events from the dead-letter queue

The queue holds records, not events, so you redrive by replaying the events from the bus. The events must still be within the bus's retention period.

1. Read the records and collect every `failedMessages[].eventId`, plus the earliest and latest `timestamp`.
2. Fix the cause the `errorCode` names: the role's policy, the target, or the expression.
3. Create a subscriber on the same bus with the same target and role, a `POINT_IN_TIME` starting position whose `StartingPoint` is the earliest timestamp and whose `EndPoint` is the latest, and a `SYSTEM_METADATA` filter on the identifiers.
4. When the subscriber has delivered the events, delete it and delete the records from the queue.

```
aws eventsv2 create-subscriber \
    --name redrive-2026-09-16 \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --starting-position POINT_IN_TIME \
    --point-in-time-configuration '{ "PointType": "TIMESTAMP", "StartingPoint":
 "2026-09-16T18:40:00Z", "EndPoint": "2026-09-16T18:41:00Z" }' \
    --filter-configuration '{ "Filters": [ { "Scope": "SYSTEM_METADATA", "Pattern":
 "{\"aws:EventId\":[\"a1b2c3d4-...\",\"e5f6a7b8-...\"]}" } ] }' \
    --invoke-configuration '{ "TargetArn": "arn:aws:sqs:us-east-1:111122223333:large-
orders", "RoleArn": "arn:aws:iam::111122223333:role/EventBusDeliveryRole" }'
```

The replayed events arrive with `aws:DeliveryType` `REPLAY`, so a consumer can tell them from live traffic.

Updating, pausing, and resuming a subscriber

`UpdateSubscriber` changes a subscriber in place. A top-level member you omit keeps its stored value; a configuration object you supply replaces the stored object whole, so to change one filter, pass the complete `FilterConfiguration` with every filter you want to keep. To pause

a subscriber, pass `--state STOPPED`; to resume, pass `--state RUNNING` with a `--resume-position`.

What you can change

The following request replaces the subscriber's whole `InvokeConfiguration`, so it names the role and every target parameter the subscriber must keep. `TargetArn` is not part of the update; a subscriber's target cannot change.

```
{
    "SubscriberArn": "arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLEabcdef1234567890",
    "InvokeConfiguration": {
        "RoleArn": "arn:aws:iam::111122223333:role/NewSubscriberTargetRole",
        "SqsParameters": {
            "MessageGroupId": "{% $events.Data.customerId %}"
        }
    }
}
```

The following table lists each member of the request and what supplying it does.

Request member	Effect
`Description`	Replaces the description. An empty or blank value clears it.
`State`	Moves the subscriber between `RUNNING` and `STOPPED`; see the section called “Pausing and resuming”
`ResumePosition`	Where a `STOPPED` to `RUNNING` transition resumes
`InvokeConfiguration`	Replaces the delivery role and the target parameters. `TargetArn` stays as it is.
`FilterConfiguration`	Keeps, clears, or replaces the whole filter configuration; see the section called “Updating filters”

Request member	Effect
`BatchConfiguration`	Replaces the whole batching configuration
`Transformer`	Replaces the whole transformer. Set `Type` to `RAW` to remove a metadata or JSONata transformer.
`RetryPolicy`	Replaces the whole retry policy. An omitted member takes its default: 5 retry attempts, 300 seconds, and `RetryStrategy ALL`, the only value, which retries every failure category.
`OnFailureConfiguration`	Replaces the whole on-failure configuration. Send `{}` to remove the destination; with no destination, an event is dropped when its retries are exhausted.
`LogConfiguration`	Replaces the whole logging configuration. Send `{}` to restore the defaults, `Level OFF` and `IncludePayload ON_ERROR_ONLY`, which turns delivery logging off.

`Name`, `EventBusArn`, `Type`, `StartingPosition`, `PointInTimeConfiguration`, and `TargetArn` identify the subscriber or fix its starting point and cannot be changed; tags change through `TagResource` and `UntagResource`. The update response carries the subscriber's identity, state, starting position, and `LastModifiedTime`; read the configuration back with `DescribeSubscriber`.

An update never moves the subscriber's position on the bus: it does not rewind, skip, or replay. To re-deliver past events under a new configuration, create a new subscriber with a starting position instead; see the section called “Replaying events”.

Troubleshooting: an update cleared settings you did not change

An update replaces each configuration object you supply as a whole; see the section called “What you can change”.

Symptom

After an update, a filter, log setting, retry limit, or retention period you did not touch is back at its default, and no error was returned.

Cause

An update leaves an omitted top-level member unchanged but replaces a supplied block whole. An update that passes `FilterConfiguration`, `LogConfiguration`, `RetryPolicy`, or `StorageConfiguration` to change one field clears the block's other fields. An event source updated without its `Pattern` comes back with no pattern, which widens it to every event.

Solution

Read the resource, modify the whole block, write the whole block back, then read it again. The read-back is the only confirmation, because create and update responses do not echo every field.

```
aws eventsv2 describe-subscriber --subscriber-arn "$SUBSCRIBER_ARN" --query
 'LogConfiguration'

aws eventsv2 update-subscriber --subscriber-arn "$SUBSCRIBER_ARN" \
    --log-configuration '{ "Level": "INFO", "IncludePayload": "FULL" }'

aws eventsv2 describe-subscriber --subscriber-arn "$SUBSCRIBER_ARN" --query
 'LogConfiguration'
```

Verification

The second describe shows every field of the block with the values you intended.

Updating filters

To replace the filters, pass a `FilterConfiguration` whose `Filters` list holds every filter the subscriber should have; EventBridge combines filters for different scopes with AND. To remove every filter, pass `FilterConfiguration` as `{}`.

Request		Result
`FilterConfiguration`	omitted	The stored filters stay

Request	Result
`FilterConfiguration` is `{}`	Every stored filter is removed
`Filters` is a non-empty list	The list is validated and replaces the whole filter configuration
`Filters` is an empty list, or `Language` is given without `Filters`	`InvalidInputException`; a supplied `FilterConfiguration` must contain at least one filter

Filter changes are limited to 24 per subscriber in any rolling 24 hours. Only a change to the set of scopes and patterns counts; re-submitting an identical `FilterConfiguration` does not. The 25th change fails with `FILTER_CHANGE_LIMIT_EXCEEDED`. This limit is not a Service Quota and cannot be raised, so a design that rewrites filters continuously should instead publish the varying value into the event and filter on it once.

Which configuration an already matched event uses

EventBridge matches an event to a subscriber and delivers it at different times, and it records which version of the filters matched each event. An update that changes what the filters match creates a new filter version, and EventBridge stores the transformer and target parameters with it. An event matched before the change is delivered with the old transformer and target parameters; an event matched after it uses the new ones. The same holds after a subscriber stops and resumes.

Filter update	Creates a new filter version
Changing a pattern	Yes
Changing which scopes have a filter	Yes
Adding the first filter	Yes
Clearing all filters	Yes, when filters were stored
Reordering equivalent filters	No
Changing only `Language`	No

So when a filter and a transformer or target parameter form one behavior, change them in the same request. The following request pairs a new DATA filter with the transformer written for it, so no event is delivered with a mismatched pair.

```
{
    "SubscriberArn": "arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLEabcdef1234567890",
    "FilterConfiguration": {
        "Filters": [
            { "Scope": "DATA", "Pattern": "{\"type\":[\"priority\"]}" }
        ]
    },
    "Transformer": {
        "Type": "JSONATA",
        "JsonataConfiguration": {
            "Expression": "{% {\"version\": 2, \"event\": $events.Data} %}"
        }
    }
}
```

An update that changes only the transformer or the target parameters creates no filter version. Once it propagates, the new configuration applies to events that are still waiting for delivery and to later retry attempts. The delivery role, retry policy, on-failure destination, batching, and logging configuration always take the latest propagated values, whatever version matched the event. Because EventBridge can evaluate the transformer and target parameters again for a retry, a time-dependent or nondeterministic JSONata function such as $now( ) can produce a different result on each attempt; do not assume a retried request carries identical bytes.

UpdateSubscriber stores the change before it returns, and the delivery components pick it up after a propagation delay, during which events can still use the previous configuration. DescribeSubscriber confirms what is stored, not that every component has switched. After an update, publish a test event and check the target payload before you rely on the new behavior.

Only one subscriber create or delete runs at a time on a bus, across every account that uses it; a concurrent call fails with ConcurrentModificationException. Retry it.

Pausing and resuming

A subscriber delivers only while its State is RUNNING. Set it to STOPPED to pause; events continue to accumulate on the bus, bounded by retention. When you set it back to RUNNING, ResumePosition decides what happens to the backlog.

- LAST_PROCESSED, the default: continue from where delivery stopped, so every event published while the subscriber was stopped is delivered, oldest first.
- LATEST: skip the backlog and deliver only events published from now on. A subscriber that is replaying cannot use it: an open-ended replay is still catching up, and a bounded replay never reaches live events.

```
aws eventsv2 update-subscriber \
    --subscriber-arn arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLEabcdef1234567890 \
    --state STOPPED

aws eventsv2 update-subscriber \
    --subscriber-arn arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLEabcdef1234567890 \
    --state RUNNING \
    --resume-position LAST_PROCESSED
```

`ResumePosition` is accepted only on the call that moves the state from `STOPPED` to `RUNNING`. On that call EventBridge first makes the subscriber's delivery endpoint discoverable again; if that step fails, the call returns `InternalException`, nothing is saved, and the subscriber stays `STOPPED`, so retry the request. A subscriber that the bus owner has revoked does not deliver in either state; see the section called “Revoking”.

Errors an update can return

EventBridge validates the updated configuration against the subscriber's stored target type, so an `SqsParameters` block on a subscriber whose target is a Lambda function is rejected.

Condition	Error
The subscriber does not exist	`ResourceNotFoundException`
The subscriber is revoked	`InvalidStateException`
The request has more than one target parameter block	`InvalidInputException`
The target parameters do not match the stored target type	`InvalidInputException`

Condition	Error
The transformer is not supported for the target	`InvalidInputException`
A JSONata expression is invalid	`InvalidInputException`
A batching value is outside the target's limits	`InvalidInputException`

Replaying retained events to a subscriber

To deliver events that the bus already holds, create a subscriber with `--starting-position POINT_IN_TIME` and a starting point inside the bus's retention window, for example `--point-in-time-configuration '{ "PointType": "TIMESTAMP", "StartingPoint": "2026-09-01T00:00:00Z" }'`. There is no separate replay API: a subscriber reads retained events from its starting position, so anything still inside the retention period can be delivered again. `DescribeEventBus` returns `RetentionWindowStartTime`, the earliest point a subscriber can read from; on a new bus that is the bus's creation time, so a recent timestamp can still fall outside the window. For the retention period itself, see the section called “Choosing the retention period”.

Starting position

When you create a subscriber, `--starting-position` is `LATEST` to receive only events published after the subscriber exists, or `POINT_IN_TIME` to read from retained history. With `POINT_IN_TIME`, `--point-in-time-configuration` sets `PointType` to `HORIZON`, the earliest retained event, or to `TIMESTAMP` with a `StartingPoint`. The starting point must fall inside the retention window: at or after `RetentionWindowStartTime` and not in the future. An optional `EndPoint` stops the subscriber at that time. Omit `EndPoint` and the subscriber reads through the retained events, catches up to live events, and keeps delivering.

```
aws eventsv2 create-subscriber \
    --name reprocess-september \
    --event-bus-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --starting-position POINT_IN_TIME \
    --point-in-time-configuration '{ "PointType": "TIMESTAMP", "StartingPoint":
 "2026-09-01T00:00:00Z" }' \
```

```
    --invoke-configuration '{
        "TargetArn": "arn:aws:sqs:us-east-1:111122223333:reprocess",
        "RoleArn": "arn:aws:iam::111122223333:role/EventBusDeliveryRole"
    }'
```

A replay has a startup delay and then runs at live speed. Allow for the delay before you conclude that a replay is not working. To pause a subscriber and later resume it with or without the backlog, see the section called “Pausing and resuming”.

Telling replayed events apart

Every delivered event carries `SystemMetadata."aws:DeliveryType"`, which is `LIVE` for an event delivered as it arrived and `REPLAY` for an event read from retained history. Use it to make consumers idempotent during a replay.

Ordering and deduplicating events on a Custom Event Bus

Ordering is a property of a subscriber. Deduplication happens when you publish. The two are configured in different places and solve different problems.

Ordering

A subscriber's `Type` is `UNORDERED` or `FIFO`. An unordered subscriber delivers events in parallel for the highest throughput. A FIFO subscriber delivers events in the order that they were published within an *event group*. You assign the group when you publish, with `SystemMetadata.EventGroupId` in each entry of either publish API.

An event group belongs to the account that published it. Two accounts that publish to a shared bus with the same `EventGroupId` create two separate groups: their events are not ordered relative to each other, and they do not share a group's throughput. For the group limits, see the section called “Custom Event Bus quotas”.

Within a FIFO subscriber, an event that cannot be delivered holds up the later events in its own group while other groups continue. Keep group identifiers bounded, for example a customer identifier or an order identifier, and do not derive them from unbounded input. A FIFO target such as an Amazon SQS FIFO queue enforces its own length and character limits on the group identifier.

A FIFO subscriber accepts `InvocationType` `EVENT` for a Lambda target, but order then holds only up to the hand-off into Lambda's asynchronous queue. Use `REQUEST_RESPONSE` for ordered processing.

Deduplication at publish time

You deduplicate either by setting `DeduplicationType=CONTENT_BASED` on the request or by setting `SystemMetadata.DeduplicationId` on each entry.

- Set `--deduplication-configuration DeduplicationType=CONTENT_BASED` on the request. EventBridge hashes the content of each entry and suppresses an entry whose hash it has already accepted.
- Omit `DeduplicationConfiguration` and set `SystemMetadata.DeduplicationId` on each entry to a key that you already have, such as an idempotency key from your producer.

EventBridge suppresses a duplicate that arrives within 5 minutes (300 seconds) of the entry it repeats. The check is scoped to your account, and to the event group when the entry carries an `EventGroupId`: an entry with the same key from another account, or in another group, is a different event. An entry without an `EventGroupId` is still deduplicated, across all of your account's ungrouped entries on that bus.

Together, this gives exactly-once ingestion: a duplicate publish within the window is suppressed at the bus, and a FIFO subscriber then hands each accepted event to its target once, in order, within its group.

Either way, a suppressed duplicate returns `SuccessCode: DEDUPLICATED`. Treat it as a success. Code that counts every success as a new event overstates delivery. EventBridge bills content-based deduplication as a separate operation, so prefer a `DeduplicationId` when your producer already has a key. For pricing, see Amazon EventBridge pricing. For the content types that deduplication applies to, see the section called “Supported content types”.

Migrating from Custom Event Bus - Classic to the Custom Event Bus

Custom Event Bus - Classic remains available, and you can run both products side by side. Your identity policies and delivery roles carry over, because the Custom Event Bus keeps the `events:` IAM namespace and the `events.amazonaws.com` service principal. The routing model changes.

Custom Event Bus - Classic	Custom Event Bus
A rule and one of its targets	One subscriber

Custom Event Bus - Classic	Custom Event Bus
A rule with five targets	Five subscribers on the same bus
Event pattern on a rule	A filter with a scope of `DATA`. When producers publish with `PutEvents` , the pattern is unchanged.
Input transformer	A transformer with a type of `JSONATA`
Dead-letter queue on a target	`OnFailureConfiguration` on the subscriber
Retry policy on a target	`RetryPolicy` on the subscriber
Archive and replay	Retention on the bus and a subscriber starting position
Rules written by the bus owner for each consumer	Subscribers created by the consumers themselves
`events:PutEvents`	`events:PutEvents` , plus `events:PutRawEvents` for non-JSON payloads

Four capabilities have no Custom Event Bus - Classic equivalent, so you can use them without a workaround:

- Retention on the bus itself
- A subscriber starting position
- Pause and resume with a backlog
- A second publish API for non-JSON payloads

You do not have to move everything. The two buses run side by side, so a Custom Event Bus - Classic rule can target a Custom Event Bus, and a subscriber can target a Custom Event Bus - Classic. You can move one consumer or one producer at a time. See the section called “Event bus”.

What does not change

The following stay the same, so the corresponding parts of your deployment carry over as they are.

Unchanged	What it means for the move
IAM action namespace `events:` and the `events.amazonaws.com` service principal	Identity policies, bus resource policies, and delivery role trust policies are reused
Target services: Amazon SQS, Lambda, Amazon SNS, Kinesis, Firehose, Step Functions , API Gateway, API destinations, event buses	Delivery roles keep the same target actions; each target has a parameter block with the same fields as its Classic counterpart. See the section called “Targets”
Event pattern syntax, except wildcards	Rule patterns become DATA filters unchanged for `PutEvents` traffic. See the section called “Filtering”
AWS service events and SaaS partner events	The same events reach a Custom Event Bus through an event source. See the section called “Event sources”
The `PutEvents` API and envelope	Producers change the endpoint and client, not the request
Input transformation with JSONata	Expressions move to `Transformer` with the event under `$events`

Defaults that differ

Important

A subscriber retries a failed delivery for 300 seconds and 5 attempts by default. A Custom Event Bus - Classic target retries for 24 hours and 185 attempts. If your consumers rely on a day of retries during an outage, set `RetryPolicy.MaxEventAgeInSeconds` to 86,400 and `MaxRetryAttempts` to 185 on each subscriber, and attach a dead-letter queue.

Otherwise, events that fail for more than 5 minutes are dead-lettered rather than delivered late. See the section called “Retries and dead-letter queues”.

Two more defaults differ. A subscriber delivers a batch to a function or a state machine as a JSON array. In contrast, a rule delivered one event per invocation. To keep one event per call, set `BatchConfiguration.MaxBatchSize` to 1.

A bus also retains events, so a subscriber created after an incident can read what it missed. A rule created late, by contrast, saw nothing.

Classic targets without a bespoke equivalent

For every Custom Event Bus - Classic target type that has no bespoke target on the Custom Event Bus, use a universal target that calls the service's API action. Set `TargetArn` to `arn:aws:events:::aws-sdk:`*`service`*`:`*`apiAction`* and build the request in `Input`. See the section called “Universal targets”.

Custom Event Bus - Classic target	Universal target action
Amazon ECS task	`arn:aws:events:::aws-sdk:ecs:runTask`
AWS Batch job	`arn:aws:events:::aws-sdk:batch:submitJob`
CodeBuild project	`arn:aws:events:::aws-sdk:codebuild:startBuild`
CodePipeline pipeline	`arn:aws:events:::aws-sdk:codepipeline:startPipelineExecution`
Systems Manager Run Command	`arn:aws:events:::aws-sdk:ssm:sendCommand`
Systems Manager Automation	`arn:aws:events:::aws-sdk:ssm:startAutomationExecution`

Custom Event Bus - Classic target	Universal target action
AWS Glue workflow	`arn:aws:events:::aws-sdk:glue:startWorkflowRun`
SageMaker pipeline	`arn:aws:events:::aws-sdk:sagemaker:startPipelineExecution`
Redshift Data API statement	`arn:aws:events:::aws-sdk:redshiftdata:executeStatement`
CloudWatch Logs log group	`arn:aws:events:::aws-sdk:cloudwatchlogs:putLogEvents`
Amazon EC2 actions (stop, reboot, terminate, create snapshot)	`arn:aws:events:::aws-sdk:ec2:stopInstances` , `rebootInstances` , `terminateInstances` , `createSnapshot`
Inspector assessment	`arn:aws:events:::aws-sdk:inspector:startAssessmentRun`

The service and action names follow the universal-target naming rules. `CreateSubscriber` rejects a name it does not recognize, so confirm each one when you create the subscriber.

Migration sequence

Move one bus at a time, in five steps, keeping the Custom Event Bus - Classic rules running until the last step. Each step is reversible until you delete the rules.

> **Important**
>
> A pattern copied from a rule matches events published with `PutEvents`, because that API produces the same envelope as Custom Event Bus - Classic. The same pattern matches nothing on events published with `PutRawEvents`, because the payload is not nested under `detail`. See the section called “Event structure”.

1. Create the Custom Event Bus with the retention period that you want, and wait for its state to become `ACTIVE`.
2. For each rule, create one subscriber for each of its targets. Use the rule's event pattern as a `DATA` filter, its input transformer as a `JSONATA` transformer, and its target role as the `RoleArn`. Attach a dead-letter queue and turn on logs before you send traffic.
3. Publish to both buses from your producers, and confirm at each target that both paths deliver the same events.
4. Move producers to the Custom Event Bus only.
5. Disable the rules on the Custom Event Bus - Classic. Delete them after the retry window and your own verification period have passed.

Access control for the Custom Event Bus

You control who can use a Custom Event Bus with IAM policies, the same way as for other AWS services. To let a role or user in your own account publish to a bus, manage subscribers, or administer buses, attach an identity-based policy to that role or user. For example, a producer's role needs `events:PutRawEvents` on the bus ARN. To let another account publish to your bus or attach subscribers to it, share the bus with that account through AWS Resource Access Manager (AWS RAM), or attach a resource policy to the bus; see the section called “Sharing a bus”. Every action name uses the `events:` prefix, for example `events:CreateSubscriber`; for the CLI, SDK, and endpoint names, see the section called “Names, endpoints, and permissions”.

- the section called “Actions and condition keys”
- the section called “Revoking”
- the section called “Permissions by scenario”

Actions, resources, and condition keys for the Custom Event Bus

Use this page when you write an IAM policy. It lists every `events:` action for the Custom Event Bus, the resource ARN each one is checked against, whether another account can be granted it, and the condition keys you can add. For example, you can limit a role to subscribers whose `METADATA` filter matches `tenant` to `acme` alone. To do so, grant `events:CreateSubscriber` and `events:UpdateSubscriber` on the bus with `Null` and `ForAllValues:StringEquals` conditions on `events:Metadata/tenant`.

Condition keys

EventBridge supplies the following condition keys from the request, so that you can narrow a grant beyond the action and the resource. For the global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Condition key	Value	Supplied on
`events:source`	The event's source, or the source being forwarded	`PutEvents` , `CreateEventSource` , `UpdateEventSource`
`events:detail-type`	The event's detail type	`PutEvents`
`events:eventBusInvocation`	`true` for a forwarded event, `false` for a direct publish	`PutEvents` , `PutRawEvents`
`events:SystemMetadata/ContentType`	The content type in the event's system metadata	`PutRawEvents` , and `PutEvents` for a forwarded event
`events:SystemMetadata/AwsSource`	The source of a forwarded event	`PutEvents` , `PutRawEvents` , for a forwarded event only
`events:SystemMetadata/AwsDetailType`	The detail type of a forwarded event	`PutEvents` , `PutRawEvents` , for a forwarded event only
`events:Metadata/` *`key`*	On `PutRawEvents` , the value of that metadata key on the event. On the subscriber actions, the values a `METADATA` filter matches on that key	`PutRawEvents` , `CreateSubscriber` , `UpdateSubscriber`
`events:Metadata/` *`key`*`/Matcher`	`exact`	`CreateSubscriber` , `UpdateSubscriber`

Condition key	Value	Supplied on
events:ContentFilterPresent	true when the filter configuration declares a DATA filter, otherwise false	CreateSubscriber , UpdateSubscriber
events:PolicyName	The resource policy that the request acts on, default or AWS_RAM	PutResourcePolicy , GetResourcePolicy , DeleteResourcePolicy
aws:ResourceTag/ *key*	A tag on the bus, subscriber, or event source being checked	Every action checked against a resource
aws:RequestTag/ *key*	The value of that tag in the request	The create operations and TagResource
aws:TagKeys	The tag keys in the request	The create operations, TagResource , and UntagResource

A *forwarded event* is one that a rule or subscriber delivers to the bus, under a role that you supplied. EventBridge checks it with events:eventBusInvocation set to true; see the section called “Events that arrive without a publish call”. A forwarded AWS service or partner event is checked as PutEvents, unless a subscriber transformed it. A subscriber transforms an event when its Transformer.Type is WITH_METADATA or JSONATA. A forwarded event checked as PutEvents carries events:source and events:detail-type with the event's source and detail type. Any other forwarded event is checked as PutRawEvents. Either way, events:SystemMetadata/AwsSource and events:SystemMetadata/AwsDetailType hold the event's source and detail type. Both keys are absent when the event has no source or detail type, or when a subscriber transformed it. So a Deny statement that tests the value of one of these keys does not apply to an event that lacks the key.

On PutRawEvents, events:Metadata/*key* holds a single value: the event's value for that metadata key. On CreateSubscriber and UpdateSubscriber, it holds every value that the METADATA filter matches on that key. A filter can match several values, so match the key with a set operator. ForAllValues:StringEquals matches only when every value is one that you list. ForAnyValue:StringEquals matches when any one value is one that you list, so it also allows a

filter that adds other values. `ForAllValues` also matches when the filter does not name the key, so add a `Null` condition to require the key. `METADATA` filters are exact-match only, which is why the matcher value is always `exact`. A `DATA` filter matches arbitrary JSON, so only its presence is exposed, not its content. A `SYSTEM_METADATA` filter supplies no condition key.

`UpdateSubscriber` supplies the three filter keys only when the request includes a `FilterConfiguration`, even one that leaves the filter unchanged. That request is also checked as `events:UpdateSubscriber` against the bus. A condition in a statement that grants only `events:CreateSubscriber` does not apply to that check. So put the same conditions on `events:UpdateSubscriber`. A `FilterConfiguration` of `{}` removes every filter. It supplies `events:ContentFilterPresent` as `false` and no metadata keys, so a `Null` condition on a metadata key denies it.

When an `UpdateEventSource` request includes a `Configuration`, EventBridge also checks `events:UpdateEventSource` against the bus. It sets `events:source` from that configuration. So put the conditions that you set on `events:CreateEventSource` on `events:UpdateEventSource` as well. A condition on `events:source` then governs a repointed event source as well as a new one.

Use `events:PolicyName` to limit a principal to one of the bus's two resource policies; see the section called “Write a custom resource policy”. The key holds the name of the policy that the request acts on. When you omit `PolicyName` in your own request, the request acts on `default`, and the key is `default`. AWS Resource Access Manager (AWS RAM) writes the `AWS_RAM` policy with the credentials of the principal that changes a resource share. EventBridge checks that write as `events:PutResourcePolicy`, with the key set to `AWS_RAM`. So a principal that you limit to `default` cannot share the bus through AWS RAM. `ListResourcePolicies` reports on both policies, so it does not supply this key. So a `StringEquals` condition on this key never matches `ListResourcePolicies`. Allow that action in a statement without that condition.

The following resource policy lets account `444455556666` create subscribers on the `orders` bus and change their filters. It allows both only when the subscriber's `METADATA` filter matches `tenant` exactly to `acme`. A subscriber with no `METADATA` filter, or with a filter that names any tenant other than `acme`, is denied. EventBridge supplies a metadata condition key only for the fields a filter declares. So the `Null` condition is what denies a filter that omits `tenant`. The conditions restrict only this statement's grant. If another statement, or an AWS RAM share, also allows `events:CreateSubscriber` for that account, the conditions do not restrict that grant.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SubscribeToAcmeEventsOnly",
            "Effect": "Allow",
            "Principal": { "AWS": "arn:aws:iam::444455556666:root" },
            "Action": [
                "events:CreateSubscriber",
                "events:UpdateSubscriber"
            ],
            "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
            "Condition": {
                "Null": { "events:Metadata/tenant": "false" },
                "ForAllValues:StringEquals": { "events:Metadata/tenant": "acme" },
                "StringEquals": { "events:Metadata/tenant/Matcher": "exact" }
            }
        }
    ]
}
```

Actions and the resources they are checked against

The following table lists every action, the resource ARN to put in the policy's `Resource` element, and whether a bus owner can grant the action to another account through a resource policy or an AWS RAM share. The three `List` actions are not checked against any resource, so grant them with `"Resource": "*"`.

IAM action	Checked against	Grantable to another account
`events:CreateEventBus`	The bus being created	No. Created in the caller's own account
`events:DescribeEventBus`	The bus	Yes
`events:UpdateEventBus`	The bus	No. Bus owner only

IAM action	Checked against	Grantable to another account
events:DeleteEventBus	The bus	No. Bus owner only
events:ListEventBuses	Nothing. Account-wide	No. Lists the caller's own and shared buses
events:PutEvents	The bus, once per entry in the request	Yes
events:PutRawEvents	The bus, once per entry in the request	Yes
events:CreateSubscriber	The bus, and the subscriber being created	Yes
events:DescribeSubscriber	The subscriber	The subscriber's owner authorizes it
events:UpdateSubscriber	The subscriber, and the bus when the request includes a FilterConfiguration	Yes, for the bus check
events:DeleteSubscriber	The subscriber	The subscriber's owner authorizes it
events:ListSubscribers	Nothing. Account-wide	No. Lists the caller's own subscribers
events:RevokeResource	The bus	No. Bus owner only
events:CreateEventSource	The bus, and the event source being created	Yes
events:DescribeEventSource	The event source	The event source's owner authorizes it

IAM action	Checked against	Grantable to another account
events:UpdateEventSource	The event source, and the bus when the request includes a Configuration	Yes, for the bus check
events:DeleteEventSource	The event source	The event source's owner authorizes it
events:ListEventSources	Nothing. Account-wide	No. Lists the caller's own event sources
events:PutResourcePolicy	The bus	No. Bus owner only
events:GetResourcePolicy	The bus	No. Bus owner only
events:DeleteResourcePolicy	The bus	No. Bus owner only
events:ListResourcePolicies	The bus	No. Bus owner only
events:TagResource	The bus, subscriber, or event source named in the request	No. The resource's owner only
events:UntagResource	The bus, subscriber, or event source named in the request	No. The resource's owner only
events:ListTagsForResource	The bus, subscriber, or event source named in the request	No. The resource's owner only

To create a subscriber or an event source on a bus, the caller needs the action on the bus as well as on the new resource. That is what lets a bus owner decide, in the bus's own policy, which accounts may attach anything to it. CreateSubscriber makes four separate authorization checks, and all four must allow: events:CreateSubscriber on the bus, events:CreateSubscriber

on the subscriber, `iam:PassRole` on the role in `InvokeConfiguration.RoleArn`, and `events:TagResource` on the subscriber when the request carries tags. `CreateEventSource` runs the same checks without `iam:PassRole`, because an event source has no role. `UpdateSubscriber` also checks `events:UpdateSubscriber` on the bus when the request includes a `FilterConfiguration`. It also checks `iam:PassRole` on the role in `InvokeConfiguration.RoleArn` when the request includes one. So another account needs `events:CreateSubscriber` and `events:UpdateSubscriber` on your bus to attach subscribers and to update their filters. The AWS RAM managed permissions for attaching subscribers include both.

> **Note**
>
> A resource that does not exist yet has no complete ARN, so a create check uses an ARN ending in a wildcard, such as `arn:aws:events:us-east-1:111122223333:subscriber/`*`name`*`/*`. A policy scoped to one subscriber therefore matches by name prefix, not exactly. An untagged create supplies no `aws:ResourceTag` for the new resource, so a `StringEquals` condition on that key denies it.

Example identity-based policies

A **producer** needs `events:PutEvents`, `events:PutRawEvents`, or both, on the bus ARN. Grant the action that matches the API it calls. A producer to a bus encrypted with a customer managed key also needs access to that key; see the section called “Encryption”. A **subscriber owner** needs `events:CreateSubscriber` on the bus and on the subscriber, `iam:PassRole` on the delivery role, and the describe, update, and delete subscriber actions on the subscribers it manages. For an `UpdateSubscriber` request that includes a `FilterConfiguration`, it also needs `events:UpdateSubscriber` on the bus. A **bus administrator** needs the create, update, and delete bus actions, the four resource policy actions, and `events:RevokeResource`.

The following policy lets a producer publish to one bus with either publish API, and read the bus so that it can confirm the bus is `ACTIVE` first. Publishing is authorized once for every entry in the request, and there is no partial authorization: if one entry fails a condition check, EventBridge denies the whole request, including the entries that would have passed.

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "PublishToOrdersBus",
            "Effect": "Allow",
            "Action": [
                "events:PutEvents",
                "events:PutRawEvents",
                "events:DescribeEventBus"
            ],
            "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef"
        }
    ]
}
```

Tagging

You can tag a bus, a subscriber, or an event source. Set `Tags` on the create call, or use `TagResource`, `UntagResource`, and `ListTagsForResource` afterward. Each resource carries its own tag set, so tagging a bus does not tag the subscribers or event sources attached to it.

```
aws eventsv2 tag-resource \
    --resource-arn arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef \
    --tags team=payments,environment=production
```

A create response does not echo the tags you supplied, so read them back with `list-tags-for-resource` when you need to confirm them. Keys that begin with `aws:` are reserved for AWS and are rejected on every write, including `UntagResource`. To control who can tag, and to write policies that depend on tags, use the three tag actions in the preceding table and the `aws:RequestTag`, `aws:ResourceTag`, and `aws:TagKeys` condition keys.

Revoking a subscriber or an event source

`RevokeResource` withdraws one subscriber or one event source from a bus. It is the only control a bus owner has over a resource that another account owns, and it needs no cooperation from that account. It is authorized against the bus, and only the account that owns the bus can call it. The account that owns the subscriber cannot revoke its own subscriber.

> **Note**
>
> You revoke with the AWS CLI (`aws eventsv2 revoke-resource`), the API, or an SDK. The console does not offer revocation.

Revocation is terminal and cannot be undone. A revoked subscriber stops delivering and refuses mutating operations with `InvalidStateException`. A revoked event source stops ingesting: each event it would have forwarded is refused as it arrives. `DeleteSubscriber` and `DeleteEventSource` stay available so that the resource's owner can clean it up. `DescribeSubscriber`, `DescribeEventSource`, and both list operations return a `Revoked` field that is present only when it is `true`.

> **Important**
>
> There is no undo, so confirm the target with `DescribeSubscriber` or `DescribeEventSource` before you revoke. Check the ARN, the bus, and the target.

Revocation covers one resource, not the account that created it. That account can create another subscriber or event source and carry on. To stop the account, also deny it `events:CreateSubscriber` and `events:CreateEventSource` in the bus resource policy, or delete the AWS RAM share. Deleting the resource is not an alternative: `DeleteSubscriber` and `DeleteEventSource` are authorized against the resource named in the request, so only its owner can delete it.

Permissions by scenario

Each role in a Custom Event Bus deployment needs a small, specific set of grants. This page lists them by scenario so that you can copy the set for the role you are creating. For example, a producer's role needs `events:PutRawEvents` on the bus ARN and, if the bus uses a customer managed key, the key's policy must allow the `events.amazonaws.com` principal.

Scenario	Who needs the grant	Grants
Publish to a bus	The producer's identity	`events:PutEvents` or `events:PutRawEvents`

Scenario	Who needs the grant	Grants
		on the bus ARN; `events:DescribeEventBus` to confirm the bus is `ACTIVE`
Publish Avro or Protobuf with a schema registry	The role in `PutRawEvents RoleArn`	Read access to the schema in the registry: the Glue Schema Registry read actions for a Glue registry, or the secret and network access your Confluent registry requires. The role trusts `events.amazonaws.com`. See the section called "Supported content types"
Publish to a bus in another account	The producer's identity, and the bus	The publish action in the producer's identity policy, and the same action granted to the producer's account by an AWS RAM share or the bus resource policy

Scenario	Who needs the grant	Grants
Create and manage subscribers	The subscriber owner's identity	`events:CreateSubscriber` on the bus and on `arn:aws:events:` *region*`:`*account*`:subscriber/` *name*`/*`; `iam:PassRole` on the delivery role; `events:DescribeSubscriber`, `events:UpdateSubscriber`, `events:DeleteSubscriber` on the subscriber; `events:UpdateSubscriber` on the bus for an update that includes a `FilterConfiguration`; `events:TagResource` if the create carries tags
Deliver to a target	The delivery role	Trust for `events.amazonaws.com`; the target's action; `sqs:SendMessage` on the dead-letter queue; the target's key actions if the target is encrypted. See the section called “The delivery role”

Scenario	Who needs the grant	Grants
Attach an event source	The event source owner's identity, the dead-letter queue, and the key	`events:CreateEventSource` on the bus and on the event source; a queue policy allowing `events.amazonaws.com` `sqs:SendMessage` on the dead-letter queue; if the bus is encrypted, the key policy allows `events.amazonaws.com`. See the section called "Event sources"
Administer a bus	The bus owner's identity	`events:CreateEventBus`, `events:UpdateEventBus`, `events:DeleteEventBus`, `events:PutResourcePolicy`, `events:GetResourcePolicy`, `events:DeleteResourcePolicy`, `events:ListResourcePolicies`, `events:RevokeResource`; `ram:CreateResourceShare` and related AWS RAM actions to share it

Scenario	Who needs the grant	Grants
Encrypt a bus with your own key	The key policy	The bus creator's identity needs `kms:DescribeKey` ; that identity, every producer and subscriber owner, and the `events.amazonaws.com` principal need `kms:Decrypt` , `kms:Encrypt` , `kms:GenerateDataKeyWithoutPlaintext` , `kms:ReEncryptFrom` , and `kms:ReEncryptTo` , scoped to the bus with the `aws:events:event-busv2:arn` encryption context. See the section called “Configuring permissions to use a Customer Managed KMS Key”

For the full action list and condition keys, see the section called “Actions and condition keys”. For sharing, see the section called “Sharing a bus”.

Encryption at Rest for Amazon EventBridge Custom Event Bus

Options for Encryption at Rest

Amazon EventBridge Custom Event Bus encrypts all of your data at rest by default, server side, before it is written to storage: event payloads, event metadata, subscriber filter patterns, and subscriber target configurations. You do not need to configure anything, create any keys, or grant any permissions to get encryption at rest, and there is no additional cost for this default encryption.

By default, your data is encrypted using 256-bit Advanced Encryption Standard (AES-256) under an **AWS owned key**, a KMS key that EventBridge owns and manages for you. You cannot view,

manage, or use this key, and its use does not appear in your AWS CloudTrail logs. AWS owned keys are free of charge and do not count against the AWS KMS quotas in your account. AWS owned keys are automatically rotated monthly.

For more information, see AWS owned keys in the *AWS Key Management Service Developer Guide*.

Optionally, you can choose to encrypt an event bus with a symmetric **customer managed KMS key** that you create, own, and manage. A customer managed key gives you:

- Full control over the key lifecycle: key policy, rotation, disabling, deletion.
- CloudTrail visibility into how EventBridge uses your key.
- The ability to scope access with KMS key policy conditions, per event bus.

Customer managed keys incur AWS KMS charges: a monthly fee for the key, and charges for the KMS API requests made to it. These requests also count against the AWS KMS request quotas in your account. Because EventBridge encrypts and decrypts events locally under a cached branch key (see the section called "How Custom Event Bus uses a Customer Managed KMS Key for encryption at rest"), its KMS request volume is a few calls per event bus per cache window plus key rotation — it does not grow with your event volume. For details, see AWS Key Management Service Pricing.

Encrypting Data at Rest using Customer Managed KMS Keys for Custom Event Bus

How Custom Event Bus uses a Customer Managed KMS Key for encryption at rest

What is encrypted with your key (per event bus):

- Event payloads and event metadata, stored for the bus's retention period (1-365 days) in the bus's event storage and in per-subscriber delivery storage.
- Subscriber filter patterns and target configurations, stored with the bus's configuration.

What is not encrypted with your key:

- Resource identifiers that EventBridge needs for routing and lookup: bus name, bus ARN, account ID.
- The message group ID, which orders FIFO delivery. Avoid placing sensitive data in this field.
- The message deduplication ID. EventBridge stores it as a one-way hash.

- Internal service metadata: subscription match results, sequence numbers.

These items are still encrypted at rest by the underlying storage services, not by your key.

How encryption works. When you configure a customer managed key on an event bus, EventBridge creates a **branch key** for that bus. The branch key is an intermediate key. It is stored in a service-managed table, wrapped by your KMS key, following the AWS Encryption SDK Hierarchical Keyring. EventBridge uses the branch key to derive per-event data encryption keys. So events are encrypted and decrypted locally, without a KMS call per event:

- When you publish an event, EventBridge encrypts it under the bus's branch key before any write to storage.
- When EventBridge delivers an event, it decrypts the event for filter evaluation and delivery.
- **EventBridge sends the decrypted event to your target.** Encryption at rest at the destination is the destination service's responsibility.
- The branch key is cached in service memory for up to **15 minutes**. On cache expiry, EventBridge calls `kms:Decrypt` against your key to unwrap the branch key again.
- EventBridge rotates the branch key by creating a new version approximately every **30 days**. Older versions are retained so previously written events remain readable for the bus's retention period.

Two access paths. EventBridge accesses your key in exactly two ways. For synchronous operations and authorization checks, it uses **your credentials**, forwarded from your API call. For asynchronous work (event delivery, the branch-key lifecycle) it acts as the **`events.amazonaws.com` service principal**, because your API call has completed and your credentials are no longer available. Your key policy governs both paths, as described below.

Configuring a Customer Managed KMS Key on an event bus

Supported key types:

- **Symmetric encryption keys only** (`SYMMETRIC_DEFAULT`, key usage `ENCRYPT_DECRYPT`). Asymmetric keys and HMAC keys are rejected.
- Multi-Region keys are supported as ordinary keys: use the replica in the event bus's Region. EventBridge does not replicate your data or your key material across Regions.
- Keys with imported key material and keys in an external key store (XKS) are supported.

Key identifier. Specify the key by its **key ARN**.

Configuring permissions to use a Customer Managed KMS Key

Your key policy needs three statements for EventBridge Custom Event Bus. Replace the example account, Region, role, and bus name.

An encryption context is a set of non-secret key-value pairs that AWS KMS cryptographically binds to a request, and that you can use as a condition for authorization in key policies. Every KMS request EventBridge makes for an event bus includes an **encryption context** that binds the request to that bus's ARN, so each statement below can be scoped to a single bus. The context key is `aws:events:event-busv2:arn` and its value is the bus ARN, on every request EventBridge makes — whether with your credentials or as the service principal. the section called “Scoping down access to the Customer Managed KMS Key” covers scoping in more detail. The trailing `/*` in the encryption-context values matches the bus ARN's suffix, which EventBridge generates at creation (`arn:aws:events:`*`region`*`:`*`account`*`:event-busv2/`*`name`*`/`*`suffix`*).

```
{
  "Version": "2012-10-17",
  "Id": "eventbridge-eventbus-v2-cmk-policy",
  "Statement": [
    {
      "Sid": "AllowKeyValidationThroughEventBridge",
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::111122223333:role/ExampleRole" },
      "Action": "kms:DescribeKey",
      "Resource": "*",
      "Condition": {
        "StringEquals": { "kms:ViaService": "events.us-east-1.amazonaws.com" }
      }
    },
    {
      "Sid": "AllowKeyUseThroughEventBridge",
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::111122223333:role/ExampleRole" },
      "Action": [
        "kms:Decrypt",
        "kms:Encrypt",
        "kms:GenerateDataKeyWithoutPlaintext",
        "kms:ReEncryptFrom",
        "kms:ReEncryptTo"
      ],
```

```
      "Resource": "*",
      "Condition": {
        "StringEquals": { "kms:ViaService": "events.us-east-1.amazonaws.com" },
        "StringLike": {
          "kms:EncryptionContext:aws:events:event-busv2:arn": "arn:aws:events:us-
east-1:111122223333:event-busv2/my-bus/*"
        }
      }
    },
    {
      "Sid": "AllowEventBridgeServicePrincipal",
      "Effect": "Allow",
      "Principal": { "Service": "events.amazonaws.com" },
      "Action": [
        "kms:Decrypt",
        "kms:Encrypt",
        "kms:GenerateDataKeyWithoutPlaintext",
        "kms:ReEncryptFrom",
        "kms:ReEncryptTo"
      ],
      "Resource": "*",
      "Condition": {
        "StringLike": {
          "kms:EncryptionContext:aws:events:event-busv2:arn": "arn:aws:events:us-
east-1:111122223333:event-busv2/my-bus/*"
        }
      }
    }
  ]
}
```

Purpose of each statement:

- **`AllowKeyValidationThroughEventBridge`** lets your principals validate the key when creating or updating an event bus. EventBridge calls `kms:DescribeKey` with your credentials to confirm the key exists, is symmetric, and is enabled. The `DescribeKey` API has no encryption-context parameter. So this statement has no encryption-context condition. The `kms:ViaService` condition limits this statement's grant to requests that come through EventBridge.
- **`AllowKeyUseThroughEventBridge`** authorizes the access checks EventBridge runs **with your credentials**. EventBridge runs these checks before it accepts a key configuration and before it writes or reads encrypted data on your behalf. So you cannot configure a key that EventBridge's

later, asynchronous processing cannot use. The checks are KMS *dry runs*: they authorize but do not encrypt or decrypt anything. Dry-run requests are billed as standard KMS API requests and count against your KMS request quotas; see Testing your permissions in the *AWS Key Management Service Developer Guide*. Per operation:

- `CreateEventBus` with a customer managed key checks `kms:GenerateDataKeyWithoutPlaintext`, `kms:ReEncryptFrom`/`kms:ReEncryptTo`, and `kms:Decrypt`. These are the actions that branch-key creation and later reads perform.
- An `UpdateEventBus` key change checks `kms:Decrypt` on the current key, plus `kms:Encrypt` and `kms:Decrypt` on the new key. These are the actions the key transition performs.
- `PutEvents` / `PutRawEvents` on a bus with a customer managed key checks `kms:Decrypt`. A caller who cannot read the bus's events back may not write them either.
- `CreateSubscriber` / `UpdateSubscriber` checks `kms:Decrypt`, because subscriber configuration is encrypted under the bus key.

Choosing principals for this statement. You can list individual role or user ARNs in the `Principal`, or use the account root principal to delegate this access to IAM policies in your account.

- **`AllowEventBridgeServicePrincipal`** authorizes the work EventBridge performs **as the service** after your API call completes, when your credentials are no longer available:
  - `kms:GenerateDataKeyWithoutPlaintext` and `kms:ReEncryptFrom`/`kms:ReEncryptTo`: EventBridge creates the bus's branch key, and creates new branch-key versions on rotation.
  - `kms:Decrypt`: EventBridge unwraps the branch key to deliver events, to load subscriber configuration, and to return decrypted configuration on describe calls.
  - `kms:Decrypt` and `kms:Encrypt`: EventBridge re-wraps the branch key when you change the bus's KMS key.

  This statement uses the same encryption-context condition as `AllowKeyUseThroughEventBridge`. Every request EventBridge makes for a bus, on either access path, carries `aws:events:event-busv2:arn` with the bus ARN, so both statements bind key use to the specific event bus.

Cross-account keys: the key can be in a different account than the event bus (for example, a central security account). In that case, put all three statements in that key's policy. The first two

statements' `Principal` entries name the bus account's roles. The encryption-context values name the bus account's bus ARN. Cross-account KMS access requires permission on both sides: in addition to the key policy statements above, the bus account's roles need IAM identity policies that allow the same KMS actions on the key's ARN. Access the key by its full key ARN.

Creating a new event bus with a Customer Managed KMS Key

Specify the key ARN in the `EncryptionConfiguration` parameter of `CreateEventBus`:

```
{
  "Name": "my-bus",
  "EncryptionConfiguration": {
    "KmsKeyIdentifier": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
  }
}
```

At creation time, EventBridge validates the key and your access with your credentials (the `DescribeKey` and dry-run checks above). EventBridge then provisions the bus's branch key asynchronously, as the service principal. If the service principal statement is missing from your key policy, the bus transitions to `CREATE_FAILED`, and `DescribeEventBus` returns a `StateReason` naming the KMS access failure.

`DescribeEventBus` on a bus with a customer managed key returns its `EncryptionConfiguration`. A bus using the default AWS owned key returns no `EncryptionConfiguration`: EventBridge never returns the AWS owned key.

Changing the encryption configuration on an existing event bus

All transitions are supported, via `UpdateEventBus`:

- AWS owned key → customer managed key
- Customer managed key → different customer managed key
- Customer managed key → AWS owned key: send `EncryptionConfiguration` **present and empty** (`"EncryptionConfiguration": {}`). Omitting the member entirely means "leave the encryption configuration unchanged".

When you change the key, the bus enters `UPDATING`. EventBridge re-wraps the branch key under the new key. The bus then returns to `ACTIVE`. The branch key itself does not change, so the material protecting your data does not change. Only the wrapping key changes. Consequences:

- **Previously written events remain readable** after the transition, regardless of the old key's state. Once the transition completes you can disable the old key immediately without losing access to existing events.
- New events continue under the same branch key, now protected by the new KMS key.
- No re-encryption of stored events takes place, so transitions complete quickly and do not depend on how much data the bus holds.

Automatic KMS key rotation: if you enable automatic key rotation on your key, KMS rotates the backing material behind the same key ARN. The rotation does not immediately affect the branch key. The branch key stays wrapped by the material that wrapped it last. The next branch-key version, created approximately every 30 days, is protected by the current backing material. So KMS-side rotation takes effect on your bus within roughly a month.

If your key becomes unavailable (disabled, deleted, or access removed):

- Publishing to the bus fails with a KMS access error once the cached branch key expires, within 15 minutes.
- Event delivery pauses at the same point, for the same reason.
- Events EventBridge already accepted are not dropped, and decrypt failures are not routed to dead-letter queues. EventBridge retries them. A key-access failure is a condition you can fix, so EventBridge holds and retries the affected events rather than diverting them. No event is lost unless the outage outlasts the bus's retention period.
- After you restore access, processing resumes automatically, for as long as the events remain within the bus's retention period.

Deleting your key is permanent. Disabling a key or removing EventBridge's access is reversible: restore access, and processing resumes. Deleting the key is not. After AWS KMS deletes the key currently configured on a bus, EventBridge can no longer unwrap the bus's branch key, and every event still stored on the bus becomes permanently unreadable for the remainder of its retention period. As a best practice, disable the key first and confirm the impact before you schedule it for deletion. See Deleting AWS KMS keys in the *AWS Key Management Service Developer Guide*.

Deleting a bus does not require the key. `DeleteEventBus` succeeds even if the key is disabled or deleted.

Scoping down access to the Customer Managed KMS Key

- **Encryption context.** Every KMS request EventBridge makes for an event bus includes an encryption context binding the request to that bus's ARN. So you can scope each statement to a single bus, as in the example policy. The context key is `aws:events:event-busv2:arn` on every request, whether made with your credentials or as the service principal. Use `StringLike` with a trailing `/*`, because the bus ARN ends in a suffix generated at creation. On an existing bus you can match the exact ARN from `DescribeEventBus` with `StringEquals`.
- **kms:ViaService.** All requests made with your credentials populate the `kms:ViaService` condition key with `events.region.amazonaws.com`. So you can restrict your principals' access to key use through EventBridge only.
- **Propagation of key-policy changes.** Individual event operations use the cached branch key and do not re-validate against your key policy per event. A key-policy change — including removing EventBridge's access — takes effect for a bus's data handling when the cached branch key expires, within 15 minutes.
- **Confused deputy protection.** Requests EventBridge makes as the service principal include the `aws:SourceArn` context (the bus ARN) and the `aws:SourceAccount` context. So you can add these conditions to the `AllowEventBridgeServicePrincipal` statement. They ensure EventBridge uses your key only on behalf of your own resources:

```
"Condition": {
  "StringEquals": { "aws:SourceAccount": "111122223333" },
  "ArnLike": { "aws:SourceArn": "arn:aws:events:us-east-1:111122223333:event-busv2/
my-bus/*" }
}
```

Monitoring EventBridge Custom Event Bus interaction with AWS KMS

EventBridge's use of your customer managed key appears in AWS CloudTrail. See Logging AWS KMS API calls with AWS CloudTrail. Events to expect:

CloudTrail eventName	When	Calling identity
DescribeKey	Bus create/update key validation	Your principal, via EventBridge

CloudTrail eventName	When	Calling identity
Decrypt, Encrypt, GenerateDataKeyWithoutPlaintext, ReEncrypt (dry run)	Authorization checks on bus create/update, publish, subscriber create/update	Your principal (via EventBridge)
GenerateDataKeyWithoutPlaintext, ReEncrypt	Branch-key creation (bus create) and branch-key rotation (~30 days)	events.amazonaws.com service principal
Decrypt	Branch-key unwrap on cache expiry: event delivery, subscriber-configuration load, describe-time decryption	events.amazonaws.com service principal
Decrypt + Encrypt	Branch-key re-wrap on UpdateEventBus key change	events.amazonaws.com service principal

CloudTrail records the ReEncrypt API call as one event. The kms:ReEncryptFrom and kms:ReEncryptTo permissions in your key policy both authorize that call.

The authorization checks are KMS dry-run requests. By design, a dry-run request returns DryRunOperationException instead of performing the operation: that response is the trace of a *passing* check, not an error in your configuration. A dry-run entry showing AccessDeniedException instead means the caller lacks the permission being checked. For details, see Testing your permissions in the *AWS Key Management Service Developer Guide*.

The requestParameters.encryptionContext of these entries includes aws:events:event-busv2:arn with your bus ARN. Entries for branch-key operations also include branch-key metadata added by the Hierarchical Keyring: branch-key-id (the bus ARN), type, create-time, tablename, hierarchy-version, and kms-arn.

Individual event encryption and decryption use data keys derived locally from the cached branch key. So per-event operations do not appear in CloudTrail. You see branch-key operations only.

Observability for the Custom Event Bus: metrics, logs, and CloudTrail

Three sources tell you what a bus and its subscribers are doing. Amazon CloudWatch metrics in the `AWS/EventsV2` namespace are on for every subscriber; to alarm when deliveries fail, alarm on `OnFailureDestinationDelivered` or `EventsDropped`, or on `EventDeliveryAttempts` minus `EventsDelivered` with metric math. Subscriber logs record each delivery attempt as a JSON record; they are off until you set `LogConfiguration.Level` on the subscriber. AWS CloudTrail records the API calls that create and change buses, subscribers, and event sources.

Metrics

Subscriber metrics carry two dimensions, `EventBus` and `Subscriber`, each holding the resource ARN. When the subscriber belongs to a different account from the bus, EventBridge also emits the metric with `EventBus` and `SubscriberAccount` so that the bus owner can see consumption per account. The following table lists the metrics.

Metric	Unit	Meaning
`FilterEvaluated`	Count	Events evaluated against the subscriber's filters
`FilterMatched`	Count	Events that matched every filter
`EventDeliveryAttempts`	Count	Delivery attempts, counted per event. Subtract `EventsDelivered` to count failed attempts
`TargetInvocations`	Count	Calls made to the target; one call can carry a batch of events
`RetryInvocationAttempts`	Count	Emitted only on a retry, valued as the retry depth: the second attempt is 1, the third

Metric	Unit	Meaning
		is 2. The sample count is the number of retries
EventsDelivered	Count	Events the target accepted
EgressBytes	Bytes	Bytes delivered to the target
EventTransformationFailures	Count	Events whose Transformer or universal-target Input expression failed
IngestionToInvocationStartTime	Milliseconds	Time from the event's arrival on the bus to the start of the target call. Not emitted when the attempt failed before calling the target
IngestionToInvocationEndTime	Milliseconds	Time from the event's arrival on the bus to the end of the target call
OnFailureDestinationDelivered	Count	Records written to the dead-letter queue
OnFailureDestinationFailed	Count	Records that could not be written to the dead-letter queue
EventsDropped	Count	Events that exhausted retries with no dead-letter queue to record them
ApproximateBacklogAge	Milliseconds	Age of the oldest event the subscriber has not yet delivered

Metric	Unit	Meaning
SubscriberLogRecordsDropped	Count	Log records that could not be written, because log delivery is best effort

Your account's bus count is also reported in the AWS/Usage namespace as ResourceCount with Service EventBridge and a Resource value that begins with EventsV2/, so you can alarm as you approach the bus quota. For the quotas, see the section called “Custom Event Bus quotas”.

Publish metrics

Every PutEvents and PutRawEvents call also produces metrics in the AWS/EventsV2 namespace, so you can alarm on throttled or failed publish calls for a bus, for example on PublishEventsApproximateThrottledCallCount with the EventBus dimension set to orders. The bus owner's account receives every publish metric. An account that publishes to a bus it does not own also receives them, in its own account, for its own calls.

Metric	Unit	Meaning
PublishEventsApproximateCallCount	Count	Publish calls received
PublishEventsApproximateSuccessCallCount	Count	Publish calls that returned HTTP 200
PublishEventsApproximateFailedCallCount	Count	Publish calls that returned an error
PublishEventsApproximateThrottledCallCount	Count	Publish calls rejected with ThrottlingException
PublishEventsEntryCount	Count	Entries in the publish calls

Metric	Unit	Meaning
PublishEventsFailedEntriesCount	Count	Entries that failed inside an accepted call, reported in the response as failed entries
PublishEventsIngressBytes	Bytes	Bytes of event payload stored. Absent, rather than 0, for calls that stored nothing. Trends with the ingress line item on your bill, which rounds each entry up to a whole KB.

The dimensions depend on the metric and on who published.

- Every metric is published with the EventBus dimension alone: the totals for the bus.
- For events that arrive through an event source, the count metrics are also published with EventBus and EventSource, so you can see one source's share. PublishEventsIngressBytes has no EventSource breakdown.
- PublishEventsIngressBytes is also published to the bus owner with EventBus and PublisherAccount, so a bus owner can see how many bytes each publishing account stored.

Subscriber logs

A subscriber can record what happened to each event it tried to deliver. Logging is per subscriber and off when you create one, so a new subscriber records nothing until you set LogConfiguration. Each record is a JSON document with a message_type that says what it describes.

- SUBSCRIBER_MATCHED: an event matched the subscriber's filters and entered delivery.
- EVENT_DELIVERY_ATTEMPT: one attempt to invoke the target, with its outcome, attempt count, and duration. This is the record the rest of this section describes.
- EVENT_TRANSFORMATION_FAILURE: the Transformer or universal-target Input expression failed for an event, with the error.

- ON_FAILURE_DESTINATION_DELIVERY_ATTEMPT: one attempt to write a record to the dead-letter queue.

Log delivery is best effort. A record that cannot be written is counted in the SubscriberLogRecordsDropped metric rather than retried indefinitely. When the bus is encrypted with a customer managed key, EventBridge encrypts the payload fields of each record under that key before it leaves EventBridge, so a destination account that cannot use the key sees the record without them. A replayed event's record carries details.delivery_type REPLAY; a live event's carries LIVE.

The two settings that produce a record

Two independent settings must both exist before you can read a single record.

1. **The subscriber's log level.** LogConfiguration.Level decides which records EventBridge emits. It defaults to OFF, which emits none.
2. **A CloudWatch Logs delivery.** Records reach you through the Amazon CloudWatch Logs vended log delivery mechanism, which pairs the subscriber with a destination you own. Without it, records have nowhere to land and no log group appears on its own.

Make both settings before you start diagnosing a delivery problem rather than after. Only subscribers have a log configuration. Buses and event sources have none, so you turn logging on one subscriber at a time. LogConfiguration has two members.

Level

The minimum level of a record. Records below it are not emitted. OFF, the default, emits nothing. ERROR emits failed delivery attempts only. INFO emits every delivery attempt, including the ones that succeeded.

IncludePayload

Whether a record carries your event payload. ON_ERROR_ONLY, the default, carries it on failure records only. FULL carries it on every record. See the section called “Payloads in log records”.

You can set LogConfiguration when you create a subscriber, or later with UpdateSubscriber. An update takes effect without recreating the subscriber. Neither the create response nor the

update response echoes the field back, so confirm the stored value with `DescribeSubscriber`, which does return it.

Turn on logging for one subscriber

The following steps record every delivery attempt for one existing subscriber and deliver the records to a CloudWatch Logs log group in the same account. The first command is an EventBridge call, and the rest are CloudWatch Logs calls. Start by raising the subscriber's log level. `INFO` records successes as well as failures, which is what tells you whether an event was delivered at all.

```
aws eventsv2 update-subscriber \
    --subscriber-arn arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLE1234567890abcdef \
    --log-configuration '{ "Level": "INFO", "IncludePayload": "FULL" }'
```

Next, create the destination log group. The name must begin with `/aws/vendedlogs/`. CloudWatch Logs manages the delivery resource policy for you only under that prefix; for a log group outside it you must manage that policy yourself.

```
aws logs create-log-group \
    --log-group-name /aws/vendedlogs/large-orders-delivery
```

Create a delivery source. Its `--resource-arn` is the subscriber ARN, which is what makes the source produce that subscriber's records. Set `--log-type` to match the level you configured: `INFO_LOGS` for `Level` `INFO`, or `ERROR_LOGS` for `Level` `ERROR`.

```
aws logs put-delivery-source \
    --name large-orders-source \
    --resource-arn arn:aws:events:us-east-1:111122223333:subscriber/large-orders/
EXAMPLE1234567890abcdef \
    --log-type INFO_LOGS
```

Create a delivery destination naming the log group. The response contains the destination ARN, which the next command needs.

```
aws logs put-delivery-destination \
    --name large-orders-destination \
    --delivery-destination-configuration destinationResourceArn=arn:aws:logs:us-
east-1:111122223333:log-group:/aws/vendedlogs/large-orders-delivery
```

Finally, pair the source with the destination. Use the destination ARN from the previous response.

```
aws logs create-delivery \
    --delivery-source-name large-orders-source \
    --delivery-destination-arn destination-arn
```

Records now appear in the log group. They arrive later than the delivery itself, because the log pipeline batches them, so allow for that lag when you read a log group just after publishing. Three properties of this wiring decide how far it stretches. A destination is a log group, an Amazon S3 bucket, or an Amazon Data Firehose stream, and the commands above are the same for each: only `destinationResourceArn` changes. One delivery pairs exactly one source with exactly one destination, so sending a subscriber's records to a second destination takes a second delivery. The destination can be in a different account from the subscriber, and in that case the destination account must call `PutDeliveryDestinationPolicy` on the destination to allow the delivery, which is how a central account collects records for subscribers it does not own.

Reading a delivery record

EventBridge emits one record per delivery attempt per event. Four fields are at the top level. `message_type` is `EVENT_DELIVERY_ATTEMPT` for a delivery attempt; the other types are listed at the start of this section. `resource_arn` is the ARN of the subscriber that made the attempt, which is how you separate subscribers that share one log group. `log_level` is the level of the record itself: a successful attempt is `INFO`, and a failed attempt is `ERROR`, which is why `Level ERROR` still records failures. `details` is an object holding the delivery state, and it holds the fields you build queries and alarms on.

`outcome` and `attempt_count`

: `SUCCESS` or `FAILURE` for this attempt, and which attempt it was, so retries are countable.

`terminal_kind`

: Present on the one record that says how the event ended: `EVENTS_DELIVERED`, `ON_FAILURE_DESTINATION_DELIVERED`, or `EVENTS_DROPPED` when retries ran out and no on-failure destination was configured.

`target_arn` and `target_properties`

: The target the attempt was made against, and the target parameters the subscriber resolved for this event. `target_properties` is omitted when the subscriber configures no parameters.

`ingestion_to_start_latency_ms` and `ingestion_to_complete_latency_ms`

Milliseconds from event ingestion to the start and to the end of this attempt.

`target_input`

The bytes sent to the target, verbatim. This is a payload field; see the section called "Payloads in log records".

`event_detail`, `event_metadata`, and `event_system_metadata`

The event as EventBridge held it. `event_system_metadata` carries the identifiers you correlate records by. `event_detail` appears only when it differs from `target_input`, which is the case when the subscriber has a transformer.

To diagnose a delivery that never arrived, read the subscriber's records in this order. `outcome` says whether EventBridge reached the target at all. `attempt_count` says whether it is still retrying, because an event with no `terminal_kind` record has not finished. `terminal_kind` says how it ended, and separates an event that dead-lettered from one that was dropped. `target_arn`, `target_properties`, and `target_input` say what was sent and where, which is where a transformer or target parameter problem shows up. A failure is visible in the record as soon as the attempt fails, so you do not have to wait for the retries to run out. For the full procedure, see the section called "Troubleshooting: you published and nothing arrived at the target".

Payloads in log records

`IncludePayload` controls two fields and no others: `details.target_input` and `details.event_detail`. Every other field is emitted whichever value you choose, so a subscriber that keeps payloads out of its records still reports the outcome, the target, the attempt count, and the latencies. `FULL` embeds the payload in every record, including the records for successful deliveries. `ON_ERROR_ONLY`, the default, embeds it in failure records only, and omits both fields from a record for a successful delivery.

> **Important**
>
> A log record that carries a payload carries your data. With `FULL`, the log group holds a copy of every event body the subscriber delivered, and anyone who can read the log group can read those bodies. Use `FULL` while you are debugging a subscriber, then return it to `ON_ERROR_ONLY`. That limits both the exposure and the volume you store.

API calls in AWS CloudTrail

Subscriber logs record deliveries, not the calls you make to configure them. AWS CloudTrail records the management API calls, so a change to a bus, a subscriber, or an event source is auditable from your trail rather than from a log group. For how EventBridge integrates with CloudTrail, see the section called “CloudTrail logs”.

Creating Custom Event Bus resources with CloudFormation

You can create a bus, its subscribers, its event sources, and its resource policy from a CloudFormation template. The resource types are in the `AWS::EventsV2` namespace. The `AWS::Events` types belong to Custom Event Bus - Classic and cannot configure a Custom Event Bus.

The resource types

- `AWS::EventsV2::EventBus` is one bus, including its retention period and its encryption key. It requires `Name`.
- `AWS::EventsV2::Subscriber` is one subscriber on a bus: its filters, its transform, its retry policy, and its single target. It requires `Name`, `EventBusArn`, and `InvokeConfiguration`, which itself requires `TargetArn` and `RoleArn`.
- `AWS::EventsV2::EventSource` is one managed feed of AWS service events or partner events into a bus. It requires `Name`, `EventBusArn`, and `Configuration`. See the section called “Event sources”.
- `AWS::EventsV2::ResourcePolicy` is the resource policy of a bus that already exists. It requires `EventBusArn` and `PolicyDocument`. See the section called “Access control”.

A subscriber's `RoleArn` must name a role in the same account as the subscriber. So a bus shared across accounts is more than one stack: the bus in the owner's account, and the target, the delivery role, and the subscriber in each consuming account.

Example: a bus, a queue, and a subscriber

The following template creates a bus that retains events for seven days, an Amazon SQS queue, a dead-letter queue, a delivery role, and a subscriber that delivers orders over 500 to the queue. CloudFormation creates the role and the queues before the subscriber, because the subscriber

refers to them with `Fn::GetAtt`. The retention period decides how far back a subscriber can start reading; see the section called “Replaying events”.

```
AWSTemplateFormatVersion: '2010-09-09'

Resources:
  OrdersBus:
    Type: AWS::EventsV2::EventBus
    Properties:
      Name: orders
      Description: Order events for the fulfillment platform
      StorageConfiguration:
        RetentionPeriodInDays: 7

  OrdersQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: large-orders

  OrdersDeadLetterQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: large-orders-dlq

  DeliveryRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: EventBusDeliveryRole
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: events.amazonaws.com
            Action: sts:AssumeRole
      Policies:
        - PolicyName: DeliverToQueues
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action: sqs:SendMessage
                Resource:
```

```
                  - !GetAtt OrdersQueue.Arn
                  - !GetAtt OrdersDeadLetterQueue.Arn

  LargeOrders:
    Type: AWS::EventsV2::Subscriber
    Properties:
      Name: large-orders
      EventBusArn: !Ref OrdersBus
      FilterConfiguration:
        Filters:
          - Scope: DATA
            Pattern: '{"detail":{"total":[{"numeric":[">",500]}]}}'
      InvokeConfiguration:
        TargetArn: !GetAtt OrdersQueue.Arn
        RoleArn: !GetAtt DeliveryRole.Arn
      RetryPolicy:
        MaxRetryAttempts: 10
        MaxEventAgeInSeconds: 3600
      OnFailureConfiguration:
        Arn: !GetAtt OrdersDeadLetterQueue.Arn

Outputs:
  OrdersBusArn:
    Value: !Ref OrdersBus
    Export:
      Name: orders-bus-arn
```

The delivery role trusts the `events.amazonaws.com` service principal and grants `sqs:SendMessage` on both queues. Without the grant on the dead-letter queue, a failed delivery cannot be recorded and nothing is observable. The identity that deploys the stack also needs `iam:PassRole` on the delivery role.

Return values

`Ref` returns the resource's primary identifier, an ARN. `Fn::GetAtt` returns the read-only properties. The following table lists the values for each resource type.

Resource type	Ref returns	Fn::GetAtt attributes
AWS::EventsV2::EventBus	EventBusArn	EventBusArn , State, CreationTime , LastModifiedTime
AWS::EventsV2::Subscriber	SubscriberArn	SubscriberArn , BusName, CreationTime , LastModifiedTime
AWS::EventsV2::EventSource	EventSourceArn	EventSourceArn , State, Revoked, CreationTime , LastModifiedTime
AWS::EventsV2::ResourcePolicy	EventBusArn	RevisionId

A subscriber's `State` is a property you set, not an attribute you read. Set it to `STOPPED` to pause delivery and back to `RUNNING` to resume. A bus's `State` is read-only, and its settled value is `ACTIVE`.

Properties that replace the resource

Six subscriber properties are create-only, so changing any of them in a template replaces the subscriber: `Name`, `EventBusArn`, `InvokeConfiguration.TargetArn`, `Type`, `StartingPosition`, and `PointInTimeConfiguration`. Every other subscriber property, including the filters, the transform, the retry policy, and the dead-letter queue, updates in place.

> **Important**
>
> A subscriber replacement deletes the old subscriber before it creates the new one. No subscriber exists between the two steps. The new subscriber then starts from its own `StartingPosition`, and `LATEST` starts from the newest events, so it never delivers the events published during that window. Changing a target ARN, an ordering type, or a starting position therefore drops events unless you plan for the gap.

To avoid the gap, deploy twice. First add the replacement subscriber under a new logical id and wait for its `State` to reach `RUNNING`. Then remove the old subscriber in a second deployment. Both subscribers deliver during the overlap, so the target receives each matching event twice; use this approach only when duplicates are acceptable.

If you cannot deploy twice, set the new subscriber's `StartingPosition` to `POINT_IN_TIME` and give `PointInTimeConfiguration` a `PointType` of `TIMESTAMP` with a `StartingPoint`, in seconds since the Unix epoch, from before the deployment. The subscriber then reads the events it missed, provided they are still within the bus's retention period.

On a bus, only `Name` is create-only, so the retention period, the description, the encryption key, and the tags all update in place. On an event source, `Name` and `EventBusArn` are create-only, and on a resource policy, `EventBusArn` is.

Referring to a bus from another stack

A subscriber needs the full bus ARN, which includes the service-generated identifier after the bus name. Store and pass the ARN, not the name. See the section called “Names, endpoints, and permissions”.

Within one account and Region, export the ARN from the stack that owns the bus, as the example does, and import it with `EventBusArn: !ImportValue orders-bus-arn`. CloudFormation does not let you change or remove an exported value while another stack imports it. Renaming the bus replaces it and changes its ARN, so a rename fails while an import exists. Remove the importing subscriber first, or export under a new name.

An export cannot be imported from another account or another Region. Pass the bus ARN into the consuming stack as a template parameter instead. The bus owner must also grant the other account permission on the bus, which you do with a resource policy or with AWS Resource Access Manager (AWS RAM). See the section called “Access control”.

Names, endpoints, and IAM permissions for the Custom Event Bus

The Custom Event Bus has its own AWS CLI command, SDK client, and endpoint, and it shares its IAM namespace and service principal with Custom Event Bus - Classic. The following table shows which name to use where.

Names to use for each interface

Interface	Name	Example
AWS CLI command	`eventsv2`	`aws eventsv2 put-events`
SDK client	`EventBridgeV2`	`EventBridgeV2Client` in the AWS SDK for Java
Endpoint	`eventsv2`	`eventsv2.us-east-1.amazonaws.com`
VPC endpoint service	`eventsv2`	`com.amazonaws.us-east-1.eventsv2` , or `com.amazonaws.us-east-1.eventsv2-fips`
Dual-stack endpoint	`eventsv2`	`eventsv2.us-east-1.api.aws`
IAM actions	`events:`, shared with Custom Event Bus - Classic	`events:PutRawEvents` , `events:CreateSubscriber`
Service principal	`events.amazonaws.com` , shared with Custom Event Bus - Classic	The principal in a delivery role's trust policy
Resource ARNs	`event-busv2`	`arn:aws:events:us-east-1:111122223333:event-busv2/orders/EXAMPLE1234567890abcdef`
CloudFormation resource types	`AWS::EventsV2`	Resource types in the `AWS::EventsV2` namespace

Why the IAM namespace is shared with Custom Event Bus - Classic

EventBridge keeps the `events:` namespace and the `events.amazonaws.com` principal for the Custom Event Bus so that you can move from Custom Event Bus - Classic without rewriting three things:

- Identity-based policies that grant `events:` actions.
- Bus resource policies.
- The roles that deliver to your targets. A role that Custom Event Bus - Classic can assume today can deliver from a Custom Event Bus subscriber with no change to its trust policy.

> **Important**
>
> Write every IAM action with the `events:` prefix, for example `events:PutRawEvents` and `events:CreateSubscriber`. The `eventsv2` name is for the CLI command, the SDK client, and the endpoint only. IAM accepts a policy that names `eventsv2:PutEvents`, but no such action exists, so the policy grants nothing and the caller receives `AccessDeniedException`. When a call to `aws eventsv2` is denied, check the action prefix in the policy first.

The following identity-based policy allows a producer to publish to one bus with either publish API. For the full action list, condition keys, and cross-account grants, see the section called "Access control"; the authoritative list of actions, resources, and condition keys is in the Service Authorization Reference.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "events:PutEvents",
                "events:PutRawEvents"
            ],
            "Resource": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef"
        }
```

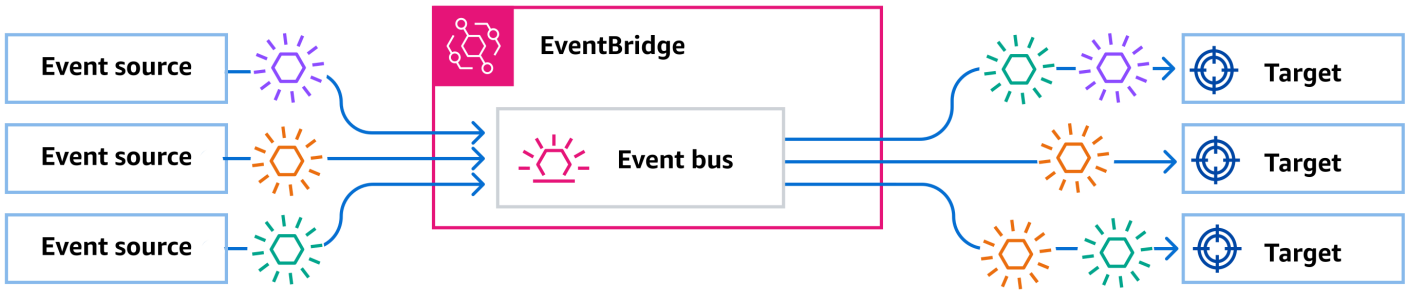
```
    ]
}
```

Publishing is authorized against the bus ARN for every entry in the request. If one entry fails an authorization condition, EventBridge denies the whole request.

Amazon EventBridge event bus (default and Classic)

This chapter describes the default event bus in every account and the Custom Event Bus - Classic buses you create: the EventBridge event buses that route events with rules and targets that the bus owner writes. EventBridge also offers the Custom Event Bus, which routes events to subscribers that consumers create themselves and retains events for replay. For new applications, start with the Custom Event Bus. To move existing rules to subscribers, see the section called "Migrating from Custom Event Bus - Classic".

A Classic event bus is a router that receives events and delivers them to zero or more destinations, or *targets*. Classic event buses are well-suited for routing events from many sources to many targets, with optional transformation of events prior to delivery to a target.



Rules associated with the event bus evaluate events as they arrive. Each rule checks whether an event matches the rule's pattern. If the event does match, EventBridge sends the event

You associate a rule with a specific event bus, so the rule only applies to events received by that event bus.

> **Note**
>
> You can also process events using EventBridge Pipes. EventBridge Pipes is intended for point-to-point integrations; each pipe receives events from a single source for processing and delivery to a single target. Pipes also include support for advanced transformations and enrichment of events prior to delivery to a target. For more information, see ???.

How event buses work in EventBridge

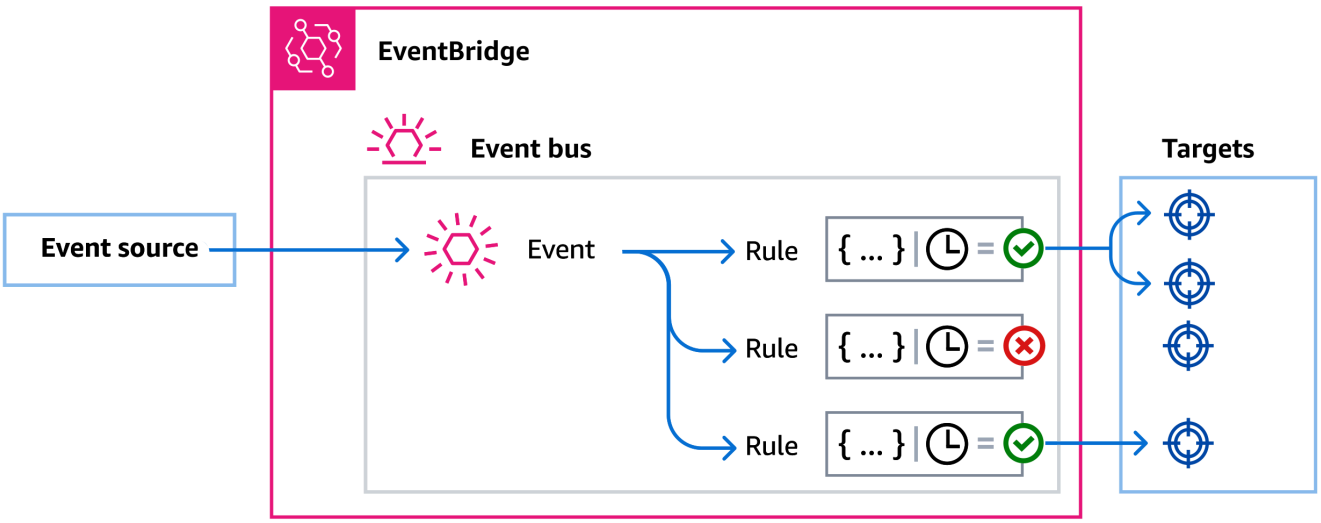
Event buses enable you to route events from multiple sources to multiple destinations, or *targets*.

At a high level, here's how it works:

1. An event source, which can be an AWS service, your own custom application, or a SaaS provider, sends an event to an event bus.
2. EventBridge then evaluates the event against each *rule* defined for that event bus.

   For each event that matches a rule, EventBridge then sends the event to the targets specified for that rule. Optionally, as part of the rule, you can also specify how EventBridge should transform the event prior to sending it to the target(s).

   An event might match multiple rules, and each rule can specify up to five targets. (An event may not match any rules, in which case EventBridge takes no action.)



Consider an example using the EventBridge default event bus, which automatically receives events from AWS services:

1. You create a rule on the default event bus for the `EC2 Instance State-change Notification` event:
   - You specify that the rule matches events where an Amazon EC2 instance has changed its `state` to `running`.

     You do this by specifying JSON that defines the attributes and values an event must match to trigger the rule. This is called an *event pattern*.

```
{
  "source": ["aws.ec2"],
  "detail-type": ["EC2 Instance State-change Notification"],
    "detail": {
      "state": ["running"]
  }
}
```

- You specify the target of the rule to be a given Lambda function.

2. Whenever an Amazon EC2 instance changes state, Amazon EC2 (the event source) automatically sends that event to the default event bus.
3. EventBridge evaluates all events sent to the default event bus against the rule you've created.

   If the event matches your rule (that is, if the event was an Amazon EC2 instance changing state to `running`), EventBridge sends the event to the specified target. In this case, that's the Lambda function.

Amazon EventBridge: What are event buses?

Amazon EventBridge: The differences between event buses

Event bus concepts in Amazon EventBridge

Here's a closer look at the main components of an event driven architecture built on event buses.

Event buses

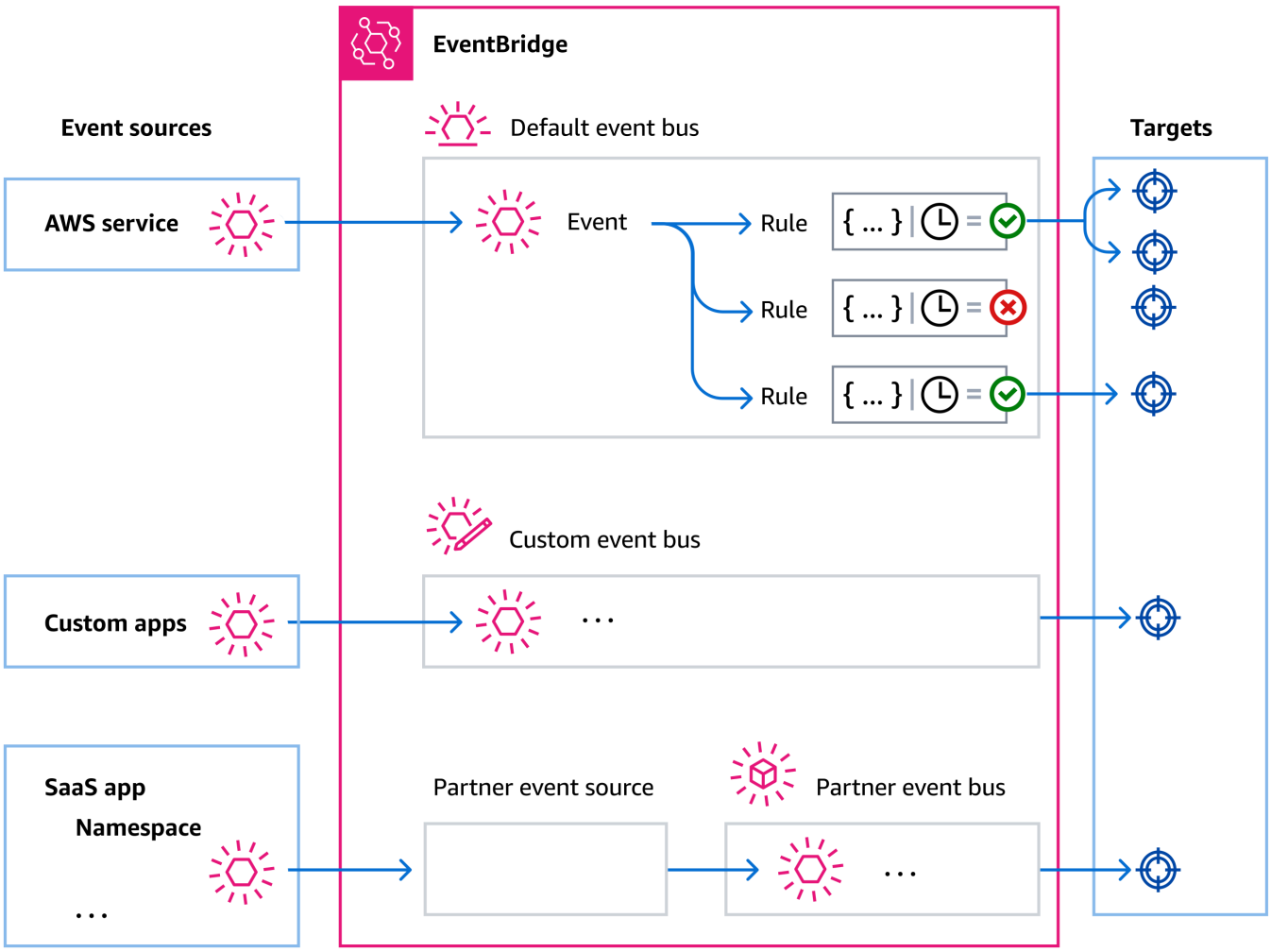
A Classic event bus is a router that receives events and delivers them to zero or more destinations, or *targets*. Use an event bus when you need to route events from many sources to many targets, with optional transformation of events prior to delivery to a target.

Your account includes a *default event bus* that automatically receives events from AWS services. You can also:

- Create additional event buses, called *custom event buses*, and specify which events they receive. A custom event bus on this page is a Custom Event Bus - Classic. It is not the Custom Event Bus.
- Create *partner event buses*, which receive events from SaaS partners.

Common use cases for event buses include:

- Using an event bus as a broker between different workloads, services, or systems.
- Using multiple event buses in your applications to divide up the event traffic. For example, creating a bus to process events containing personal identification information (PII), and another bus for events that don't.
- Aggregating events by sending events from multiple event buses to a centralized event bus. This centralized bus can be in the same account as the other buses, but can also be in a different account or Region.



Events

At its simplest, an EventBridge event is a JSON object sent to an event bus or pipe.

In the context of event-driven architecture (EDA), an event often represents an indicator of a change in a resource or environment.

For more information, see ???.

Event sources

EventBridge can receive events from event sources including:

- AWS services
- Custom applications
- Software as a service (SaaS) partners

Rules

A rule receives incoming events and sends them as appropriate to targets for processing. You can specify how each rule invokes their target(s) based on either:

- An event pattern, which contains one or more filters to match events. Event patterns can include filters that match on:
  - **Event metadata** – Data *about* the event, such as the event source, or the account or Region in which the event originated.
  - **Event data** – The properties of the event itself. These properties vary according to event.
  - **Event content** – The actual property *values* of the event data.
- A schedule to invoke the target(s) at regular intervals.

  You can specify a scheduled rule within EventBridge, or by using EventBridge Scheduler.

> **Note**
>
> Scheduled rules are a legacy feature of EventBridge.
> EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.

> Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
> For more information, see ??? or the *EventBridge Scheduler User Guide*.

Each rule is defined for a specific event bus, and only apply to events on that event bus.

A single rule can send an event to up to five targets.

By default, you can configure up to 300 rules per event bus. This quota can be raised to thousands of rules in the Service Quotas console. Since the rule limit apply to each bus, if you require even more rules, you can create additional custom event buses (each a Custom Event Bus - Classic) in your account.

You can customize how events are received in your account by creating event buses with different permissions for different services.

To customize the structure or date of an event before EventBridge passes it to a target, use the input transformer to edit the information before it goes to the target.

For more information, see ???.

Targets

A target is a resource or endpoint to which EventBridge sends an event when the event matches the event pattern defined for a rule.

A target can receive multiple events from multiple event buses.

For more information, see ???.

Advanced features for event buses

EventBridge includes the following features to help you develop, manage, and use event buses.

Using API destinations to enable REST API calls between services

EventBridge *API destinations* are HTTP endpoints that you can set as the target of a rule, in the same way that you would send event data to an AWS service or resource. By using API destinations,

you can use API calls to route events between AWS services, integrated SaaS applications, and your applications outside of AWS. When you create an API destination, you specify a connection to use for it. Each connection includes the details about the authorization type and parameters to use to authorize with the API destination endpoint.

Archiving and replaying events to aid development and disaster recovery

You can *archive*, or save, events and then replay them at a later time from the archive. Archiving is useful for:

- Testing an application because you have a store of events to use rather than having to wait for new events.
- Hydrating a new service when it first comes online.
- Adding more durability to your event-driven applications.

Using the Schema Registry to jump-start event pattern creation

When you build serverless applications that use EventBridge, it can be helpful to know the structure of typical events without having to generate the event. The event structure are described in schemas, which are available for all events generated by AWS services on EventBridge.

For events that don't come from AWS services, you can:

- Create or upload custom schemas.
- Use Schema Discovery to have EventBridge automatically create schemas for events sent to the event bus.

Once you have a schema for an event, you can download code bindings for popular programming languages.

Managing resources and access with policies

To organize AWS resources or to track costs in EventBridge, you can assign a custom label, or *tag*, to AWS resources. Using tag-based policies, you can control what resources can and can't do within EventBridge.

In addition to tag-based policies, EventBridge supports identity-based and resource-based policies to control access to EventBridge. Use identity-based policies to control the permissions of a group,

role, or user. Use resource-based policies to give specific permissions to each resource, such as a Lambda function or Amazon SNS topic.

Getting started: Create an Amazon EventBridge event bus rule

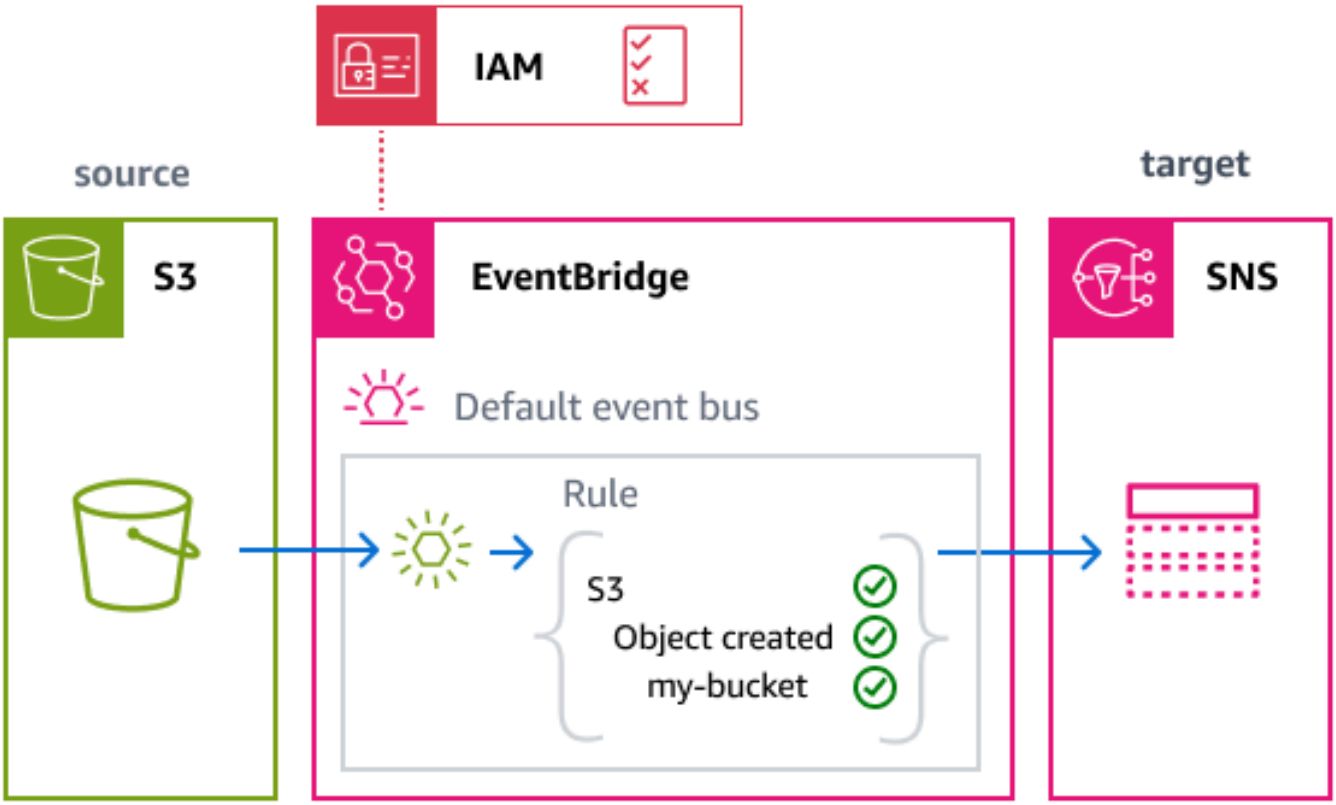
To get familiar with EventBridge rules and their capabilities, we'll use a CloudFormation template to set up an event bus rule and associated components, including an event source, event pattern, and target. Then we can explore how rules work to select the events you want.

The template creates a rule on the default event bus. This rule uses an event pattern to filter for events from a specific Amazon S3 bucket. The rule sends matching events to the specified target, an Amazon SNS topic. Every time an object is created in the bucket, the rule sends a notification to the topic, which then sends an email to your specified email address.

The deployed resources consist of:

- An Amazon S3 bucket with EventBridge notifications enabled to act as the event source.
- An Amazon SNS topic and email subscription as the target for notifications.
- An execution role that grants EventBridge the necessary permissions to publish to the Amazon SNS topic.
- The rule itself, which:
  - Defines an event pattern that matches only `Object Created` events from the specific Amazon S3 bucket.
  - Specifies the Amazon SNS topic as a target to which EventBridge delivers matching events.

For specific technical details of the template, see Template details.



Before you begin

To receive Amazon S3 events in EventBridge, you must enable EventBridge within Amazon S3. This topic assumes EventBridge is enabled. For more information, see Enabling EventBridge in the *Amazon S3 User Guide*.

Creating the rule using CloudFormation

To create the rule and its associated resources, we'll create a CloudFormation template and use it to create a stack containing a sample rule, complete with source and target.

> **Important**
>
> You will be billed for the Amazon resources used if you create a stack from this template.

Creating the template

First, create the CloudFormation template.

1. In the Template section, click the copy icon on the **JSON** or **YAML** tab to copy the template contents.
2. Paste the template contents into a new file.
3. Save the file locally.

Creating the stack

Next, use the template you've saved to provision a CloudFormation stack.

Create the stack using CloudFormation (console)

1. Open the CloudFormation console at https://console.aws.amazon.com/cloudformation/.
2. On the **Stacks** page, from the **Create stack** menu, choose **with new resources (standard)**.
3. Specify the template:
   a. Under **Prerequisite**, choose **Choose an existing template**.
   b. Under **Specify template**, choose **Upload a template file**.
   c. Choose **Choose file**, navigate to the template file, and choose it.
   d. Choose **Next**.
4. Specify the stack details:
   a. Enter a stack name.
   b. For **BucketName**, enter a globally unique bucket name. Amazon S3 bucket names must be unique across all AWS accounts.
   c. For **SNSTopicDisplayName**, **SNSTopicName**, and **RuleName**, accept the default values or enter your own.
   d. For **EmailAddress**, enter a valid email address where you want to receive notifications.
   e. Choose **Next**.
5. Configure the stack options:
   a. Under **Stack failure options**, choose **Delete all newly created resources**.

> **Note**
>
> Choosing this option prevents you from possibly being billed for resources whose deletion policy specifies they be retained even if the stack creation fails. For more information, see DeletionPolicy attribute in the *CloudFormation User Guide*.

b. Accept all other default values.

c. Under **Capabilities**, check the box to acknowledge that CloudFormation might create IAM resources in your account.

d. Choose **Next**.

6. Review the stack details and choose **Submit**.

Create the stack using CloudFormation (AWS CLI)

You can also use the AWS CLI to create the stack.

- Use the `create-stack` command.
  - Accept the default template parameter values, specifying the stack name, email address, and bucket name. Use the `template-body` parameter to pass the template contents, or `template-url` to specify a URL location.

```
aws cloudformation create-stack \
  --stack-name eventbridge-rule-tutorial \
  --template-body template-contents \
  --parameters \
    ParameterKey=EmailAddress,ParameterValue=your.email@example.com \
    ParameterKey=BucketName,ParameterValue=my-unique-bucket-name \
  --capabilities CAPABILITY_IAM
```

  - Override the default value(s) of one or more template parameters. For example:

```
aws cloudformation create-stack \
  --stack-name eventbridge-rule-tutorial \
  ----template-body template-contents \
  --parameters \
    ParameterKey=EmailAddress,ParameterValue=your.email@example.com \
    ParameterKey=BucketName,ParameterValue=my-custom-bucket-name \
    ParameterKey=RuleName,ParameterValue=my-custom-rule-name \
```

```
  --capabilities CAPABILITY_IAM
```

CloudFormation creates the stack. Once the stack creation is complete, the stack resources are ready to use. You can use the **Resources** tab on the stack detail page to view the resources that were provisioned in your account.

After the stack is created, you will receive a subscription confirmation email at the address you provided. You must confirm this subscription to receive notifications.

Exploring rule capabilities

Once the rule has been created, you can use the EventBridge console to observe rule operation and test event delivery.

1. Open the EventBridge console at https://console.aws.amazon.com/events/home?#/rules.
2. Choose the rule you created.

   On the rule detail page, the **Rule details** section displays information about the rule, including its event pattern and targets.

Examining the event pattern

Before we test the rule operation, let's examine the event pattern we've specified to control which events are sent to the target. The rule will only send events that match the pattern criteria to the target. In this case, we only want the event that Amazon S3 generates when an object is created in our specific bucket.

- On the rule detail page, under **Event pattern**, you can see the event pattern selects only events where:
  - The source is the Amazon S3 service (`aws.s3`)
  - The detail-type is `Object Created`
  - The bucket name matches the name of the bucket we created

```
{
  "source": ["aws.s3"],
  "detail-type": ["Object Created"],
```

```
  "detail": {
    "bucket": {
      "name": ["eventbridge-rule-example-source"]
    }
  }
}
```

Sending events through the rule

Next, we'll generate events in the event source to test that the rule matching and delivery is operating correctly. To do this, we'll upload an object to the S3 bucket we specified as the event source.

1. Open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. In the **Buckets** list, choose the bucket you created with the template.
3. Choose **Upload**.
4. Upload a test file to generate an `Object Created` event:
   a. Choose **Add files** and select a file from your computer.
   b. Choose **Upload**.
5. Wait a few moments for the event to be processed by EventBridge and for the notification to be sent.
6. Check your email for a notification about the object creation event. The email will contain details about the S3 event, including the bucket name and the object key.

Viewing rule metrics

You can view metrics for your rule to confirm that events are being processed correctly.

1. In the EventBridge console, choose your rule.
2. Choose the **Metrics** tab.
3. You can view metrics such as:
   - **Invocations**: the number of times the rule was triggered.
   - **TriggeredRules**: the number of rules that were triggered by matching events.

Clean up: deleting resources

As a final step, we'll delete the stack and the resources it contains.

> **Important**
>
> You will be billed for the Amazon resources contained in the stack for as long as it exists.

1. Open the CloudFormation console at https://console.aws.amazon.com/cloudformation/.
2. On the **Stacks** page, choose the stack created from the template, and choose **Delete**, then confirm **Delete**.

   CloudFormation initiates deletion of the stack and all resources it includes.

CloudFormation template details

This template creates resources and grants permissions in your account.

Resources

The CloudFormation template for this tutorial will create the following resources in your account:

> **Important**
>
> You will be billed for the Amazon resources used if you create a stack from this template.

- `AWS::S3::Bucket`: An Amazon S3 bucket that acts as the event source for the rule, with EventBridge notifications enabled.
- `AWS::SNS::Topic`: An Amazon SNS topic that acts as the target for the events matched by the rule.
- `AWS::SNS::Subscription`: An email subscription to the SNS topic.
- `AWS::IAM::Role`: IAM execution roles granting permissions to the EventBridge service and Lambda cleanup function.
- `AWS::Events::Rule`: The rule connecting the Amazon S3 bucket events to the Amazon SNS topic.

- AWS::Lambda::Function: A Lambda function that empties the Amazon S3 bucket when the stack is deleted, enabling clean deletion of all resources.

Permissions

The template includes an AWS::IAM::Role resource that represents an execution role. This role grants the EventBridge service (events.amazonaws.com) the following permissions in your account.

The following permissions are granted through the managed policy AmazonSNSFullAccess:

- Full access to Amazon SNS resources and operations

CloudFormation template

Save the following YAML code as a separate file to use as the CloudFormation template for this tutorial.

YAML

```
AWSTemplateFormatVersion: '2010-09-09'
Description: '[AWSDocs] EventBridge: event-bus-rule-get-started'

Parameters:
  BucketName:
    Type: String
    Description: Name of the S3 bucket (must be globally unique)

  SNSTopicDisplayName:
    Type: String
    Description: Display name for the SNS topic
    Default: eventbridge-rule-example-target

  SNSTopicName:
    Type: String
    Description: Name for the SNS topic
    Default: eventbridge-rule-example-target

  RuleName:
    Type: String
    Description: Name for the EventBridge rule
```

```
    Default: eventbridge-rule-example

  EmailAddress:
    Type: String
    Description: Email address to receive notifications
    AllowedPattern: '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+[a-zA-Z0-9-]*(\\.[a-zA-
Z0-9-]+)*$'

Resources:
  # S3 Bucket with notifications enabled
  S3Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Ref BucketName
      NotificationConfiguration:
        EventBridgeConfiguration:
          EventBridgeEnabled: true

  # Lambda function to empty the S3 bucket before deletion
  EmptyBucketFunction:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.12
      Handler: index.handler
      Timeout: 60
      Role: !GetAtt EmptyBucketRole.Arn
      Code:
        ZipFile: |
          import boto3
          import cfnresponse
          def handler(event, context):
              bucket = event['ResourceProperties']['BucketName']
              if event['RequestType'] == 'Delete':
                  s3 = boto3.resource('s3')
                  bucket_resource = s3.Bucket(bucket)
                  bucket_resource.objects.all().delete()
              cfnresponse.send(event, context, cfnresponse.SUCCESS, {})

  # IAM Role for the bucket cleanup Lambda function
  EmptyBucketRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
```

```
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      Policies:
        - PolicyName: EmptyBucketPolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - s3:DeleteObject
                  - s3:ListBucket
                Resource:
                  - !Sub arn:aws:s3:::${BucketName}
                  - !Sub arn:aws:s3:::${BucketName}/*

  # Custom resource to trigger bucket cleanup on stack deletion
  EmptyBucketOnDelete:
    Type: Custom::EmptyBucket
    Properties:
      ServiceToken: !GetAtt EmptyBucketFunction.Arn
      BucketName: !Ref S3Bucket

  # SNS Topic for email notifications
  SNSTopic:
    Type: AWS::SNS::Topic
    Properties:
      DisplayName: !Ref SNSTopicDisplayName
      TopicName: !Ref SNSTopicName

  # SNS Subscription for email
  SNSSubscription:
    Type: AWS::SNS::Subscription
    Properties:
      Protocol: email
      Endpoint: !Ref EmailAddress
      TopicArn: !Ref SNSTopic

  # EventBridge Rule to match S3 object creation events and send them to the SNS
topic
```

```
  EventBridgeRule:
    Type: AWS::Events::Rule
    Properties:
      Name: !Ref RuleName
      Description: "Rule to detect S3 object creation and send email notification"
      EventPattern:
        source:
          - aws.s3
        detail-type:
          - "Object Created"
        detail:
          bucket:
            name:
              - !Ref BucketName
      State: ENABLED
      Targets:
        - Id: SendToSNS
          Arn: !Ref SNSTopic
          RoleArn: !GetAtt EventBridgeRole.Arn

  # IAM Role for EventBridge to publish to SNS
  EventBridgeRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: events.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/AmazonSNSFullAccess

Outputs:
  BucketName:
    Description: Name of the S3 bucket
    Value: !Ref S3Bucket
  SNSTopicARN:
    Description: ARN of the SNS topic
    Value: !Ref SNSTopic
  EmailSubscription:
    Description: Email address for notifications
```

```
    Value: !Ref EmailAddress
```

JSON

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "[AWSDocs] EventBridge: event-bus-rule-get-started",
  "Parameters": {
    "BucketName": {
      "Type": "String",
      "Description": "Name of the S3 bucket (must be globally unique)"
    },
    "SNSTopicDisplayName": {
      "Type": "String",
      "Description": "Display name for the SNS topic",
      "Default": "eventbridge-rule-example-target"
    },
    "SNSTopicName": {
      "Type": "String",
      "Description": "Name for the SNS topic",
      "Default": "eventbridge-rule-example-target"
    },
    "RuleName": {
      "Type": "String",
      "Description": "Name for the EventBridge rule",
      "Default": "eventbridge-rule-example"
    },
    "EmailAddress": {
      "Type": "String",
      "Description": "Email address to receive notifications",
      "AllowedPattern": "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+[a-zA-Z0-9-]*(\\.[a-zA-
Z0-9-]+)*$"
    }
  },
  "Resources": {
    "S3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": {
          "Ref": "BucketName"
        },
        "NotificationConfiguration": {
          "EventBridgeConfiguration": {
```

```
            "EventBridgeEnabled": true
          }
        }
      }
    },
    "EmptyBucketFunction": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Runtime": "python3.12",
        "Handler": "index.handler",
        "Timeout": 60,
        "Role": {
          "Fn::GetAtt": ["EmptyBucketRole", "Arn"]
        },
        "Code": {
          "ZipFile": "import boto3\nimport cfnresponse\ndef
 handler(event, context):\n    bucket = event['ResourceProperties']
['BucketName']\n    if event['RequestType'] == 'Delete':\n        s3 =
 boto3.resource('s3')\n        bucket_resource = s3.Bucket(bucket)\n
 bucket_resource.objects.all().delete()\n    cfnresponse.send(event, context,
 cfnresponse.SUCCESS, {})"
        }
      }
    },
    "EmptyBucketRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
            {
              "Effect": "Allow",
              "Principal": {
                "Service": "lambda.amazonaws.com"
              },
              "Action": "sts:AssumeRole"
            }
          ]
        },
        "ManagedPolicyArns": [
          "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        ],
        "Policies": [
          {
```

```
            "PolicyName": "EmptyBucketPolicy",
            "PolicyDocument": {
              "Version": "2012-10-17",
              "Statement": [
                {
                  "Effect": "Allow",
                  "Action": ["s3:DeleteObject", "s3:ListBucket"],
                  "Resource": [
                    {"Fn::Sub": "arn:aws:s3:::${BucketName}"},
                    {"Fn::Sub": "arn:aws:s3:::${BucketName}/*"}
                  ]
                }
              ]
            }
          }
        ]
      }
    },
    "EmptyBucketOnDelete": {
      "Type": "Custom::EmptyBucket",
      "Properties": {
        "ServiceToken": {
          "Fn::GetAtt": ["EmptyBucketFunction", "Arn"]
        },
        "BucketName": {
          "Ref": "S3Bucket"
        }
      }
    },
    "SNSTopic": {
      "Type": "AWS::SNS::Topic",
      "Properties": {
        "DisplayName": {
          "Ref": "SNSTopicDisplayName"
        },
        "TopicName": {
          "Ref": "SNSTopicName"
        }
      }
    },
    "SNSSubscription": {
      "Type": "AWS::SNS::Subscription",
      "Properties": {
        "Protocol": "email",
```

```
        "Endpoint": {
          "Ref": "EmailAddress"
        },
        "TopicArn": {
          "Ref": "SNSTopic"
        }
      }
    },
    "EventBridgeRule": {
      "Type": "AWS::Events::Rule",
      "Properties": {
        "Name": {
          "Ref": "RuleName"
        },
        "Description": "Rule to detect S3 object creation and send email
notification",
        "EventPattern": {
          "source": [
            "aws.s3"
          ],
          "detail-type": [
            "Object Created"
          ],
          "detail": {
            "bucket": {
              "name": [
                {
                  "Ref": "BucketName"
                }
              ]
            }
          }
        },
        "State": "ENABLED",
        "Targets": [
          {
            "Id": "SendToSNS",
            "Arn": {
              "Ref": "SNSTopic"
            },
            "RoleArn": {
              "Fn::GetAtt": [
                "EventBridgeRole",
                "Arn"
```

```
            ]
          }
        }
      ]
    }
  },
  "EventBridgeRole": {
    "Type": "AWS::IAM::Role",
    "Properties": {
      "AssumeRolePolicyDocument": {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "events.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      },
      "ManagedPolicyArns": [
        "arn:aws:iam::aws:policy/AmazonSNSFullAccess"
      ]
    }
  }
},
"Outputs": {
  "BucketName": {
    "Description": "Name of the S3 bucket",
    "Value": {
      "Ref": "S3Bucket"
    }
  },
  "SNSTopicARN": {
    "Description": "ARN of the SNS topic",
    "Value": {
      "Ref": "SNSTopic"
    }
  },
  "EmailSubscription": {
    "Description": "Email address for notifications",
    "Value": {
      "Ref": "EmailAddress"
```

```
        }
      }
    }
}
```

Amazon EventBridge tutorials

EventBridge integrates with a number of AWS services and SaaS partners. These tutorials are designed to help you get familiar with the basics of EventBridge and how it can be part of your serverless architecture.

Getting started

The following tutorials help you explore the features of EventBridge and how to use them.

- Archive and replay events
- Create a sample application
- Download code bindings
- Use input transformer

AWS tutorials

Amazon EventBridge works with other AWS services to process events or invoke an AWS resource as the target of a rule. The following tutorials show you how to integrate EventBridge with other AWS services.

- Log Auto Scaling group states
- Create rule for AWS API calls via CloudTrail
- Log Amazon EC2 instance states
- Log Amazon S3 object operations
- Send events using schemas
- Schedule automated Amazon EBS snapshots
- Send an email when events happen
- Schedule a Lambda function

SaaS providers tutorials

EventBridge can work directly with SaaS partner applications and services to send and receive events. The following tutorials show you how to integrate EventBridge with SaaS partners.

- Send events to Datadog
- Send events to Salesforce
- Send events to Zendesk

Tutorial: Create a sample Amazon EventBridge application

You can use EventBridge to route events to specific Lambda functions using rules.

In this tutorial, you'll use the AWS CLI, Node.js, and the code in the GitHub repo to create the following:

- An AWS Lambda function that produces events for bank ATM transactions.
- Three Lambda functions to use as targets of an EventBridge rule.
- and the rule that routes the created events to the correct downstream function based on an event pattern.

This example uses AWS SAM templates to define the EventBridge rules. To learn more about using AWS SAM templates with EventBridge see ???.

In the repo, the *atmProducer* subdirectory contains `handler.js`, which represents the ATM service producing events. This code is a Lambda handler written in Node.js, and publishes events to EventBridge via the AWS SDK using this line of JavaScript code.

```
const result = await eventbridge.putEvents(params).promise()
```

This directory also contains `events.js`, listing several test transactions in an Entries array. A single event is defined in JavaScript as follows:

```
{
  // Event envelope fields
  Source: 'custom.myATMapp',
  EventBusName: 'default',
  DetailType: 'transaction',
  Time: new Date(),

  // Main event body
  Detail: JSON.stringify({
    action: 'withdrawal',
    location: 'MA-BOS-01',
    amount: 300,
    result: 'approved',
    transactionId: '123456',
    cardPresent: true,
    partnerBank: 'Example Bank',
```

```
    remainingFunds: 722.34
  })
}
```

The *Detail* section of the event specifies transaction attributes. These include the location of the ATM, the amount, the partner bank, and the result of the transaction.

The `handler.js` file in the *atmConsumer* subdirectory contains three functions:

```
exports.case1Handler = async (event) => {
  console.log('--- Approved transactions ---')
  console.log(JSON.stringify(event, null, 2))
}

exports.case2Handler = async (event) => {
  console.log('--- NY location transactions ---')
  console.log(JSON.stringify(event, null, 2))
}

exports.case3Handler = async (event) => {
  console.log('--- Unapproved transactions ---')
  console.log(JSON.stringify(event, null, 2))
}
```

Each function receives transaction events, which are logged via the `console.log` statements to Amazon CloudWatch Logs. The consumer functions operate independently of the producer and are unaware of the source of the events.

The routing logic is contained in the EventBridge rules that are deployed by the application's AWS SAM template. The rules evaluate the incoming stream of events, and route matching events to the target Lambda functions.

The rules use event patterns that are JSON objects with the same structure as the events they match. Here's the event pattern for the one of the rules.

```
{
  "detail-type": ["transaction"],
  "source": ["custom.myATMapp"],
  "detail": {
    "location": [{
      "prefix": "NY-"
    }]
```

```
  }
}
```

Steps:

- Prerequisites
- Step 1: Create application
- Step 2: Run application
- Step 3: Check the logs and verify the application works
- Step 4: Clean up your resources

Prerequisites

To complete this tutorial, you'll need the following resources:

- An AWS account. Create an AWS account if you don't already have one.
- AWS CLI installed. To install the AWS CLI, see the Installing, updating, and uninstalling the AWS CLI version 2.
- Node.js 12.x installed. To install Node.js, see Downloads.

Step 1: Create application

To set up the example application, you'll use the AWS CLI and Git to create the AWS resources you'll need.

To create the application

1. Sign in to AWS.
2. Install Git and install the AWS Serverless Application Model CLI on your local machine.
3. Create a new directory, and then navigate to that directory in a terminal.
4. At the command line, enter `git clone https://github.com/aws-samples/amazon-eventbridge-producer-consumer-example`.
5. At the command line run the following command:

```
cd ./amazon-eventbridge-producer-consumer-example
sam deploy --guided
```

6. In the terminal, do the following:

   a. For **`Stack Name`**, enter a name for the stack. For example, name the stack `Test`.

   b. For **`AWS Region`**, enter the Region. For example, `us-west-2`.

   c. For **`Confirm changes before deploy`**, enter Y.

   d. For **`Allow SAM CLI IAM role creation`**, enter Y

   e. For **`Save arguments to configuration file`**, enter Y

   f. For **`SAM configuration file`**, enter `samconfig.toml`.

   g. For **`SAM configuration environment`**, enter `default`.

Step 2: Run application

Now that you've set up the resources, you'll use the console to test the functions.

To run the application

1. Open the Lambda console in the same Region where you deployed the AWS SAM application.
2. There are four Lambda functions with the prefix **atm-demo**. Select the **atmProducerFn** function, then choose **Actions**, **Test**.
3. Enter `Test` for the **Name**.
4. Choose **Test**.

Step 3: Check the logs and verify the application works

Now that you've run the application, you'll use the console to check the CloudWatch Logs.

To check the logs

1. Open the CloudWatch console in the same Region where you ran the AWS SAM application.
2. Choose **Logs**, and then choose **Log groups**.
3. Select the log group containing **atmConsumerCase1**. You see two streams representing the two transactions approved by the ATM. Choose a log stream to view the output.
4. Navigate back to the list of log groups, and then select the log group containing **atmConsumerCase2**. You'll see two streams representing the two transactions matching the *New York* location filter.

5. Navigate back to the list of log groups, and select the log group containing **atmConsumerCase3**. Open the stream to see the denied transactions.

Step 4: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

To delete the CloudWatch Logs log group(s)

1. Open the Cloudwatch console.
2. Choose **Logs**, **Log groups**.
3. Select the log group(s) that were created in this tutorial.
4. Choose **Actions**, **Delete log group(s)**.
5. Choose **Delete**.

Tutorial: Archive and replay events in Amazon EventBridge

You can use EventBridge to route events to specific AWS Lambda functions using rules.

In this tutorial, you'll create a function to use as the target for the EventBridge rule using the Lambda console. Then, you'll create an archive and a rule that'll archive test events using the EventBridge console. Once there are events in that archive, you'll replay them.

Steps:

- Step 1: Create a Lambda function
- Step 2: Create archive
- Step 3: Create rule
- Step 4: Send test events
- Step 5: Replay events
- Step 6: Clean up your resources

Step 1: Create a Lambda function

First, create a Lambda function to log the events.

To create a Lambda function:

1. Open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Enter a name and description for the Lambda function. For example, name the function `LogScheduledEvent`.
5. Leave the rest of the options as the defaults and choose **Create function**.
6. On the **Code** tab of the function page, double-click **index.js**.
7. Replace the existing JavaScript code with the following code:

```
'use strict';

exports.handler = (event, context, callback) => {
    console.log('LogScheduledEvent');
```

```
    console.log('Received event:', JSON.stringify(event, null, 2));
    callback(null, 'Finished');
};
```

8. Choose **Deploy**.

Step 2: Create archive

Next, create the archive that will hold all the test events.

To create an archive

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Archives**.
3. Choose **Create archive**.
4. Enter a name and description for the archive. For example, name the archive `ArchiveTest`.
5. Leave the rest of the options as the defaults and choose **Next**.
6. Choose **Create archive**.

Step 3: Create rule

Create a rule to archive events that are sent to the event bus.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `ARTestRule`.

   A rule can't have the same name as another rule in the same Region and on the same event bus.
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.

6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **Other**.
9. For **Event pattern**, enter the following:

```
{
  "detail-type": [
    "customerCreated"
  ]
}
```

10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **Lambda function** from the drop-down list.
13. For **Function**, select the Lambda function that you created in the **Step 1: Create a Lambda function** section. In this example, select `LogScheduledEvent`.
14. Choose **Next**.
15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Step 4: Send test events

Now that you've set up the archive and the rule, we'll send test events to make sure the archive is working correctly.

> **Note**
>
> It can take some time for events to get to the archive.

To send test events (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. In the **Default event bus** tile, choose **Actions**, **Send events**.

4. Enter an event source. For example, `TestEvent`.
5. For **Detail type**, enter `customerCreated`.
6. For **Event detail**, enter `{}`.
7. Choose **Send**.

Step 5: Replay events

Once the test events are in the archive you can replay them.

To replay archived events (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Replays**.
3. Choose **Start new replay**.
4. Enter a name and description for the replay. For example, name the replay `ReplayTest`.
5. For **Source**, select the archive you created in the **Step 2: Create archive** section.
6. For **Replay time frame**, do the following.
   a. For **Start time**, select the date you sent test events and a time before you sent them. For example, `2021/08/11` and `08:00:00`.
   b. For **End time**, select the current date and time. For example, `2021/08/11` and `09:15:00`.
7. Choose **Start Replay**.

Step 6: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.

4. Choose **Delete**.

To delete the EventBridge archives(s)

1. Open the Archives page of the EventBridge console.
2. Select the archive(s) you created.
3. Choose **Delete**.
4. Enter the archive name and choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Download code bindings for event schemas in EventBridge

You can generate code bindings for event schemas to speed development for Golang, Java, Python, and TypeScript. You can get code bindings for existing AWS services, schemas you create, and for schemas you generate based on events on an event bus. You can generate code bindings for a schema using one of the following:

- EventBridge console
- EventBridge schema registry API
- Your IDE with an AWS toolkit

In this tutorial you generate and download code bindings from an EventBridge schema for the events of an AWS service.

To generate code bindings from an EventBridge schema

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas**.
3. Select the **AWS event schema registry** tab.
4. Find the schema for the AWS service that you would like code bindings for, either by browsing through the schema registry, or by searching for a schema.
5. Select the schema name.
6. On the **Schema details** page, in the **Version** section, select **Download code bindings**.
7. On the **Download code bindings** page, select the language of the code bindings you want to download.
8. Select **Download**.

   It may take a few seconds for your download to begin. The download file will be a .zip file of code bindings for the language you selected.
9. Unzip the downloaded file and add it to your project.

   The downloaded package contains a README file that explains how to configure the package's dependencies in various frameworks.

Use these code bindings in your own code to help quickly build applications using this EventBridge event.

Tutorial: Use input transformers to transform events in EventBridge

You can use the Input transformer in EventBridge to customize text from an event before you send it to the target of a rule.

To do this, you define JSON paths from the event and assign their outputs to different variables. Then you can use those variables in the input template. The characters < and > can't be escaped. For more information, see Amazon EventBridge input transformation

> **Note**
>
> If you specify a variable to match a JSON path that doesn't exist in the event, that variable isn't created and doesn't appear in the output.

In this tutorial, you create a rule that matches an event with `detail-type: "customerCreated"`. The input transformer maps the `type` variable to the $.detail-type JSON path from the event. Then EventBridge puts the variable into the input template "This event was <type>." The result is the following Amazon SNS message.

```
"This event was of customerCreated type."
```

Steps:

- Step 1: Create an Amazon SNS topic
- Step 2: Create an Amazon SNS subscription
- Step 3: Create a rule
- Step 4: Send test events
- Step 5: Confirm success
- Step 6: Clean up your resources

Step 1: Create an Amazon SNS topic

Create a topic to receive the events from EventBridge.

To create a topic

1. Open the Amazon SNS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sns/v3/home.
2. In the navigation pane, choose **Topics**.
3. Choose **Create topic**.
4. For **Type**, choose **Standard**.
5. Enter **`eventbridge-IT-test`** as the name of the topic.
6. Choose **Create topic**.

Step 2: Create an Amazon SNS subscription

Create a subscription to get emails with the transformed information.

To create a subscription

1. Open the Amazon SNS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sns/v3/home.
2. In the navigation pane, choose **Subscriptions**.
3. Choose **Create subscription**.
4. For **Topic ARN**, choose the topic you created in step 1. For this tutorial, choose **eventbridge-IT-test**.
5. For **Protocol**, choose **Email**.
6. For **Endpoint**, enter your email address.
7. Choose **Create subscription**.
8. Confirm the subscription by choosing **Confirm subscription** in the email you receive from AWS notifications.

Step 3: Create a rule

Create a rule to use the input transformer to customize the instance state information that goes to a target.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `ARTestRule`
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **Other**.
9. For **Event pattern**, enter the following:

```
{
  "detail-type": [
    "customerCreated"
  ]
}
```

10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **SNS topic** from the drop-down list.
13. For **Topic**, select the Amazon SNS topic that you created in step 1. For this tutorial, choose **eventbridge-IT-test**.
14. For **Additional settings**, do the following:

    a. For **Configure target input**, choose **Input transformer** from the drop-down list.

    b. Choose **Configure input transformer**

    c. for **Sample events**, enter the following:

```
{
  "detail-type": "customerCreated"
}
```

d. For **Target input transformer** do the following:

i. For **Input Path**, enter the following:

```
{"detail-type":"$.detail-type"}
```

ii. For **Input Template**, enter the following:

```
"This event was of <detail-type> type."
```

e. Choose **Confirm.**.

15. Choose **Next**.
16. Choose **Next**.
17. Review the details of the rule and choose **Create rule**.

Step 4: Send test events

Now that you've set up the SNS topic and the rule, we'll send test events to make sure the rule is working correctly.

To send test events (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. In the **Default event bus** tile, choose **Actions**, **Send events**.
4. Enter an event source. For example, `TestEvent`.
5. For **Detail type**, enter `customerCreated`.
6. For **Event detail**, enter `{}`.
7. Choose **Send**.

Step 5: Confirm success

If you get an email from AWS notifications that matches the expected output, you've successfully completed the tutorial.

Step 6: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the SNS topic

1. Open the Topics page of the SNS console.
2. Select the topic that you created.
3. Choose **Delete**.
4. Enter **`delete me`**.
5. Choose **Delete**.

To delete the SNS subscription

1. Open the Subscriptions page of the SNS console.
2. Select the subscription that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Log the state of an Auto Scaling group using EventBridge

You can run an AWS Lambda function that logs an events whenever an Auto Scaling group launches or terminates an Amazon EC2 instance that indicates whether an event was successful.

For information about more scenarios that use Amazon EC2 Auto Scaling events, see Use EventBridge to handle Auto Scaling events in the *Amazon EC2 Auto Scaling User Guide*.

In this tutorial, you create a Lambda function, and you create a rule in the EventBridge console that calls that function when an Amazon EC2 Auto Scaling group launches or terminates an instance.

Steps:

- Prerequisites
- Step 1: Create a Lambda function
- Step 2: Create a rule
- Step 3: Test the rule
- Step 4: Confirm success
- Step 5: Clean up your resources

Prerequisites

To complete this tutorial, you'll need the following resources:

- An Auto Scaling group. For more information about creating one, see Creating an Auto Scaling group using a launch configuration in the Amazon EC2 Auto Scaling User Guide.

Step 1: Create a Lambda function

Create a Lambda function to log the scale-out and scale-in events for your Auto Scaling group.

To create a Lambda function

1. Open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Choose **Create function**.
3. Choose **Author from scratch**.

4. Enter a name for the Lambda function. For example, name the function `LogAutoScalingEvent`.
5. Leave the rest of the options as the defaults and choose **Create function**.
6. On the **Code** tab of the function page, double-click **index.js**.
7. Replace the existing code with the following code.

```
'use strict';

exports.handler = (event, context, callback) => {
    console.log('LogAutoScalingEvent');
    console.log('Received event:', JSON.stringify(event, null, 2));
    callback(null, 'Finished');
};
```

8. Choose **Deploy**.

Step 2: Create a rule

Create a rule to run the Lambda function you created in Step 1. The rule runs when your Auto Scaling group starts or stops an instance.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `TestRule`
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS services**.
9. For **Event pattern**, do the following:

    a. For **Event source**, select **Auto Scaling** from the drop-down list.

b. For **Event type**, select **Instance Launch and Terminate** from the drop-down list.

c. Choose **Any instance event** and **Any group name**.

10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **Lambda function** from the drop-down list.
13. For **Function**, select the Lambda function that you created in the **Step 1: Create a Lambda function** section. In this example, select `LogAutoScalingEvent`.
14. Choose **Next**.
15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Step 3: Test the rule

You can test your rule by manually scaling an Auto Scaling group so that it launches an instance. Wait a few minutes for the scale-out event to occur, and then verify that your Lambda function was invoked.

To test your rule using an Auto Scaling group

1. To increase the size of your Auto Scaling group, do the following:

   a. Open the Amazon EC2 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/ec2/.

   b. In the navigation pane, choose **Auto Scaling**, **Auto Scaling Groups**.

   c. Select the check box for your Auto Scaling group.

   d. On the **Details** tab, choose **Edit**. For **Desired**, increase the desired capacity by one. For example, if the current value is **2**, enter **3**. The desired capacity must be less than or equal to the maximum size of the group. If your new value for **Desired** is greater than **Max**, you must update **Max**. When you're finished, choose **Save**.

2. To view the output from your Lambda function, do the following:

   a. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.

   b. In the navigation pane, choose **Logs**.

c. Select the name of the log group for your Lambda function (`/aws/lambda/function-name`).

d. Select the name of the log stream to view the data provided by the function for the instance that you launched.

3. (Optional) When you're finished, you can decrease the desired capacity by one so that the Auto Scaling group returns to its previous size.

Step 4: Confirm success

If you see the Lambda event in the CloudWatch logs, you've successfully completed this tutorial. If the event isn't in your CloudWatch logs, start troubleshooting by verifying the rule was created successfully and, if the rule looks correct, verify the code of your Lambda function is correct.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

Tutorial: Create an EventBridge rule that reacts to AWS API calls via CloudTrail

You can use Amazon EventBridge rules to react to API calls made by an AWS service that are recorded by AWS CloudTrail.

In this tutorial, you create an AWS CloudTrail trail, a Lambda function, and a rule in the EventBridge console. The rule invokes the Lambda function when an Amazon EC2 instance is stopped.

Steps:

- Step 1: Create an AWS CloudTrail trail
- Step 2: Create an AWS Lambda function
- Step 3: Create a rule
- Step 4: Test the rule
- Step 5: Confirm success
- Step 6: Clean up your resources

Step 1: Create an AWS CloudTrail trail

If you already have a trail set up, skip to step 2.

To create a trail

1. Open the CloudTrail console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudtrail/.
2. Choose **Trails**, **Create trail**.
3. For **Trail name**, type a name for the trail.
4. For **Storage location**, in **Create a new S3 bucket**.
5. For **AWS KMS alias**, type an alias for the KMS key.
6. Choose **Next**.
7. Choose **Next**.
8. Choose **Create trail**.

Step 2: Create an AWS Lambda function

Create a Lambda function to log the API call events.

To create a Lambda function

1. Open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Enter a name and description for the Lambda function. For example, name the function `LogEC2StopInstance`.
5. Leave the rest of the options as the defaults and choose **Create function**.
6. On the **Code** tab of the function page, double-click **index.js**.
7. Replace the existing code with the following code.

```
'use strict';

exports.handler = (event, context, callback) => {
    console.log('LogEC2StopInstance');
    console.log('Received event:', JSON.stringify(event, null, 2));
    callback(null, 'Finished');
};
```

8. Choose **Deploy**.

Step 3: Create a rule

Create a rule to run the Lambda function you created in step 2 whenever you stop an Amazon EC2 instance.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `TestRule`

5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS services**.
9. For **Event pattern**, do the following:
    a. For **Event source**, select **EC2** from the drop-down list.
    b. For **Event type**, select **AWS API Call via CloudTrail** from the drop-down list.
    c. Choose **Specific operation(s)** and enter `StopInstances`.
10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **Lambda function** from the drop-down list.
13. For **Function**, select the Lambda function that you created in the **Step 1: Create a Lambda function** section. In this example, select `LogEC2StopInstance`.
14. Choose **Next**.
15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Step 4: Test the rule

You can test your rule by stopping an Amazon EC2 instance using the Amazon EC2 console. Wait a few minutes for the instance to stop, and then check your AWS Lambda metrics on the CloudWatch console to verify that your function ran.

To test your rule by stopping an instance

1. Open the Amazon EC2 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/ec2/.
2. Launch an instance. For more information, see Launch Your Instance in the *Amazon EC2 User Guide*.
3. Stop the instance. For more information, see Stop and Start Your Instance in the *Amazon EC2 User Guide*.
4. To view the output from your Lambda function, do the following:

a. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
b. In the navigation pane, choose **Logs**.
c. Select the name of the log group for your Lambda function (`/aws/lambda/function-name`).
d. Select the name of the log stream to view the data provided by the function for the instance that you stopped.

5. (Optional) When you're finished, terminate the stopped instance. For more information, see Terminate Your Instance in the *Amazon EC2 User Guide*.

Step 5: Confirm success

If you see the Lambda event in the CloudWatch logs, you've successfully completed this tutorial. If the event isn't in your CloudWatch logs, start troubleshooting by verifying the rule was created successfully and, if the rule looks correct, verify the code of your Lambda function is correct.

Step 6: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

To delete the CloudTrail trail(s)

1. Open the Trails page of the CloudTrail console.
2. Select the trail(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Log the state of an Amazon EC2 instance using EventBridge

You can create an AWS Lambda function that logs a state change for an Amazon EC2 instance. Then you can create a rule that runs your Lambda function whenever there is a state transition or a transition to one or more states that are of interest. In this tutorial, you log the launch of any new instance.

Steps:

- Step 1: Create an AWS Lambda function
- Step 2: Create a rule
- Step 3: Test the rule
- Step 4: Confirm success
- Step 5: Clean up your resources

Step 1: Create an AWS Lambda function

Create a Lambda function to log the state change events. When you create your rule in Step 2, you specify this function.

To create a Lambda function

1. Open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Enter a name and description for the Lambda function. For example, name the function `LogEC2InstanceStateChange`.
5. Leave the rest of the options as the defaults and choose **Create function**.
6. On the **Code** tab of the function page, double-click **index.js**.
7. Replace the existing code with the following code.

```
'use strict';

exports.handler = (event, context, callback) => {
    console.log('LogEC2InstanceStateChange');
    console.log('Received event:', JSON.stringify(event, null, 2));
```

```
    callback(null, 'Finished');
};
```

8. Choose **Deploy**.

Step 2: Create a rule

Create a rule to run the Lambda function you created in Step 1. The rule runs when you launch an Amazon EC2 instance.

To create the EventBridge rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `TestRule`
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS services**.
9. For **Event pattern**, do the following:
    a. For **Event source**, select **EC2** from the drop-down list.
    b. For **Event type**, choose **EC2 Instance State-change Notification** from the drop-down list.
    c. Choose **Specific states(s)** and choose **running** from the drop-down list.
    d. Choose **Any instance**
10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **Lambda function** from the drop-down list.
13. For **Function**, select the Lambda function that you created in the **Step 1: Create a Lambda function** section. In this example, select `LogEC2InstanceStateChange`.
14. Choose **Next**.

15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Step 3: Test the rule

You can test your rule by stopping an Amazon EC2 instance using the Amazon EC2 console. Wait a few minutes for the instance to stop, and then check your AWS Lambda metrics on the CloudWatch console to verify that your function ran.

To test your rule by stopping an instance

1. Open the Amazon EC2 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/ec2/.
2. Launch an instance. For more information, see Launch Your Instance in the *Amazon EC2 User Guide*.
3. Stop the instance. For more information, see Stop and Start Your Instance in the *Amazon EC2 User Guide*.
4. To view the output from your Lambda function, do the following:
    a. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
    b. In the navigation pane, choose **Logs**.
    c. Select the name of the log group for your Lambda function (`/aws/lambda/function-name`).
    d. Select the name of the log stream to view the data provided by the function for the instance that you stopped.
5. (Optional) When you're finished, terminate the stopped instance. For more information, see Terminate Your Instance in the *Amazon EC2 User Guide*.

Step 4: Confirm success

If you see the Lambda event in the CloudWatch logs, you've successfully completed this tutorial. If the event isn't in your CloudWatch logs, start troubleshooting by verifying the rule was created successfully and, if the rule looks correct, verify the code of your Lambda function is correct.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

Tutorial: Log Amazon S3 object-level operations using EventBridge

You can log the object-level API operations on your Amazon S3 buckets. Before Amazon EventBridge can match these events, you must use AWS CloudTrail to set up and configure a trail to receive these events.

In this tutorial, you create CloudTrail trail, create a AWS Lambda function, and then create rule in the EventBridge console that invokes that function in response to an S3 data event.

Steps:

- Step 1: Configure your AWS CloudTrail trail
- Step 2: Create an AWS Lambda function
- Step 3: Create a Rule
- Step 4: Test the Rule
- Step 5: Confirm success
- Step 6: Clean up your resources

Step 1: Configure your AWS CloudTrail trail

To log data events for an S3 bucket to AWS CloudTrail and EventBridge, you first create a trail. A *trail* captures API calls and related events in your account and then delivers the log files to an S3 bucket that you specify. You can update an existing trail or create one.

For more information, see Data Events in the *AWS CloudTrail User Guide*.

To create a trail

1. Open the CloudTrail console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudtrail/.
2. Choose **Trails**, **Create trail**.
3. For **Trail name**, type a name for the trail.
4. For **Storage location**, in **Create a new S3 bucket**.
5. For **AWS KMS alias**, type an alias for the KMS key.
6. Choose **Next**.
7. For **Event type**, choose **Data events**

8. For **Data events,** do one of the following:

   - To log data events for all Amazon S3 objects in a bucket, specify an S3 bucket and an empty prefix. When an event occurs on an object in that bucket, the trail processes and logs the event.
   - To log data events for specific Amazon S3 objects in a bucket, specify an S3 bucket and the object prefix. When an event occurs on an object in that bucket and the object starts with the specified prefix, the trail processes and logs the event.
9. For each resource, choose whether to log **Read** events, **Write** events, or both.
10. Choose **Next**.
11. Choose **Create trail**.

Step 2: Create an AWS Lambda function

Create a Lambda function to log data events for your S3 buckets.

To create a Lambda function

1. Open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Enter a name and description for the Lambda function. For example, name the function `LogS3DataEvents`.
5. Leave the rest of the options as the defaults and choose **Create function**.
6. On the **Code** tab of the function page, double-click **index.js**.
7. Replace the existing code with the following code.

```
'use strict';

exports.handler = (event, context, callback) => {
    console.log('LogS3DataEvents');
    console.log('Received event:', JSON.stringify(event, null, 2));
    callback(null, 'Finished');
};
```

8. Choose **Deploy**.

Step 3: Create a Rule

Create a rule to run the Lambda function you created in Step 2. This rule runs in response to an Amazon S3 data event.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `TestRule`
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **default**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS services**.
9. For **Event pattern**, do the following:
    a. For **Event source**, select **Simple Storage Service (S3)** from the drop-down list.
    b. For **Event type**, select **Object-Level API call via CloudTrail** from the drop-down list.
    c. Choose **Specific operation(s)**, and then choose **PutObject**.
    d. By default, the rule matches data events for all buckets in the Region. To match data events for specific buckets, choose **Specify bucket(s) by name** and enter one or more buckets.
10. Choose **Next**.
11. For **Target types**, choose **AWS service**.
12. For **Select a target**, choose **Lambda function** from the drop-down list.
13. For **Function**, select the `LogS3DataEvents` Lambda function that you created in step 1.
14. Choose **Next**.
15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Step 4: Test the Rule

To test the rule, put an object in your S3 bucket. You can verify that your Lambda function was invoked.

To view the logs for your Lambda function

1. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
2. In the navigation pane, choose **Logs**.
3. Select the name of the log group for your Lambda function (`/aws/lambda/`*`function-name`*).
4. Select the name of the log stream to view the data provided by the function for the instance that you launched.

You can also check your CloudTrail logs in the S3 bucket that you specified for your trail. For more information, see Getting and Viewing Your CloudTrail Log Files in the *AWS CloudTrail User Guide*.

Step 5: Confirm success

If you see the Lambda event in the CloudWatch logs, you've successfully completed this tutorial. If the event isn't in your CloudWatch logs, start troubleshooting by verifying the rule was created successfully and, if the rule looks correct, verify the code of your Lambda function is correct.

Step 6: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Lambda function(s)

1. Open the Functions page of the Lambda console.
2. Select the function(s) that you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

To delete the CloudTrail trail(s)

1. Open the Trails page of the CloudTrail console.
2. Select the trail(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Send events to Amazon Kinesis using EventBridge schemas

You can send AWS API call events in EventBridge to an Amazon Kinesis stream, create Kinesis Data Streams applications, and process large amounts of data. In this tutorial, you create a Kinesis stream, and then create a rule in the EventBridge console that sends events to that stream when an Amazon EC2 instance stops.

Steps:

- Prerequisites
- Step 1: Create an Amazon Kinesis stream
- Step 2: Create a rule
- Step 3: Test the rule
- Step 4: Verify that the event was sent
- Step 5: Clean up your resources

Prerequisites

In this tutorial, you'll use the following:

- Use the AWS CLI to work with Kinesis streams.

  To install the AWS CLI, see the Installing, updating, and uninstalling the AWS CLI version 2.

> **Note**
>
> This tutorial uses AWS events and the built in `aws.events` schema registry. You can also create an EventBridge rule based on the schema of your custom events by adding them to a custom schema registry manually, or by using schema discovery.
> For more information on schemas, see ???. For more information on creating a rule using other event pattern options, see ???.

Step 1: Create an Amazon Kinesis stream

To create a stream, at a command prompt, use the `create-stream` AWS CLI command.

```
aws kinesis create-stream --stream-name test --shard-count 1
```

When the stream status is ACTIVE, the stream is ready. To check the stream status, use the `describe-stream` command.

```
aws kinesis describe-stream --stream-name test
```

Step 2: Create a rule

Create a rule to send events to your stream when you stop an Amazon EC2 instance.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `TestRule`
5. For **Event bus**, select **default**.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS events or EventBridge partner events**.
9. For **Creation method**, choose **Use schema**.
10. For **Event pattern**, do the following:

    a. For **Schema type**, choose **Select schema from Schema registry**.

    b. For **Schema registry**, choose **aws.events** from the drop-down list.

    c. For **Schema**, choose **aws.ec2@EC2InstanceStateChangeNotification** from the drop-down list.

    EventBridge displays the event schema under **Models**.

    EventBridge displays a red asterisk next to any properties that are required *for the event*, not for the event pattern.

    d. In **Models**, set the following event filter properties:

i. Select **+ Edit** next to the **state** property.

   Leave **Relationship** empty. For **Value**, enter `running`. Choose **Set**.

ii. Select **+ Edit** next to the **source** property.

   Leave **Relationship** empty. For **Value**, enter `aws.ec2`. Choose **Set**.

iii. Select **+ Edit** next to the **detail-type** property.

   Leave **Relationship** empty. For **Value**, enter `EC2 Instance State-change Notification`. Choose **Set**.

e. To view the event pattern you've constructed, choose **Generate event pattern in JSON**

   EventBridge displays the event pattern in JSON:

```
{
  "detail": {
    "state": ["running"]
  },
  "detail-type": ["EC2 Instance State-change Notification"],
  "source": ["aws.ec2"]
}
```

11. Choose **Next**.
12. For **Target types**, choose **AWS service**.
13. For **Select a target**, choose **Kinesis stream** from the drop-down list.
14. For **Stream**, select the Kinesis stream that you created in the **Step 1: Create an Amazon Kinesis stream** section. In this example, select `test`.
15. For **Execution role**, choose **Create a new for role for this specific resource**.
16. Choose **Next**.
17. Choose **Next**.
18. Review the details of the rule and choose **Create rule**.

Step 3: Test the rule

To test your rule, stop an Amazon EC2 instance. Wait a few minutes for the instance to stop, and then check your CloudWatch metrics to verify that your function ran.

To test your rule by stopping an instance

1. Open the Amazon EC2 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/ec2/.
2. Launch an instance. For more information, see Launch Your Instance in the *Amazon EC2 User Guide*.
3. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
4. In the navigation pane, choose **Rules**.

   Choose the name of the rule that you created and choose **Metrics for the rule**.
5. (Optional) When you're finished, terminate the instance. For more information, see Terminate Your Instance in the *Amazon EC2 User Guide*.

Step 4: Verify that the event was sent

You can use the AWS CLI to get the record from the stream to verify that the event was sent.

To get the record

1. To start reading from your Kinesis stream, at a command prompt, use the `get-shard-iterator` command.

   ```
   aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-iterator-type TRIM_HORIZON --stream-name test
   ```

   The following is example output.

   ```
   {
       "ShardIterator": "AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="
   }
   ```
2. To get the record, use the following `get-records` command. Use the shard iterator from the output in the previous step.

   ```
   aws kinesis get-records --shard-iterator AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
   ```

```
+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd
+efGN2aHFdkH1rJl4BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

If the command is successful, it requests records from your stream for the specified shard. You can receive zero or more records. Any records returned might not represent all records in your stream. If you don't receive the data that you expect, keep calling `get-records`.

3. Records in Kinesis are encoded in Base64. Use a Base64 decoder to decode the data so that you can verify that it's the event that was sent to the stream in JSON form.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the Kinesis stream(s)

1. Open the Data streams page of the Kinesis console.
2. Select the stream(s) that you created.
3. Choose **Actions**, **Delete**.
4. Enter **delete** in the fiekd and choose **Delete**.

Tutorial: Schedule automated Amazon EBS snapshots

> **Note**
>
> Scheduled rules are a legacy feature of EventBridge.
> EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.
> Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
> For more information, see ??? or the *EventBridge Scheduler User Guide*.

This tutorial previously demonstrated how to create automated Amazon EBS snapshots using EventBridge scheduled rules. We now recommend using one of the following alternatives, depending on your use case.

Recommended: Amazon Data Lifecycle Manager

For automated Amazon EBS snapshot lifecycle management, including creation, retention, and deletion on a schedule, we recommend Amazon Data Lifecycle Manager (Amazon Data Lifecycle Manager). Amazon Data Lifecycle Manager is purpose-built for Amazon EBS snapshot automation and provides policy-based lifecycle management without requiring you to build and maintain scheduling infrastructure.

Alternative: Amazon EventBridge Scheduler

If you need more flexible scheduling options or want to combine snapshot creation with other AWS API actions, you can use Amazon EventBridge Scheduler. EventBridge Scheduler offers improved scalability over EventBridge scheduled rules, with a wider set of target API operations, flexible time windows, and built-in retry support.

To get started with EventBridge Scheduler, see Getting started with Amazon EventBridge Scheduler in the *Amazon EventBridge Scheduler User Guide*.

Legacy: EventBridge scheduled rules

If you still need to use EventBridge scheduled rules, see Creating a scheduled rule (legacy) in Amazon EventBridge.

Tutorial: Send an email when events happen using Amazon EventBridge

You can send email notifications when Amazon Simple Storage Service (Amazon S3) objects are created using Amazon EventBridge and Amazon SNS. In this tutorial, you will create an SNS topic and subscription. Then, you will create a rule in the EventBridge console that sends events to that topic when Amazon S3 `Object Created` events are received.

Steps:

- Prerequisites
- Step 1: Create an Amazon SNS topic
- Step 2: Create an Amazon SNS subscription
- Step 3: Create a rule
- Step 4: Test the rule
- Step 5: Clean up your resources

Prerequisites

To recieve Amazon S3 events in EventBridge, you must enable EventBridge in the Amazon S3 console. This tutorial assumes EventBridge is enabled. For more information, see Enabling Amazon EventBridge in the S3 console.

Step 1: Create an Amazon SNS topic

Create a topic to receive the events from EventBridge.

To create a topic

1. Open the Amazon SNS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sns/v3/home.
2. In the navigation pane, choose **Topics**.
3. Choose **Create topic**.
4. For **Type**, choose **Standard**.
5. Enter **`eventbridge-test`** as the name of the topic.
6. Choose **Create topic**.

Step 2: Create an Amazon SNS subscription

Create a subscription to get email notifications from Amazon S3 when events are received by the topic.

To create a subscription

1. Open the Amazon SNS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sns/v3/home.
2. In the navigation pane, choose **Subscriptions**.
3. Choose **Create subscription**.
4. For **Topic ARN**, choose the topic you created in step 1. For this tutorial, choose **eventbridge-test**.
5. For **Protocol**, choose **Email**.
6. For **Endpoint**, enter your email address.
7. Choose **Create subscription**.
8. Confirm the subscription by choosing **Confirm subscription** in the email you receive from AWS notifications.

Step 3: Create a rule

Create a rule to send events to your topic when an Amazon S3 object is created.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, name the rule `s3-test`
5. For **Event bus**, select **default**.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS events or EventBridge partner events**.
9. For **Creation method**, choose **Use pattern form**.

10. For **Event pattern**, do the following:
    a. For **Event source**, select **AWS services** from the drop-down list.
    b. For **AWS service**, select **Simple Storage Service (S3)** from the drop-down list.
    c. For **Event type**, choose **Amazon S3 Event Notification** from the drop-down list.
    d. Choose **Specific events(s)** and choose **Object Created** from the drop-down list.
    e. Choose **Any bucket**
11. Choose **Next**.
12. For **Target types**, choose **AWS service**.
13. For **Select a target**, choose **SNS topic** from the drop-down list.
14. For **Topic**, select the Amazon SNS topic that you created in the **Step 1: Create an SNS topic** section. In this example, select `eventbridge-test`.
15. Choose **Next**.
16. Choose **Next**.
17. Review the details of the rule and choose **Create rule**.

Step 4: Test the rule

To test your rule, create an Amazon S3 object by uploading a file to an EventBridge-enabled bucket. Then, wait a few minutes and verify if you receive an email from AWS notifications.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the SNS topic

1. Open the Topics page of the SNS console.
2. Select the topic that you created.
3. Choose **Delete**.
4. Enter **`delete me`**.
5. Choose **Delete**.

To delete the SNS subscription

1. Open the Subscriptions page of the SNS console.
2. Select the subscription that you created.
3. Choose **Delete**.
4. Choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Schedule a AWS Lambda function

> **Note**
>
> Scheduled rules are a legacy feature of EventBridge.
> EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.
> Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
> For more information, see ??? or the *EventBridge Scheduler User Guide*.

This tutorial previously demonstrated how to invoke a Lambda function on a schedule using EventBridge scheduled rules. We now recommend using Amazon EventBridge Scheduler instead. EventBridge Scheduler offers improved scalability, a wider set of target API operations, flexible time windows, and built-in retry and dead-letter queue support.

For a complete walkthrough of scheduling a Lambda function using EventBridge Scheduler, see Invoke a Lambda function on a schedule in the *AWS Lambda Developer Guide*.

For more information about EventBridge Scheduler, including how to create schedules using the console, AWS CLI, or SDKs, see the Amazon EventBridge Scheduler User Guide.

If you still need to use EventBridge scheduled rules, see Creating a scheduled rule (legacy) in Amazon EventBridge.

Tutorial: Send events to Datadog from Amazon EventBridge

You can use EventBridge to route events to third-party services,such as Datadog.

In this tutorial, you'll use the EventBridge console to create a connection to Datadog, an API destination that points to Datadog, and a rule to route events to Datadog.

Steps:

- Prerequisites
- Step 1: Create connection
- Step 2: Create API destination
- Step 3: Create rule
- Step 4: Test the rule
- Step 5: Clean up your resources

Prerequisites

To complete this tutorial, you'll need the following resources:

- A Datadog account.
- A Datadog API key.
- An EventBridge-enabled Amazon Simple Storage Service (Amazon S3) bucket.

Step 1: Create connection

To send events to Datadog, you'll first have to establish a connection to the Datadog API.

To create the connection

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose the **Connections** tab, and then choose **Create connection**.
4. Enter a name and description for the connection. For example, enter **`Datadog`** as a name, and **`Datadog API Connection`** as a description.

5. For **Authorization type**, choose **API key**.
6. For **API key name**, enter **`DD-API-KEY`**.
7. For **Value**, paste your Datadog secret API key.
8. Choose **Create**.

Step 2: Create API destination

Now that you've created the connection, next you'll create the API destination to use as the target of the rule.

To create the API Destination

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose **Create API destination**.
4. Enter a name and description for the API destination. For example, enter **`DatadogAD`** for the name, and **`Datadog API Destination`** for the description..
5. For **API destination endpoint**, enter the Datadog Logs endpoint: **`https://http-intake.logs.datadoghq.com/api/v2/logs`**.

   > **Note**
   >
   > This tutorial delivers events to Datadog Logs. You can also deliver events to Datadog using the events endpoint: `https://api.datadoghq.com/api/v1/events`.

6. For **HTTP method**, choose **POST**.
7. For **Invocation rate limit**, enter **`300`**.
8. For **Connection**, choose **Use an existing connection** and choose the `Datadog` connection you created in step 1.
9. Choose **Create**.

Step 3: Create rule

Next, you'll create a rule to send events to Datadog when an Amazon S3 object is created.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, enter **`DatadogRule`** for the name, and **`Rule to send events to Datadog for S3 object creation`** for the description.
5. For **Event bus**, choose **default**.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **Other**.
9. For **Event pattern**, enter the following:

```
{
  "source": ["aws.s3"]
}
```

10. Choose **Next**.
11. For **Target types**, choose **EventBridge API destination**.
12. For **API destination**, choose **Use an existing API destination**, and then choose the `DatadogAD` destination you created in step 2.
13. For **Execution role**, choose **Create a new for role for this specific resource**.
14. For **Additional settings**, do the following:

    a. For **Configure target input**, choose **Input transformer** from the drop-down list.

    b. Choose **Configure input transformer**

    c. for **Sample events**, enter the following:

```
{
  "detail":[]
}
```

    d. For **Target input transformer** do the following:

        i. For **Input Path**, enter the following:

```
{"detail":"$.detail"}
```

ii. For **Input Template**, enter the following:

```
{"message": <detail>}
```

e. Choose **Confirm.**.

15. Choose **Next**.
16. Choose **Next**.
17. Review the details of the rule and choose **Create rule**.

Step 4: Test the rule

To test your rule, create an Amazon S3 object by uploading a file to an EventBridge-enabled bucket. The created object will be logged in the Datadog Logs console.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge Connections(s)

1. Open the API destination page of the EventBridge console.
2. Choose the **Connections** tab.
3. Select the Connection(s) you created.
4. Choose **Delete**.
5. Enter the name of the connection and choose **Delete**.

To delete the EventBridge API destination(s)

1. Open the API destination page of the EventBridge console.
2. Select the API destinations(s) you created.
3. Choose **Delete**.
4. Enter the name of the API destination and choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Send events to Salesforce from Amazon EventBridge

You can use EventBridge to route events to third-party services, such as Salesforce.

In this tutorial, you'll use the EventBridge console to create a connection to Salesforce, an API destination that points to Salesforce, and a rule to route events to Salesforce.

Steps:

- Prerequisites
- Step 1: Create connection
- Step 2: Create API destination
- Step 3: Create rule
- Step 4: Test the rule
- Step 5: Clean up your resources

Prerequisites

To complete this tutorial, you'll need the following resources:

- A Salesforce account.
- A Salesforce connected app.
- A Salesforce security token.
- A Salesforce custom platform event.
- An EventBridge-enabled Amazon Simple Storage Service (Amazon S3) bucket.

Step 1: Create connection

To send events to Salesforce, you'll first have to establish a connection to the Salesforce API.

To create the connection

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose the **Connections** tab, and then choose **Create connection**.

4. Enter a name and description for the connection. For example, enter **`Salesforce`** as a name, and **`Salesforce API Connection`** as a description.
5. For **Destination type**, choose **Partners** and for **Partner Destinations**, select Salesforce from the drop-down list.
6. For **Authorization endpoint**, enter one of these:
   - If you're using a production org, enter **`https://`*`MyDomainName`*`.my.salesforce.com./services/oauth2/token`**
   - If you're using a sandbox without enhanced domains, enter **`https://`*`MyDomainName`*`--`*`SandboxName`*`.my. salesforce.com/services /oauth2/token`**
   - If you're using a sandbox with enhanced domains, enter **`https://`*`MyDomainName`*`--`*`SandboxName`*`.sandbox.my.salesforce.com/services/oauth2/token`**
7. For **HTTP method**, choose **POST** from the drop-down list.
8. For **Client ID**, enter the client ID from your Salesforce connected app.
9. For **Client secret**, enter the client secret from your Salesforce connected app.
10. For **OAuth Http Parameters**, enter the following key/value pair:

Key	Value
grant_type	client_credentials

11. Choose **Create**.

Step 2: Create API destination

Now that you've created the connection, next you'll create the API destination to use as the target of the rule.

To create the API Destination

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose **Create API destination**.

4. Enter a name and description for the API destination. For example, enter **`SalesforceAD`** for the name, and **`Salesforce API Destination`** for the description..
5. For **API destination endpoint**, enter **`https://`*`MyDomainName`*`.my.salesforce.com/services/data/v54.0/sobjects/`*`MyEvent__e`*** where **Myevent__e** is the platform event where you want to send information.
6. For **HTTP method**, choose **POST** from the drop-down list.
7. For **Invocation rate limit**, enter **`300`**.
8. For **Connection**, choose **Use an existing connection** and choose the `Salesforce` connection you created in step 1.
9. Choose **Create**.

Step 3: Create rule

Next, you'll create a rule to send events to Salesforce when an Amazon S3 object is created.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, enter **`SalesforceRule`** for the name, and **`Rule to send events to Salesforce for S3 object creation`** for the description.
5. For **Event bus**, choose **default**.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **Other**.
9. For **Event pattern**, enter the following:

```
{
  "source": ["aws.s3"]
}
```

10. Choose **Next**.

11. For **Target types**, choose **EventBridge API destination**.
12. For **API destination**, choose **Use an existing API destination**, and then choose the `SalesforceAD` destination you created in step 2.
13. For **Execution role**, choose **Create a new for role for this specific resource**.
14. For **Additional settings**, do the following:

    a. For **Configure target input**, choose **Input transformer** from the drop-down list.

    b. Choose **Configure input transformer**

    c. for **Sample events**, enter the following:

    ```
    {
      "detail":[]
    }
    ```

    d. For **Target input transformer** do the following:

    i. For **Input Path**, enter the following:

    ```
    {"detail":"$.detail"}
    ```

    ii. For **Input Template**, enter the following:

    ```
    {"message": <detail>}
    ```

    e. Choose **Confirm.**.
15. Choose **Next**.
16. Choose **Next**.
17. Review the details of the rule and choose **Create rule**.

Step 4: Test the rule

To test your rule, create an Amazon S3 object by uploading a file to an EventBridge-enabled bucket. The information about the created object will be sent to the Salesforce platform event.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge Connections(s)

1. Open the API destination page of the EventBridge console.
2. Choose the **Connections** tab.
3. Select the Connection(s) you created.
4. Choose **Delete**.
5. Enter the name of the connection and choose **Delete**.

To delete the EventBridge API destination(s)

1. Open the API destination page of the EventBridge console.
2. Select the API destinations(s) you created.
3. Choose **Delete**.
4. Enter the name of the API destination and choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.
2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Tutorial: Send events to Zendesk from Amazon EventBridge

You can use EventBridge to route events to third-party services like Zendesk.

In this tutorial, you'll use the EventBridge console to create a connection to Zendesk, an API destination that points to Zendesk, and a rule to route events to Zendesk.

Steps:

- Prerequisites
- Step 1: Create connection
- Step 2: Create API destination
- Step 3: Create rule
- Step 4: Test the rule
- Step 5: Clean up your resources

Prerequisites

To complete this tutorial, you'll need the following resources:

- A Zendesk account.
- An EventBridge-enabled Amazon Simple Storage Service (Amazon S3) bucket.

Step 1: Create connection

To send events to Zendesk, you'll first have to establish a connection to the Zendesk API.

To create the connection

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose the **Connections** tab, and then choose **Create connection**.
4. Enter a name and description for the connection. For example, enter **`Zendesk`** for the name, and **`Connection to Zendesk API`** for the description.
5. For **Authorization type**, choose **Basic (Username/Password)**.
6. For **Username**, enter your Zendesk username.

7. For **Password**, enter your Zendesk password.
8. Choose **Create**.

Step 2: Create API destination

Now that you've created the connection, you'll next create the API destination to use as the target of the rule.

To create the API Destination

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **API destinations**.
3. Choose **Create API destination**.
4. Enter a name and description for the API destination. For example, enter **`ZendeskAD`** for the name, and **`Zendesk API destination`** for the description.
5. For **API destination endpoint**, enter **`https://`*`your-subdomain`*`.zendesk.com/api/v2/tickets.json`**, where *`your-subdomain`* is the subdomain associated with your Zendesk account.
6. For **HTTP method**, choose **POST**.
7. For **Invocation rate limit**, enter **`10`**.
8. For **Connection**, choose **Use an existing connection** and choose the Zendesk connection you created in step 1.
9. Choose **Create**.

Step 3: Create rule

Next, create a rule to send events to Zendesk when an Amazon S3 object is created.

To create a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.

4. Enter a name and description for the rule. For example, enter **`ZendeskRule`** for the name, and **`Rule to send events to Zendesk when S3 objects are created`** for the description.
5. For **Event bus**, choose **default**.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **Other**.
9. For **Event pattern**, enter the following:

```
{
  "source": ["aws.s3"]
}
```

10. Choose **Next**.
11. For **Target types**, choose **EventBridge API destination**.
12. For **API destination**, choose **Use an existing API destination**, and then choose the `ZendeskAD` destination you created in step 2.
13. For **Execution role**, choose **Create a new for role for this specific resource**.
14. For **Additional settings**, do the following:

    a. For **Configure target input**, choose **Input transformer** from the drop-down list.

    b. Choose **Configure input transformer**

    c. for **Sample events**, enter the following:

```
{
  "detail":[]
}
```

    d. For **Target input transformer** do the following:

        i. For **Input Path**, enter the following:

```
{"detail":"$.detail"}
```

        ii. For **Input Template**, enter the following:

```
{"message": <detail>}
```

e. Choose **Confirm.**.

15. Choose **Next**.
16. Choose **Next**.
17. Review the details of the rule and choose **Create rule**.

Step 4: Test the rule

To test your rule, create an Amazon S3 object by uploading a file to an EventBridge-enabled bucket. When the event matches the rule, EventBridge will call the Zendesk Create Ticket API. The new ticket will appear in the Zendesk dashboard.

Step 5: Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the EventBridge Connections(s)

1. Open the API destination page of the EventBridge console.
2. Choose the **Connections** tab.
3. Select the Connection(s) you created.
4. Choose **Delete**.
5. Enter the name of the connection and choose **Delete**.

To delete the EventBridge API destination(s)

1. Open the API destination page of the EventBridge console.
2. Select the API destinations(s) you created.
3. Choose **Delete**.
4. Enter the name of the API destination and choose **Delete**.

To delete the EventBridge rule(s)

1. Open the Rules page of the EventBridge console.

2. Select the rule(s) that you created.
3. Choose **Delete**.
4. Choose **Delete**.

Creating an event bus in Amazon EventBridge

You can create a custom event bus to receive events from your applications. Your applications can also send events to the default event bus. When you create an event bus, you can attach a resource-based policy to grant permissions to other accounts. Then other accounts can send events to the event bus in the current account.

The following video goes through creating event buses: Creating an event bus

To create a Custom Event Bus - Classic

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Choose **Create event bus**.
4. Enter a name for the new event bus.
5. Choose the KMS key for EventBridge to use when encrypting the event data stored on the event bus.

> **Note**
>
> Schema discovery is not supported for event buses encrypted using a customer managed key. To enable schema discovery on an event bus, choose to use an AWS owned key. For more information, see KMS key options.

- Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

  This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

  This is the default.
- Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

  Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

  a. Specify an existing customer managed key, or choose **Create a new KMS key**.

     EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

  b. Choose the Amazon SQS queue to use as the dead-letter queue (DLQ) for this event bus, if any.

     EventBridge sends events that aren't successfully encrypted to the DLQ, if configured, so you can process them later.

6. (Optional) Under **Logs - optional**, you can set up how EventBridge logs event information, including how to configure those logs.

   For more information about event bus logs, see Logging event buses .

   CloudWatch logs is selected as a log destination by default, as is the ERROR log level. So, by default, EventBridge creates a new CloudWatch log group to which it sends log records containing the ERROR level of detail.

To have EventBridge send log records to any of the supported log destinations, do the following:

a. Under **Logs - optional**, choose the destinations to which you want log records delivered.

b. For **Log level**, choose the level of information for EventBridge to include in log records. The ERROR log level is selected by default.

For more information, see Specifying log level.

c. Select **Include detail data** if you want EventBridge to include event and target information in log records.

For more information, see Including details in logs.

d. Configure each log destination you selected.

7. Configure optional event bus features:

- Specify a resource-based policy by doing one of the following:
  - Enter the policy that includes the permissions to grant for the event bus. You can paste in a policy from another source or enter the JSON for the policy. You can use one of the example policies and modify it for your environment.
  - To use a template for the policy, choose **Load template**. Modify the policy as appropriate for your environment, including adding additional actions that you authorize the principal in the policy to use.

  For more information about granting permissions to an event bus through resource-based policies, see ???.

- Enable an archive (optional)

  You can create an archive of events so that you can easily replay them at a later time. For example, you might want to replay events to recover from errors or to validate new functionality in your application. For more information, see ???

  a. Under **Archives**, choose **Enabled**.

  b. Specify a name and description for the archive.

  You can't change the archive name once it has been created.

When creating an archive as part of creating a new event bus, you can't set the event retention period or an event pattern for the archive. You can specify these for the archive once it has been created. For more information, see Updating archives.

- Enable schema discovery (optional)

  Enable schema discovery to have EventBridge automatically infer schemas directly from events running on this event bus. For more information, see ???

  a. Under **Schema discovery**, choose **Enabled**.

  > **Note**
  >
  > Schema discovery is not supported for event buses encrypted using a customer managed key. To enable schema discovery on an event bus, choose to use an AWS owned key. For more information, see KMS key options.

- Specify tags (optional)

  A tag is a custom attribute label that you assign to an AWS resource. Use tags to identify and organize your AWS resources. Many AWS services support tagging, so you can assign the same tag to resources from different services to indicate that the resources are related. For more information, see ???

  a. Under **Tags**, choose **Add new tag**.

  b. Specify a key and, optionally, a value for the new tag.

8. Choose **Create**.

Updating an event bus in Amazon EventBridge

You can update the configuration of event buses after you create them. This includes the default event bus, which EventBridge creates in your account automatically.

To update an event bus (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Choose the event bus you want to update.

4. Do one or more of the following:

   - To create, edit, or delete an archive, see the following procedures:

     Creating archives

     Updating archives

     Deleting archives
   - To add or remove tags, see the following procedure:

     Managing event bus tags
   - To manage event bus permissions, see the following procedure:

     Managing event bus permissions
   - To change the AWS KMS key used to encrypt events, see the following procedure:

     Update encryption on an event bus

Updating a default event bus using AWS CloudFormation in EventBridge

CloudFormation enables you to configure and manage your AWS resources across accounts and regions in a centralized and repeatable manner by treating infrastructure as code. CloudFormation does this by letting you create *templates*, which define the resources you want to provision and manage.

Because EventBridge provisions the default event bus into your account automatically, you cannot create it using a CloudFormation template, as you normally would for any resource you wanted to include in a CloudFormation stack. To include the default event bus in a CloudFormation stack, you must first *import* it into a stack. Once you have imported the default event bus into a stack, you can then update the event bus properties as desired.

To import an existing resource into a new or existing CloudFormation stack, you need the following information:

- A unique identifier for the resource to import.

  For default event buses, the identifier is `Name` and then identifier value is `default`.

- A template that accurately describes the current properties of the existing resource.

  The template snippet below contains an `AWS::Events::EventBus` resource that describes the current properties of a default event bus. In this example, the event bus has been configured to use a customer managed key and DLQ for encryption at rest.

  Also, the `AWS::Events::EventBus` resource that describes the default event bus you want to import should include a `DeletionPolicy` property set to `Retain`.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Default event bus import example",
    "Resources": {
        "defaultEventBus": {
            "Type" : "AWS::Events::EventBus",
            "DeletionPolicy": "Retain",
            "Properties" : {
                "Name" : "default",
                "KmsKeyIdentifier" : "KmsKeyArn",
                "DeadLetterConfig" : {
                    "Arn" : "DLQ_ARN"
                }
            }
        }
    }
}
```

For more information, see Bringing existing resources into CloudFormation management in the *CloudFormation User Guide*.

Deleting an event bus in Amazon EventBridge

You can delete a custom or partner event bus. You cannot delete the default event bus. Deleting an event bus deletes the rules associated with that event bus.

To delete an event bus using the EventBridge console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.

3. Choose the event bus you want to delete.
4. Do one of the following:
   - Choose **Delete**.
   - Choose the name of the event bus.

     On the event bus details page, choose **Delete**.

Generating an CloudFormation template from an existing EventBridge event bus

AWS CloudFormation enables you to configure and manage your AWS resources across accounts and regions in a centralized and repeatable manner by treating infrastructure as code. CloudFormation does this by letting you create *templates*, which define the resources you want to provision and manage.

EventBridge enables you to generate templates from the existing event buses in your account, as an aid to help you jumpstart developing CloudFormation templates. In addition, EventBridge provides the option of including the rules associated with that event bus in your template. You can then use these templates as the basis for creating stacks of resources under CloudFormation management.

For more information on CloudFormation see *The CloudFormation User Guide.*

> **Note**
>
> EventBridge does not include managed rules in the generated template.

You can also generate a template from one or more rules contained in a selected event bus.

To generate an CloudFormation template from an event bus

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Choose the event bus from which you want to generate a CloudFormation template.

4. From the **Actions** menu, choose **CloudFormation Template**, and then choose which format you want EventBridge to generate the template in: **JSON** or **YAML**.

   EventBridge displays the template, generated in the selected format. By default, all rules associated with the event bus are included in the template.

   - To generate the template without including rules, deselect **Include rules on this EventBus**.

5. EventBridge gives you the option of downloading the template file, or copying the template to the clipboard.

   - To download the template file, choose **Download**.
   - To copy the template to the clipboard, choose **Copy**.

6. To exit the template, choose **Cancel**.

Once you've customized your CloudFormation template as necessary for your use case, you can use it to create stacks in CloudFormation.

Considerations when using CloudFormation templates generated from Amazon EventBridge

Consider the following factors when using a CloudFormation template you generated from an event bus:

- EventBridge does not include any passwords in the generate template.

  You can edit the template to include template parameters that enable users to specify passwords or other sensitive information when using the template to create or update a CloudFormation stack.

  In addition, users can use Secrets Manager to create a secret in the desired region and then edit the generated template to employ dynamic parameters.

- Targets in the generated template remain exactly as they were specified in the original event bus. This can lead to cross-region issues if you do not appropriately edit the template before using it to create stacks in other regions.

  Additionally, the generated template will not create the downstream targets automatically.

Permissions for event buses in Amazon EventBridge

The default event bus in your AWS account only allows events from one account. You can grant additional permissions to an event bus by attaching a resource-based policy to it. With a resource-based policy, you can allow `PutEvents`, `PutRule`, and `PutTargets` API calls from another account. You can also use IAM conditions in the policy to grant permissions to an organization, apply tags, or filter events to only those from a specific rule or account. You can set a resource-based policy for an event bus when you create it or afterward.

EventBridge APIs that accept an event bus `Name` parameter such as `PutRule`, `PutTargets`, `DeleteRule`, `RemoveTargets`, `DisableRule`, and `EnableRule` also accept the event bus ARN. Use these parameters to reference cross-account or cross-Region event buses through the APIs. For example, you can call `PutRule` to create a rule on an event bus in a different account without needing to assume a role.

You can attach the example policies in this topic to an IAM role to grant permission to send events to a different account or Region. Use IAM roles to set organization control policies and boundaries on who can send events from your account to other accounts. We recommend always using IAM roles when the target of a rule is an event bus. You can attach IAM roles using `PutTarget` calls. For information about creating a rule to send events to a different account or Region, see Sending and receiving events between AWS accounts in Amazon EventBridge.

Managing event bus permissions in Amazon EventBridge

Use the following procedure to modify the permissions for an existing event bus. For information about how to use CloudFormation to create an event bus policy, see AWS::Events::EventBusPolicy.

To manage permissions for an existing event bus

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the left navigation pane, choose **Event buses**.
3. In **Name**, choose the name of the event bus to manage permissions for.

   If a resource policy is attached to the event bus, the policy displays.
4. Choose **Manage permissions**, and then do one of the following:
   - Enter the policy that includes the permissions to grant for the event bus. You can paste in a policy from another source, or enter the JSON for the policy.

- To use a template for the policy, choose **Load template**. Modify the policy as appropriate for your environment, and add additional actions that you authorize the principal in the policy to use.

5. Choose **Update**.

The template provides example policy statements that you can customize for your account and environment. The template isn't a valid policy. You can modify the template for your use case, or you can copy one of the example policies and customize it.

The template loads policies that include an example of how to grant permissions to an account to use the `PutEvents` action, how to grant permissions to an organization, and how to grant permissions to the account to manage rules in the account. You can customize the template for your specific account, and then delete the other sections from the template. More example policies are included later in this section.

If you try to update the permissions for the bus but the policy contains an error, an error message indicates the specific issue in the policy.

```
 ### Choose which sections to include in the policy to match your use case. ###
 ### Be sure to remove all lines that start with ###, including the ### at the end of
the line. ###

 ### The policy must include the following: ###

 {
   "Version": "2012-10-17",
   "Statement": [

     ### To grant permissions for an account to use the PutEvents action, include the
following, otherwise delete this section: ###

     {

       "Sid": "AllowAccountToPutEvents",
       "Effect": "Allow",
       "Principal": {
         "AWS": "<ACCOUNT_ID>"
       },
       "Action": "events:PutEvents",
       "Resource": "arn:aws:events:us-east-1:123456789012:event-bus/default"
```

```
    },

    ### Include the following section to grant permissions to all members of your AWS
Organizations to use the PutEvents action ###

    {
      "Sid": "AllowAllAccountsFromOrganizationToPutEvents",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "events:PutEvents",
      "Resource": "arn:aws:events:us-east-1:123456789012:event-bus/default",
      "Condition": {
        "StringEquals": {
          "aws:PrincipalOrgID": "o-yourOrgID"
        }
      }
    },

    ### Include the following section to grant permissions to the account to manage
the rules created in the account ###

    {
      "Sid": "AllowAccountToManageRulesTheyCreated",
      "Effect": "Allow",
      "Principal": {
        "AWS": "<ACCOUNT_ID>"
      },
      "Action": [
        "events:PutRule",
        "events:PutTargets",
        "events:DeleteRule",
        "events:RemoveTargets",
        "events:DisableRule",
        "events:EnableRule",
        "events:TagResource",
        "events:UntagResource",
        "events:DescribeRule",
        "events:ListTargetsByRule",
        "events:ListTagsForResource"],
      "Resource": "arn:aws:events:us-east-1:123456789012:rule/default",
      "Condition": {
        "StringEqualsIfExists": {
          "events:creatorAccount": "<ACCOUNT_ID>"
        }
```

```
        }
    }]
}
```

Example policy: Send events to the default bus in a different account in Amazon EventBridge

The following example policy grants the account 111122223333 permission to publish events to the default event bus in the account 123456789012.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
       {
        "Sid": "sid1",
        "Effect": "Allow",
        "Principal": {"AWS":"arn:aws-eusc:iam::111112222333:root"},
        "Action": "events:PutEvents",
        "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
default"
        }
    ]
  }
```

Example policy: Send events to a custom bus in a different account in Amazon EventBridge

The following example policy grants the account 111122223333 permission to publish events to the `central-event-bus` in account 123456789012, but only for events with a source value set to `com.exampleCorp.webStore` and a `detail-type` set to `newOrderCreated`.

You attach this resource-based policy to the `central-event-bus` in the bus-owner account (123456789012), which is the account that receives the events. The publishing account (111122223333) is the IAM principal that the policy grants access to, so replace the example account IDs with your own bus-owner and publishing account IDs.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "WebStoreCrossAccountPublish",
      "Effect": "Allow",
      "Action": [
        "events:PutEvents"
      ],
      "Principal": {
        "AWS": "arn:aws-eusc:iam::123456789012:root"
      },
      "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
central-event-bus",
      "Condition": {
        "ForAllValues:StringEquals": {
          "events:source": "com.exampleCorp.webStore",
          "events:detail-type": "newOrderCreated"
        }
      }
    }
  ]
}
```

Example policy: Send events to an event bus in the same account in Amazon EventBridge

The following example policy attached to an event bus named `CustomBus1` allows the event bus to receive events from the same account and Region.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "events:PutEvents"
            ],
            "Resource": [
                "arn:aws-eusc:events:eusc-de-east-1:111122223333:event-bus/
CustomBus1"
            ]
        }
    ]
}
```

Example policy: Send events from specific rules to an Amazon EventBridge bus in a different Region

The following example policy grants the account 111122223333 permission to send events that match a rule named SendToUSE1AnotherAccount in the Middle East (Bahrain) and US West (Oregon) Regions to an event bus named CrossRegionBus in the US East (N. Virginia) in account 123456789012. The example policy is added to the event bus named CrossRegionBus in account 123456789012. The policy allows events only if they match a rule specified for the event bus in account 111122223333. The Condition statement restricts events to only events that match the rules with the specified rule ARN.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowSpecificRulesAsCrossRegionSource",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::111112222333:root"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
CrossRegionBus",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": [
```

```
            "arn:aws-eusc:events:us-west-2:111112222333:rule/CrossRegionBus/
SendToUSE1AnotherAccount",
            "arn:aws-eusc:events:me-south-1:111112222333:rule/CrossRegionBus/
SendToUSE1AnotherAccount"
          ]
        }
      }
    }
  ]
}
```

Example policy: Send events only from a specific Region to a different Region in Amazon EventBridge

The following example policy grants account 111122223333 permission to send all events that are generated in the Middle East (Bahrain) and US West (Oregon) Regions to the event bus named `CrossRegionBus` in account 123456789012 in the US East (N. Virginia) Region. Account 111122223333 doesn't have permission to send events that are generated in any other Region.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCrossRegionEventsFromUSWest2AndMESouth1",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::111112222333:root"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
CrossRegionBus",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": [
            "arn:aws-eusc:events:us-west-2:*:*",
            "arn:aws-eusc:events:me-south-1:*:*"
          ]
        }
```

```
      }
    }
  ]
}
```

Example policy: Deny sending events from specific Regions in Amazon EventBridge

The following example policy attached to an event bus named `CrossRegionBus` in account 123456789012 grants permission for the event bus to receive events from the account 111122223333, but not events that are generated in the US West (Oregon) Region.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1AllowAnyEventsFromAccount111112222333",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::111112222333:root"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
CrossRegionBus"
    },
    {
      "Sid": "2DenyAllCrossRegionUSWest2Events",
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-bus/
CrossRegionBus",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": [
            "arn:aws-eusc:events:us-west-2:*:*"
```

```
          ]
        }
      }
    }
  ]
}
```

Events in Amazon EventBridge

An *event* indicates a change in an environment such as an AWS environment, a SaaS partner service or application, or one of your applications or services. The following are examples of events:

- Amazon EC2 generates an event when the state of an instance changes, such as from pending to running.
- AWS CloudFormation generates an event when it creates, updates, or deletes a stack.
- AWS CloudTrail publishes events when you make API calls.

You can also set up scheduled events that are generated on a periodic basis.

Events are represented as JSON objects and they all have a similar structure, and the same top-level fields. For more information, see Event structure in the *EventBridge Events Reference*.

Events from AWS services

Many AWS services generate events that EventBridge receives. When an AWS service in your account sends an event to EventBridge, it goes to your account's default event bus.

For a list of AWS services that send events to EventBridge, and the events they send, see the EventBridge Events Reference.

AWS services send events to EventBridge on a *durable* or *best-effort basis*. For more information, see Event delivery level in the *EventBridge Events Reference*.

Amazon EventBridge: What is an event?

Amazon EventBridge: Where do events come from?

Sending events with PutEvents in Amazon EventBridge

The PutEvents action sends multiple events to EventBridge in a single request. For more information, see PutEvents in the *Amazon EventBridge API Reference* and put-events in the *AWS CLI Command Reference*.

Each PutEvents request can support a limited number of entries. For more information, see Amazon EventBridge quotas. The PutEvents operation attempts to process all entries in the natural order of the request. After you call PutEvents, EventBridge assigns each event a unique ID.

The following example Java code sends two identical events to EventBridge.

AWS SDK for Java Version 2.x

```
EventBridgeClient eventBridgeClient =
    EventBridgeClient.builder().build();

PutEventsRequestEntry requestEntry = PutEventsRequestEntry.builder()
    .resources("resource1", "resource2")
    .source("com.mycompany.myapp")
    .detailType("myDetailType")
    .detail("{ \"key1\": \"value1\", \"key2\": \"value2\" }")
    .build();

List <
PutEventsRequestEntry > requestEntries = new ArrayList <
PutEventsRequestEntry > ();
requestEntries.add(requestEntry);

PutEventsRequest eventsRequest = PutEventsRequest.builder()
    .entries(requestEntries)
    .build();

PutEventsResponse result = eventBridgeClient.putEvents(eventsRequest);

for (PutEventsResultEntry resultEntry: result.entries()) {
    if (resultEntry.eventId() != null) {
        System.out.println("Event Id: " + resultEntry.eventId());
    } else {
        System.out.println("PutEvents failed with Error Code: " +
 resultEntry.errorCode());
```

```
    }
}
```

AWS SDK for Java Version 1.0

```
EventBridgeClient eventBridgeClient =
    EventBridgeClient.builder().build();

PutEventsRequestEntry requestEntry = new PutEventsRequestEntry()
        .withTime(new Date())
        .withSource("com.mycompany.myapp")
        .withDetailType("myDetailType")
        .withResources("resource1", "resource2")
        .withDetail("{ \"key1\": \"value1\", \"key2\": \"value2\" }");

PutEventsRequest request = new PutEventsRequest()
        .withEntries(requestEntry, requestEntry);

PutEventsResult result = awsEventsClient.putEvents(request);

for (PutEventsResultEntry resultEntry : result.getEntries()) {
    if (resultEntry.getEventId() != null) {
        System.out.println("Event Id: " + resultEntry.getEventId());
    } else {
        System.out.println("Injection failed with Error Code: " +
 resultEntry.getErrorCode());
    }
}
```

After you run this code, the `PutEvents` result includes an array of response entries. Each entry in the response array corresponds to an entry in the request array in order from the beginning to the end of the request and response. The response `Entries` array always includes the same number of entries as the request array.

Handling failures with `PutEvents`

By default, if an individual entry within a request fails, EventBridge continues processing the rest of the entries in the request. A response `Entries` array can include both successful and unsuccessful entries. You must detect unsuccessful entries and include them in a subsequent call.

Successful result entries include an `Id` value, and unsuccessful result entries include `ErrorCode` and `ErrorMessage` values. `ErrorCode` describes the type of error. `ErrorMessage` provides more information about the error. The following example has three result entries for a `PutEvents` request. The second entry is unsuccessful.

```
{
    "FailedEntryCount": 1,
    "Entries": [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {   "ErrorCode": "InternalFailure",
            "ErrorMessage": "Internal Service Failure"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ]
}
```

Note

If you use `PutEvents` to publish an event to an event bus that does not exist, EventBridge event matching will not find a corresponding rule and will drop the event. Although EventBridge will send a `200` response, it will not fail the request or include the event in the `FailedEntryCount` value of the request response.

You can include entries that are unsuccessful in subsequent `PutEvents` requests. First, to find out if there are failed entries in the request, check the `FailedRecordCount` parameter in `PutEventsResult`. If it isn't zero, then you can add each `Entry` that has an `ErrorCode` that is not null to a subsequent request. The following example shows a failure handler.

```
PutEventsRequestEntry requestEntry = new PutEventsRequestEntry()
        .withTime(new Date())
        .withSource("com.mycompany.myapp")
        .withDetailType("myDetailType")
        .withResources("resource1", "resource2")
        .withDetail("{ \"key1\": \"value1\", \"key2\": \"value2\" }");
```

```
List<PutEventsRequestEntry> putEventsRequestEntryList = new ArrayList<>();
for (int i = 0; i < 3; i++) {
    putEventsRequestEntryList.add(requestEntry);
}

PutEventsRequest putEventsRequest = new PutEventsRequest();
putEventsRequest.withEntries(putEventsRequestEntryList);
PutEventsResult putEventsResult = awsEventsClient.putEvents(putEventsRequest);

while (putEventsResult.getFailedEntryCount() > 0) {
    final List<PutEventsRequestEntry> failedEntriesList = new ArrayList<>();
    final List<PutEventsResultEntry> PutEventsResultEntryList =
 putEventsResult.getEntries();
    for (int i = 0; i < PutEventsResultEntryList.size(); i++) {
        final PutEventsRequestEntry putEventsRequestEntry =
 putEventsRequestEntryList.get(i);
        final PutEventsResultEntry putEventsResultEntry =
 PutEventsResultEntryList.get(i);
        if (putEventsResultEntry.getErrorCode() != null) {
            failedEntriesList.add(putEventsRequestEntry);
        }
    }
    putEventsRequestEntryList = failedEntriesList;
    putEventsRequest.setEntries(putEventsRequestEntryList);
    putEventsResult = awsEventsClient.putEvents(putEventsRequest);
    }
```

Sending events using the AWS CLI

You can use the AWS CLI to send custom events to EventBridge so they can be processed. The following example puts one custom event into EventBridge:

```
aws events put-events \
--entries '[{"Time": "2016-01-14T01:02:03Z", "Source": "com.mycompany.myapp",
 "Resources": ["resource1", "resource2"], "DetailType": "myDetailType", "Detail":
 "{ \"key1\": \"value1\", \"key2\": \"value2\" }"}]'
```

You can also create a JSON file that contains custom events.

```
[
  {
    "Time": "2016-01-14T01:02:03Z",
```

```
    "Source": "com.mycompany.myapp",
    "Resources": [
      "resource1",
      "resource2"
    ],
    "DetailType": "myDetailType",
    "Detail": "{ \"key1\": \"value1\", \"key2\": \"value2\" }"
  }
]
```

Then, to use the AWS CLI to read the entries from this file and send events, at a command prompt, type:

```
aws events put-events --entries file://entries.json
```

Calculating PutEvents event entry size

When you send custom events to EventBridge using the `PutEvents` action, you can batch up to 10 event entries into one request for efficiency. The total request size—that is, the sum of all event entries in the request—must be less than 1 MB (1,048,576 bytes). This limit applies to the request as a whole, not to individual entries. A single event can use up to the full 1 MB if it is the only entry in the request. You can calculate the entry size before you send the events.

> **Note**
>
> The size limit is imposed on the *entry*. Even if the entry is less than the size limit, the *event* in EventBridge is always larger than the entry size due to the necessary characters and keys of the JSON representation of the event. For more information, see Events in Amazon EventBridge.

EventBridge calculates the `PutEventsRequestEntry` size as follows:

- If specified, the `Time` parameter is 14 bytes.
- The `Source` and `DetailType` parameters are the number of bytes for their UTF-8 encoded forms.
- If specified, the `Detail` parameter is the number of bytes for its UTF-8 encoded form.
- If specified, each entry of the `Resources` parameter is the number of bytes for its UTF-8 encoded forms.

The following example Java code calculates the size of a given `PutEventsRequestEntry` object. In order to verify the 1MB limit is not violated, you need to perform the calculation for all events in a request.

```
int getSize(PutEventsRequestEntry entry) {
    int size = 0;
    if (entry.getTime() != null) {
        size += 14;
    }
    size += entry.getSource().getBytes(StandardCharsets.UTF_8).length;
    size += entry.getDetailType().getBytes(StandardCharsets.UTF_8).length;
    if (entry.getDetail() != null) {
```

```
        size += entry.getDetail().getBytes(StandardCharsets.UTF_8).length;
    }
    if (entry.getResources() != null) {
        for (String resource : entry.getResources()) {
            if (resource != null) {
                size += resource.getBytes(StandardCharsets.UTF_8).length;
            }
        }
    }
    return size;
}
```

Note

If the entry size is larger than 1 MB (1,048,576 bytes), we recommend uploading the event to an Amazon S3 bucket and including the `Object URL` in the `PutEvents` entry.

How EventBridge retries delivering events

Sometimes an event isn't successfully delivered to the target specified in a rule. This can happen, for example:

- If the target resource is unavailable
- Due to network conditions

When an event isn't successfully delivered to a target because of retriable errors, EventBridge retries sending the event. You set the length of time it tries, and number of retry attempts in the **Retry policy** settings for the target. By default, EventBridge retries sending the event for 24 hours and up to 185 times with an exponential back off and *jitter*, or randomized delay.

If an event isn't delivered after all retry attempts are exhausted, the event is dropped and EventBridge doesn't continue to process it.

To avoid losing events after they fail to be delivered to a target, configure a dead-letter queue (DLQ) to receive all failed events. For more information, see the section called “Using dead-letter queues”.

Using dead-letter queues to process undelivered events in EventBridge

To avoid losing events after they fail to be delivered to a target, you can configure a dead-letter queue (DLQ) and send all failed events to it for processing later.

EventBridge DLQs are standard Amazon SQS queues that EventBridge uses to store events that couldn't successfully be delivered to a target. When you create a rule and add a target, you can choose whether or not to use a DLQ. When you configure a DLQ, you can retain any events that weren't successfully delivered. Then you can resolve the issue that resulted in the failed event delivery and process the events at a later time.

When you configure a DLQ for a target of a rule, EventBridge sends the events with failed invocations to the Amazon SQS queue selected.

Event errors are handled in different ways. Some events are dropped or sent to a DLQ without any retry attempts. For example, for errors that result from missing permissions to a target, a target resource that no longer exists, or a target that cannot be found due to an invalid address or DNS lookup failure, no retry attempts will happen until action is taken to resolve the underlying issue. EventBridge sends these events directly to the target DLQ, if you have specified one.

When an event delivery fails, EventBridge publishes an event to Amazon CloudWatch metrics indicating that a target `invocation` failed. If you use a DLQ, additional metrics are sent to CloudWatch including `InvocationsSentToDLQ` and `InvocationsFailedToBeSentToDLQ`.

You can also specify DLQs for event buses, if you use AWS KMS customer managed keys to encrypt events at rest. For more information, see ???.

Each message in your DLQ will include the following custom attributes:

- `RULE_ARN`
- `TARGET_ARN`
- `ERROR_CODE`

  The following is a sample of the error codes a DLQ can return:

  - `ACTION_DECRYPTION_FAILURE`
  - `CONNECTION_FAILURE`
  - `CROSS_ACCOUNT_INGESTION_FAILED`
  - `CROSS_REGION_INGESTION_FAILED`

  - ERROR_FROM_TARGET
  - EVENT_DECRYPTION_FAILURE
  - EVENT_ENCRYPTION_FAILURE
  - EVENTS_IN_BATCH_REQUEST_REJECTED
  - FAILED_TO_ASSUME_ROLE
  - INTERNAL_ERROR
  - INVALID_JSON
  - INVALID_PARAMETER
  - NO_PERMISSIONS
  - NO_RESOURCE
  - RESOURCE_ALREADY_EXISTS
  - RESOURCE_LIMIT_EXCEEDED
  - RESOURCE_MODIFICATION_COLLISION
  - RULE_DECRYPTION_FAILURE
  - SDK_CLIENT_ERROR
  - THIRD_ACCOUNT_HOP_DETECTED
  - THIRD_REGION_HOP_DETECTED
  - THROTTLING
  - TIMEOUT
  - TRANSIENT_ASSUME_ROLE
  - UNKNOWN
- ERROR_MESSAGE
- EXHAUSTED_RETRY_CONDITION

  The following conditions can be returned:

  - MaximumRetryAttempts
  - MaximumEventAgeInSeconds
- RETRY_ATTEMPTS

The following video goes over setting up DLQs: Using dead-letter queues (DLQs)

Topics

- Considerations for using a dead-letter queue
- Granting permissions to the dead-letter queue
- How to resend events from a dead-letter queue

Considerations for using a dead-letter queue

Consider the following when configuring a DLQ for EventBridge.

- Only standard queues are supported. You can't use a FIFO queue for a DLQ in EventBridge.
- EventBridge includes event metadata and message attributes in the message, including: the Error Code, Error Message, the Exhausted Retry Condition, Rule ARN, Retry Attempts, and the Target ARN. You can use these values to identify an event and the cause of the failure.
- Permissions for DLQs in the same account:
  - If you add a target to a rule using the console, and you choose an Amazon SQS queue in the same account, a resource-based policy that grants EventBridge access to the queue is attached to the queue for you.
  - If you use the `PutTargets` operation of the EventBridge API to add or update a target for a rule, and you choose an Amazon SQS queue in the same account, you must manually grant permissions to the queue selected. To learn more, see Granting permissions to the dead-letter queue.

- Permissions for using Amazon SQS queues from a different AWS account.
  - If you create a rule from the console, queues from other accounts aren't displayed for you to select. You must provide the ARN for the queue in the other account, and then manually attach a resource-based policy to grant permission to the queue. To learn more, see Granting permissions to the dead-letter queue.
  - If you create a rule using the API, you must manually attach a resource-based policy to the SQS queues in another account that is used as the dead-letter queue. To learn more, see Granting permissions to the dead-letter queue.
- The Amazon SQS queue you use must be in the same Region in which you create the rule.

Granting permissions to the dead-letter queue

To successfully deliver events to the queue, EventBridge must have permission to do so. When you specify a DLQ using the EventBridge console, the permissions are automatically added. This includes:

- When you configure a DLQ for a target of a rule.
- When you configure a DLQ for an event bus where you've specified that EventBridge use an AWS KMS customer managed key to encrypt events at rest.

  For more information, see ???.

If you specify a DLQ using the API, or use a queue that is in a different AWS account, you must manually create a resource-based policy that grants the required permissions and then attach it to the queue.

Target dead-letter queue permissions example

The following resource-based policy demonstrates how to grant the required permissions for EventBridge to send event messages to an Amazon SQS queue. The policy example grants the EventBridge service permissions to use the `SendMessage` operation to send messages to a queue named "MyEventDLQ". The queue must be in the us-west-2 Region in AWS account 123456789012. The `Condition` statement allows only requests that come from a rule named "MyTestRule" that is created in the us-west-2 Region in the AWS account 123456789012.

```
{
  "Sid": "Dead-letter queue permissions",
```

```
  "Effect": "Allow",
  "Principal": {
     "Service": "events.amazonaws.com"
  },
  "Action": "sqs:SendMessage",
  "Resource": "arn:aws:sqs:us-west-2:123456789012:MyEventDLQ",
  "Condition": {
    "ArnEquals": {
      "aws:SourceArn": "arn:aws:events:us-west-2:123456789012:rule/MyTestRule"
    }
  }
}
```

Event bus dead-letter queue permissions example

The following resource-based policy demonstrates how to grant the required permissions when specifying a DLQ for an event bus. In this case, `aws:SourceArn` specifies the ARN of the event bus sending the events to the DLQ. Here again in this example, the queue must be in the same Region as the event bus.

```
{
  "Sid": "Dead-letter queue permissions",
  "Effect": "Allow",
  "Principal": {
     "Service": "events.amazonaws.com"
  },
  "Action": "sqs:SendMessage",
  "Resource": "arn:aws:sqs:region:account-id:queue-name",
  "Condition": {
    "ArnEquals": {
      "aws:SourceArn": "arn:aws:events:region:account-id:event-bus/event-bus-arn"
    }
  }
}
```

To attach the policy to the queue, use the Amazon SQS console, open the queue, then choose the **Access policy** and edit the policy. You can also use the AWS CLI. To learn more, see Amazon SQS permissions.

How to resend events from a dead-letter queue

You can move messages out of a DLQ in two ways:

- Avoid writing Amazon SQS consumer logic – Set your DLQ as an event source to the Lambda function to drain your DLQ.
- Write Amazon SQS consumer logic – Use the Amazon SQS API, AWS SDK, or AWS CLI to write custom consumer logic for polling, processing, and deleting the messages in the DLQ.

AWS service events delivered via AWS CloudTrail

AWS CloudTrail is a service that automatically records events such as AWS API calls. You can create EventBridge rules that use the information from CloudTrail. For more information about CloudTrail, see What is AWS CloudTrail?.

> **Note**
>
> CloudTrail also records the Custom Event Bus management API calls, such as `CreateEventBus`, `CreateSubscriber`, and `PutResourcePolicy`, made through the `eventsv2` endpoint. See the section called "Observability".

CloudTrail sends the following types of events to the default EventBridge event bus. In each case, the `detail-type` value of the event is the listed event type:

- `AWS API Call via CloudTrail`

  Events that represent a request to a public AWS service API.

  For more information, see Understanding CloudTrail events in the *AWS CloudTrail User Guide*.
- `AWS Console Signin via CloudTrail`

  Attempts to sign in to the AWS Management Console, the AWS Discussion Forums, and the AWS Support Center.

  For more information, see AWS Management Console sign-in events in the *AWS CloudTrail User Guide*.
- `AWS Console Action via CloudTrail`

  Actions that were taken in the console that were not an API calls.

  For more information, see AWS Management Console sign-in events in the *AWS CloudTrail User Guide*.

- `AWS Service Event via CloudTrail`

  Events created by AWS services but are not directly triggered by a request to a public AWS service API.

  For more information, see AWS service events in the *AWS CloudTrail User Guide*.

- `AWS Insight via CloudTrail`

  Insights events are triggered by CloudTrail when customer enables the CloudTrail Insight feature.

  For more information, see CloudTrail Insights in the *AWS CloudTrail User Guide*.

- `AWS Network Activity Event via CloudTrail`

  Network activity events capture API calls made through VPC endpoints from private VPCs. These events require a trail configured with network activity event selectors for the relevant event source.

  For more information, see Logging network activity events in the *AWS CloudTrail User Guide*.

To record events with one of the CloudTrail `detail-type` values, you must enable a CloudTrail trail with logging. For more information, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.

> **Note**
>
> All CloudTrail events are delivered to the default event bus only. To process CloudTrail events on a Custom Event Bus - Classic, create a rule on the default bus that forwards matching events to your Classic custom bus.

> **Note**
>
> If you forward read-only management events to another event bus (a Custom Event Bus - Classic, or a bus in another account or Region), two rules need the state `ENABLED_WITH_ALL_CLOUDTRAIL_MANAGEMENT_EVENTS`: the forwarding rule on the default event bus, and the rule on the destination event bus. A rule in the default `ENABLED` state does not match read-only management events, so its targets are not invoked.

The rule state controls which event categories are matched:

- *Write (mutating) management events* — Matched by rules in the default `ENABLED` state. No special configuration needed beyond an active trail.
- *Read-only management events* — Matched only by rules with state set to `ENABLED_WITH_ALL_CLOUDTRAIL_MANAGEMENT_EVENTS`. For more information, see ???.
- *Data events* — Matched by rules in the default `ENABLED` state. The trail must be configured to capture the specific data event types. For more information, see Logging data events in the *AWS CloudTrail User Guide*.
- *Network activity events* — Matched by rules in the default `ENABLED` state. The trail must be configured with network activity event selectors for the relevant event source, and the API call must be made through a VPC endpoint.

Some occurrences in AWS services can be reported to EventBridge both by the service itself and by CloudTrail. For example, an Amazon EC2 API call that starts an instance generates multiple events:

- `EC2 Instance State-change Notification` events sent directly from Amazon EC2 to EventBridge, as the instance enters the `pending` and then `running` states. For example:

```
{
    . . .
   "detail-type":"EC2 Instance State-change Notification",
   "source":"aws.ec2",
    . . .
   "detail":{
      "instance-id":"i-abcd1111",
      "state":"pending"
   }
}
```

- An `AWS API Call via CloudTrail` event sent from CloudTrail to EventBridge that represents the API call itself. For example:

```
{
    . . .
   "detail-type":"AWS API Call via CloudTrail",
   "source":"aws.ec2",
    . . .
   ],
```

```
  "detail": {
    "eventSource": "ec2.amazonaws.com",
    "eventName": "StartInstances"
    }
}
```

Note

If you use a `Put*Events` API call event as the basis for creating an event pattern, make sure the final event pattern does not exceed 1 MB. The maximum size of any `Put*Events` requests is 1 MB. For more information, see ???.

For more information about the services that CloudTrail supports, see CloudTrail supported services and integrations in the *CloudTrail User Guide*.

Receiving read-only management events from AWS services

You can set up rules on your default event bus to receive read-only *management events* from AWS services via CloudTrail. Management events provide visibility into management operations that are performed on resources in your AWS account. These are also known as control plane operations. For more information, see Logging management events in the *CloudTrail User Guide*.

For each rule on the default event bus, you can set the rule state to control the types of events to receive:

- Disable the rule so that EventBridge does not match events against the rule.
- Enable the rule so that EventBridge matches events against the rule, except for read-only AWS management events delivered through CloudTrail.
- Enable the rule so that EventBridge matches all events against the rule, *including* read-only management events delivered through CloudTrail.

Partner event buses do not receive AWS events.

Some things to consider when deciding whether to receive read-only management events:

- Certain read-only management events, such as AWS Key Management Service `GetKeyPolicy` and `DescribeKey`, or IAM `GetPolicy` and `GetRole` events, occur at a much higher volume than typical change events.
- You may already be receiving read-only management events even if those events don't start with `Describe`, `Get`, or `List`. One such example is `TestEventPattern` from EventBridge.

  For a list of read-only management events that do not adhere to the `Describe`, `Get`, or `List` naming convention, by AWS services, see ???.

To create a rule that receives read-only management events using the AWS CLI

- Use the `put-rule` command to create or update the rule, using parameters to:
  - Specify that the rule belongs on the default event bus, or a specific Custom Event Bus - Classic
  - Set rule state as `ENABLED_WITH_ALL_CLOUDTRAIL_MANAGEMENT_EVENTS`

```
aws events put-rule --name "ruleForManagementEvents" --event-bus-name
"default" --state "ENABLED_WITH_ALL_CLOUDTRAIL_MANAGEMENT_EVENTS"
```

> **Note**
>
> Enabling a rule for CloudWatch management events is supported through the AWS CLI and CloudFormation templates only.

Example

The following example illustrates how to match against specific events. Best practice is to define a dedicated rule for matching specific events, for clarity and ease of editing.

In this case, the dedicated rule matches the `AssumeRole` management event from AWS Security Token Service.

```
{
    "source" : [ "aws.sts" ],
    "detail-type": ["AWS API Call via CloudTrail"],
    "detail" : {
```

```
        "eventName" : ["AssumeRole"]
    }
}
```

Management events generated by AWS services in EventBridge

In general, APIs that generate management (or read-only) events start with the verbs `Describe`, `Get`, or `List`. The table below list AWS services and the management events they generate that do not follow this naming convention. For more information on management events, see ???.

Management events that don't start with `Describe`, `Get`, or `List`

The following table list AWS services and the management events they generate that do not follow typical naming conventions of starting with `Describe`, `Get`, or `List`.

Service	Event name	Event type
Alexa for Business	ResolveRoom	API call
Alexa for Business	SearchAddressBooks	API call
Alexa for Business	SearchContacts	API call
Alexa for Business	SearchDevices	API call
Alexa for Business	SearchProfiles	API call
Alexa for Business	SearchRooms	API call
Alexa for Business	SearchSkillGroups	API call
Alexa for Business	SearchUsers	API call
IAM Access Analyzer	ValidatePolicy	API call
AWS AdSpace Clean Rooms	BatchGetSchema	API call
AWS Amplify UI Builder	ExportComponents	API call
AWS Amplify UI Builder	ExportForms	API call
AWS Amplify UI Builder	ExportThemes	API call

Service	Event name	Event type
Amazon OpenSearch Service	BatchGetCollection	API call
Amazon API Gateway	ExportApi	API call
AWS AppConfig	ValidateConfiguration	API call
Amazon AppFlow	RetrieveConnectorData	API call
Amazon CloudWatch Application Insights	UpdateApplicationDashboardConfiguration	API call
Amazon Athena	BatchGetNamedQuery	API call
Amazon Athena	BatchGetPreparedStatement	API call
Amazon Athena	BatchGetQueryExecution	API call
Amazon Athena	CheckQueryCompatibility	API call
Amazon Athena	ExportNotebook	API call
AWS Auto Scaling	AreScalableTargetsRegistered	API call
AWS Auto Scaling	Test	API call
AWS Marketplace	SearchAgreements	API call
AWS Backup	CreateLegalHold	API call
AWS Backup	ExportBackupPlanTemplate	API call
AWS Backup gateway	TestHypervisorConfiguration	API call
AWS Billing and Cost Management	AWSPaymentInstrumentGateway.Get	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.DescribeMakePaymentPage	Console action

Service	Event name	Event type
AWS Billing and Cost Management	AWSPaymentPortalService.DescribePaymentsDashboard	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetAccountPreferences	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetAdvancePaySummary	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetAsoBulkDownload	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetBillingContactAddress	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetDocuments	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetEligiblePaymentInstruments	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetEntitiesByIds	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetFundingDocuments	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetKybcValidationStatus	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetOneTimePasswordStatus	Console action

Service	Event name	Event type
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentHistory	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentProfileByArn	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentProfileCurrencies	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentProfiles	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentProfileServiceProviders	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetPaymentsDue	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetRemittanceInformation	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetTaxInvoiceMetadata	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetTermsAndConditionsForProgramGroup	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetTransactionsHistory	Console action
AWS Billing and Cost Management	AWSPaymentPortalService.GetUnappliedFunds	Console action

Service	Event name	Event type
AWS Billing and Cost Management	AWSPaymentPortalService.GetUnpaidInvoices	Console action
AWS Billing and Cost Management	AWSPaymentPreferenceGateway.Get	Console action
AWS Billing and Cost Management	CancelBulkDownload	Console action
AWS Billing and Cost Management	DownloadCommercialInvoice	Console action
AWS Billing and Cost Management	DownloadCsv	Console action
AWS Billing and Cost Management	DownloadDoc	Console action
AWS Billing and Cost Management	DownloadECSVForBillingPeriod	Console action
AWS Billing and Cost Management	DownloadPaymentHistory	Console action
AWS Billing and Cost Management	DownloadRegistrationDocument	Console action
AWS Billing and Cost Management	DownloadTaxInvoice	Console action
AWS Billing and Cost Management	FindBankRedirectPaymentInstruments	Console action
AWS Billing and Cost Management	FindECSVForBillingPeriod	Console action
AWS Billing and Cost Management	ValidateReportDestination	Console action

Service	Event name	Event type
AWS Billing and Cost Management	VerifyChinaPaymentEligibility	Console action
Amazon Braket	SearchCompilations	API call
Amazon Braket	SearchDevices	API call
Amazon Braket	SearchQuantumTasks	API call
Amazon Connect Customer Cases	BatchGetField	API call
Amazon Connect Customer Cases	SearchCases	API call
Amazon Connect Customer Cases	SearchRelatedItems	API call
Amazon Chime	RetrieveDataExports	API call
Amazon Chime	SearchChannels	API call
Amazon Chime SDK Identity	DeleteProfile	Service event
Amazon Chime SDK Identity	DeleteWorkTalkAccount	Service event
AWS Clean Rooms	BatchGetSchema	API call
Amazon Cloud Directory	BatchRead	API call
Amazon Cloud Directory	LookupPolicy	API call
CloudFormation	DetectStackDrift	API call
CloudFormation	DetectStackResourceDrift	API call
CloudFormation	DetectStackSetDrift	API call
CloudFormation	EstimateTemplateCost	API call

Service	Event name	Event type
CloudFormation	ValidateTemplate	API call
AWS CloudShell	RedeemCode	API call
AWS CloudTrail	LookupEvents	API call
AWS CodeArtifact	ReadFromRepository	API call
AWS CodeArtifact	SearchPackages	API call
AWS CodeArtifact	VerifyResourcesExistForTagris	API call
AWS CodeBuild	BatchGetBuildBatches	API call
AWS CodeBuild	BatchGetBuilds	API call
AWS CodeBuild	BatchGetProjects	API call
AWS CodeBuild	BatchGetReportGroups	API call
AWS CodeBuild	BatchGetReports	API call
AWS CodeBuild	BatchPutCodeCoverages	API call
AWS CodeBuild	BatchPutTestCases	API call
AWS CodeBuild	RequestBadge	Service event
AWS CodeCommit	BatchDescribeMergeConflicts	API call
AWS CodeCommit	BatchGetCommits	API call
AWS CodeCommit	BatchGetPullRequests	API call
AWS CodeCommit	BatchGetRepositories	API call
AWS CodeCommit	EvaluatePullRequestApprovalRules	API call
AWS CodeCommit	GitPull	API call

Service	Event name	Event type
AWS CodeDeploy	BatchGetApplicationRevisions	API call
AWS CodeDeploy	BatchGetApplications	API call
AWS CodeDeploy	BatchGetDeploymentGroups	API call
AWS CodeDeploy	BatchGetDeployment Instances	API call
AWS CodeDeploy	BatchGetDeployments	API call
AWS CodeDeploy	BatchGetDeploymentTargets	API call
AWS CodeDeploy	BatchGetOnPremises Instances	API call
Amazon CodeGuru Profiler	BatchGetFrameMetricData	API call
Amazon CodeGuru Profiler	SubmitFeedback	API call
AWS CodePipeline	PollForJobs	API call
AWS CodePipeline	PollForThirdPartyJobs	API call
AWS CodeConnections	StartAppRegistrationHandshake	API call
AWS CodeConnections	StartOAuthHandshake	API call
AWS CodeConnections	ValidateHostWebhook	API call
Amazon CodeWhisperer	CreateCodeScan	API call
Amazon CodeWhisperer	CreateProfile	API call
Amazon CodeWhisperer	CreateUploadUrl	API call
Amazon CodeWhisperer	GenerateRecommendations	API call
Amazon CodeWhisperer	UpdateProfile	API call

Service	Event name	Event type
Amazon Cognito Identity	LookupDeveloperIdentity	API call
Amazon Cognito user pools	AdminGetDevice	API call
Amazon Cognito user pools	AdminGetUser	API call
Amazon Cognito user pools	AdminListDevices	API call
Amazon Cognito user pools	AdminListGroupsForUser	API call
Amazon Cognito user pools	AdminListUserAuthEvents	API call
Amazon Cognito user pools	Beta_Authorize_GET	Service event
Amazon Cognito user pools	Confirm_GET	Service event
Amazon Cognito user pools	ConfirmForgotPassword_GET	Service event
Amazon Cognito user pools	Error_GET	Service event
Amazon Cognito user pools	ForgotPassword_GET	Service event
Amazon Cognito user pools	IntrospectToken	API call
Amazon Cognito user pools	Login_Error_POST	Service event
Amazon Cognito user pools	Login_GET	Service event
Amazon Cognito user pools	Mfa_GET	Service event
Amazon Cognito user pools	MfaOption_GET	Service event
Amazon Cognito user pools	ResetPassword_GET	Service event
Amazon Cognito user pools	Signup_GET	Service event
Amazon Cognito user pools	UserInfo_GET	Service event
Amazon Cognito user pools	UserInfo_POST	Service event

Service	Event name	Event type
Amazon Cognito Sync	BulkPublish	API call
Amazon Comprehend	BatchContainsPiiEntities	API call
Amazon Comprehend	BatchDetectDominantLanguage	API call
Amazon Comprehend	BatchDetectEntities	API call
Amazon Comprehend	BatchDetectKeyPhrases	API call
Amazon Comprehend	BatchDetectPiiEntities	API call
Amazon Comprehend	BatchDetectSentiment	API call
Amazon Comprehend	BatchDetectSyntax	API call
Amazon Comprehend	BatchDetectTargetedSentiment	API call
Amazon Comprehend	ClassifyDocument	API call
Amazon Comprehend	ContainsPiiEntities	API call
Amazon Comprehend	DetectDominantLanguage	API call
Amazon Comprehend	DetectEntities	API call
Amazon Comprehend	DetectKeyPhrases	API call
Amazon Comprehend	DetectPiiEntities	API call
Amazon Comprehend	DetectSentiment	API call
Amazon Comprehend	DetectSyntax	API call
Amazon Comprehend	DetectTargetedSentiment	API call
Amazon Comprehend	DetectToxicContent	API call

Service	Event name	Event type
AWS Compute Optimizer	ExportAutoScalingGroupRecommendations	API call
AWS Compute Optimizer	ExportEBSVolumeRecommendations	API call
AWS Compute Optimizer	ExportECInstanceRecommendations	API call
AWS Compute Optimizer	ExportECSServiceRecommendations	API call
AWS Compute Optimizer	ExportLambdaFunctionRecommendations	API call
AWS Compute Optimizer	ExportRDSInstanceRecommendations	API call
AWS Config	BatchGetAggregateResourceConfig	API call
AWS Config	BatchGetResourceConfig	API call
AWS Config	SelectAggregateResourceConfig	API call
AWS Config	SelectResourceConfig	API call
Amazon Connect Customer	AdminGetEmergencyAccessToken	API call
Amazon Connect Customer	SearchQueues	API call
Amazon Connect Customer	SearchRoutingProfiles	API call
Amazon Connect Customer	SearchSecurityProfiles	API call
Amazon Connect Customer	SearchUsers	API call

Service	Event name	Event type
AWS Glue DataBrew	SendProjectSessionAction	API call
AWS Data Pipeline	EvaluateExpression	API call
AWS Data Pipeline	QueryObjects	API call
AWS Data Pipeline	ValidatePipelineDefinition	API call
AWS DataSync	VerifyResourcesExistForTagris	API call
AWS DeepLens	BatchGetDevice	API call
AWS DeepLens	BatchGetModel	API call
AWS DeepLens	BatchGetProject	API call
AWS DeepLens	CreateDeviceCertificates	API call
AWS DeepRacer	AdminGetAccountConfig	API call
AWS DeepRacer	AdminListAssociatedUsers	API call
AWS DeepRacer	TestRewardFunction	API call
AWS DeepRacer	VerifyResourcesExistForTagris	API call
Amazon Detective	BatchGetGraphMembe rDatasources	API call
Amazon Detective	BatchGetMembership Datasources	API call
Amazon Detective	SearchGraph	API call
Amazon DevOps Guru	SearchInsights	API call
Amazon DevOps Guru	SearchOrganizationInsights	API call
AWS Database Migration Service	BatchStartRecommendations	API call

Service	Event name	Event type
AWS Database Migration Service	ModifyRecommendation	API call
AWS Database Migration Service	StartRecommendations	API call
AWS Database Migration Service	VerifyResourcesExistForTagris	API call
AWS Directory Service	VerifyTrust	API call
Amazon Elastic Compute Cloud	ConfirmProductInstance	API call
Amazon Elastic Compute Cloud	ReportInstanceStatus	API call
Amazon Elastic Container Registry	BatchCheckLayerAvailability	API call
Amazon Elastic Container Registry	BatchGetImage	API call
Amazon Elastic Container Registry	BatchGetImageReferrer	API call
Amazon Elastic Container Registry	BatchGetRepository ScanningConfiguration	API call
Amazon Elastic Container Registry	DryRunEvent	Service event
Amazon Elastic Container Registry	PolicyExecutionEvent	Service event
Amazon Elastic Container Registry Public	BatchCheckLayerAvailability	API call

Service	Event name	Event type
Amazon Elastic Container Service	DiscoverPollEndpoint	API call
Amazon Elastic Container Service	FindSubfleetRoute	API call
Amazon Elastic Container Service	ValidateResources	API call
Amazon Elastic Container Service	VerifyTaskSetsExist	API call
Amazon Elastic Kubernetes Service	AccessKubernetesApi	API call
AWS Elastic Beanstalk	CheckDNSAvailability	API call
AWS Elastic Beanstalk	RequestEnvironmentInfo	API call
AWS Elastic Beanstalk	RetrieveEnvironmentInfo	API call
AWS Elastic Beanstalk	ValidateConfigurationSettings	API call
Amazon Elastic File System	NewClientConnection	Service event
Amazon Elastic File System	UpdateClientConnection	Service event
Amazon Elastic Transcoder	ReadJob	API call
Amazon Elastic Transcoder	ReadPipeline	API call
Amazon Elastic Transcoder	ReadPreset	API call
Amazon EventBridge	TestEventPattern	API call
Amazon EventBridge	TestScheduleExpression	API call
Amazon FinSpace API	BatchListCatalogNodesByDataset	API call

Service	Event name	Event type
Amazon FinSpace API	BatchListNodesByDataset	API call
Amazon FinSpace API	BatchValidateAccess	API call
Amazon FinSpace API	CreateAuditRecordsQuery	API call
Amazon FinSpace API	SearchDatasets	API call
Amazon FinSpace API	SearchDatasetsV	API call
Amazon FinSpace API	ValidateIdToken	API call
AWS Firewall Manager	DisassociateAdminAccount	API call
Amazon Forecast	InvokeForecastEndpoint	API call
Amazon Forecast	QueryFeature	API call
Amazon Forecast	QueryForecast	API call
Amazon Forecast	QueryWhatIfForecast	API call
Amazon Forecast	VerifyResourcesExistForTagris	API call
Amazon Fraud Detector	BatchGetVariable	API call
Amazon Fraud Detector	VerifyResourcesExistForTagris	API call
FreeRTOS	VerifyEmailAddress	API call
Amazon GameLift Servers	RequestUploadCredentials	API call
Amazon GameLift Servers	ResolveAlias	API call
Amazon GameLift Servers	SearchGameSessions	API call
Amazon GameLift Servers	ValidateMatchmakingRuleSet	API call
Amazon GameSparks	ExportSnapshot	API call

Service	Event name	Event type
Amazon Location Service	BatchGetDevicePosition	API call
Amazon Location Service	CalculateRoute	API call
Amazon Location Service	CalculateRouteMatrix	API call
Amazon Location Service	SearchPlaceIndexForPosition	API call
Amazon Location Service	SearchPlaceIndexForSuggestions	API call
Amazon Location Service	SearchPlaceIndexForText	API call
Amazon Glacier	InitiateJob	API call
AWS Glue	BatchGetBlueprints	API call
AWS Glue	BatchGetColumnStatisticsForTable	API call
AWS Glue	BatchGetCrawlers	API call
AWS Glue	BatchGetCustomEntityTypes	API call
AWS Glue	BatchGetDataQualityResult	API call
AWS Glue	BatchGetDevEndpoints	API call
AWS Glue	BatchGetJobs	API call
AWS Glue	BatchGetMLTransform	API call
AWS Glue	BatchGetPartition	API call
AWS Glue	BatchGetTriggers	API call
AWS Glue	BatchGetWorkflows	API call
AWS Glue	QueryJobRuns	API call

Service	Event name	Event type
AWS Glue	QueryJobRunsAggregated	API call
AWS Glue	QueryJobs	API call
AWS Glue	QuerySchemaVersion Metadata	API call
AWS Glue	SearchTables	API call
AWS HealthLake	ReadResource	API call
AWS HealthLake	SearchWithGet	API call
AWS HealthLake	SearchWithPost	API call
AWS Identity and Access Management	GenerateCredentialReport	API call
AWS Identity and Access Management	GenerateOrganizationsAccess Report	API call
AWS Identity and Access Management	GenerateServiceLastAccessed Details	API call
AWS Identity and Access Management	SimulateCustomPolicy	API call
AWS Identity and Access Management	SimulatePrincipalPolicy	API call
AWS Identity Store	IsMemberInGroups	API call
AWS Identity Store Auth	BatchGetSession	API call
Amazon Inspector Classic	PreviewAgents	API call
Amazon Inspector Classic	BatchGetAccountStatus	API call
Amazon Inspector Classic	BatchGetFreeTrialInfo	API call

Service	Event name	Event type
Amazon Inspector Classic	BatchGetMember	API call
AWS Invoicing	ValidateDocumentDeliveryS3LocationInfo	API call
AWS IoT	SearchIndex	API call
AWS IoT	TestAuthorization	API call
AWS IoT	TestInvokeAuthorizer	API call
AWS IoT	ValidateSecurityProfileBehaviors	API call
AWS IoT Analytics	SampleChannelData	API call
AWS IoT SiteWise	GatewaysVerifyResourcesExistForTagrisInternal	API call
AWS IoT Things Graph	SearchEntities	API call
AWS IoT Things Graph	SearchFlowExecutions	API call
AWS IoT Things Graph	SearchFlowTemplates	API call
AWS IoT Things Graph	SearchSystemInstances	API call
AWS IoT Things Graph	SearchSystemTemplates	API call
AWS IoT Things Graph	SearchThings	API call
AWS IoT TwinMaker	ExecuteQuery	API call
AWS IoT Wireless	CreateNetworkAnalyzerConfiguration	API call
AWS IoT Wireless	DeleteNetworkAnalyzerConfiguration	API call

Service	Event name	Event type
AWS IoT Wireless	DeregisterWirelessDevice	API call
Amazon Interactive Video Service	BatchGetChannel	API call
Amazon Interactive Video Service	BatchGetStreamKey	API call
Amazon Kendra	BatchGetDocumentStatus	API call
Amazon Kendra	Query	API call
Amazon Managed Service for Apache Flink	DiscoverInputSchema	API call
AWS Key Management Service	Decrypt	API call
AWS Key Management Service	Encrypt	API call
AWS Key Management Service	GenerateDataKey	API call
AWS Key Management Service	GenerateDataKeyPair	API call
AWS Key Management Service	GenerateDataKeyPairWithoutPlaintext	API call
AWS Key Management Service	GenerateDataKeyWithoutPlaintext	API call
AWS Key Management Service	GenerateMac	API call
AWS Key Management Service	GenerateRandom	API call

Service	Event name	Event type
AWS Key Management Service	ReEncrypt	API call
AWS Key Management Service	Sign	API call
AWS Key Management Service	Verify	API call
AWS Key Management Service	VerifyMac	API call
AWS Lake Formation	SearchDatabasesByLFTags	API call
AWS Lake Formation	SearchTablesByLFTags	API call
AWS Lake Formation	StartQueryPlanning	API call
Amazon Lex	BatchCreateCustomVocabularyItem	API call
Amazon Lex	BatchDeleteCustomVocabularyItem	API call
Amazon Lex	BatchUpdateCustomVocabularyItem	API call
Amazon Lex	DeleteCustomVocabulary	API call
Amazon Lex	SearchAssociatedTranscripts	API call
Amazon Lightsail	CreateGUISessionAccessDetails	API call
Amazon Lightsail	DownloadDefaultKeyPair	API call
Amazon Lightsail	IsVpcPeered	API call
Amazon CloudWatch Logs	FilterLogEvents	API call

Service	Event name	Event type
Amazon Macie	BatchGetCustomDataIdentifiers	API call
Amazon Macie	UpdateFindingsFilter	API call
AWS Elemental MediaConnect	ManagedDescribeFlow	API call
AWS Elemental MediaConnect	PrivateDescribeFlowMeta	API call
AWS Transform MGN	OperationalDescribeJobLogItems	API call
AWS Transform MGN	OperationalDescribeJobs	API call
AWS Transform MGN	OperationalDescribeReplicationConfigurationTemplates	API call
AWS Transform MGN	OperationalDescribeSourceServer	API call
AWS Transform MGN	OperationalGetLaunchConfiguration	API call
AWS Transform MGN	OperationalListSourceServers	API call
AWS Transform MGN	VerifyClientRoleForMgn	API call
AWS HealthOmics	VerifyResourceExists	API call
AWS HealthOmics	VerifyResourcesExistForTagris	API call
Amazon Polly	SynthesizeLongSpeech	API call
Amazon Polly	SynthesizeSpeech	API call
Amazon Polly	SynthesizeSpeechGet	API call
AWS service providing managed private networks	Ping	API call

Service	Event name	Event type
AWS Proton	DeleteEnvironmentTemplateVersion	API call
AWS Proton	DeleteServiceTemplateVersion	API call
Amazon QLDB	ShowCatalog	API call
Amazon Quick	GenerateEmbedUrlForAnonymousUser	API call
Amazon Quick	GenerateEmbedUrlForRegisteredUser	API call
Amazon Quick	QueryDatabase	Service event
Amazon Quick	SearchAnalyses	API call
Amazon Quick	SearchDashboards	API call
Amazon Quick	SearchDataSets	API call
Amazon Quick	SearchDataSources	API call
Amazon Quick	SearchFolders	API call
Amazon Quick	SearchGroups	API call
Amazon Quick	SearchUsers	API call
Amazon Relational Database Service	DownloadCompleteDBLogFile	API call
Amazon Relational Database Service	DownloadDBLogFilePortion	API call
Amazon Rekognition	CompareFaces	API call
Amazon Rekognition	DetectCustomLabels	API call

Service	Event name	Event type
Amazon Rekognition	DetectFaces	API call
Amazon Rekognition	DetectLabels	API call
Amazon Rekognition	DetectModerationLabels	API call
Amazon Rekognition	DetectProtectiveEquipment	API call
Amazon Rekognition	DetectText	API call
Amazon Rekognition	RecognizeCelebrities	API call
Amazon Rekognition	SearchFaces	API call
Amazon Rekognition	SearchFacesByImage	API call
Amazon Rekognition	SearchUsers	API call
Amazon Rekognition	SearchUsersByImage	API call
AWS Resource Explorer	BatchGetView	API call
AWS Resource Explorer	Search	API call
AWS Resource Groups	SearchResources	API call
AWS Resource Groups	ValidateResourceSharing	API call
AWS RoboMaker	BatchDescribeSimulationJob	API call
Amazon Route 53	TestDNSAnswer	API call
Amazon Route 53 Domains	checkAvailabilities	API call
Amazon Route 53 Domains	CheckDomainAvailability	API call
Amazon Route 53 Domains	checkDomainTransferability	API call
Amazon Route 53 Domains	CheckDomainTransferability	API call

Service	Event name	Event type
Amazon Route 53 Domains	isEmailReachable	API call
Amazon Route 53 Domains	searchDomains	API call
Amazon Route 53 Domains	sendVerificationMessage	API call
Amazon Route 53 Domains	ViewBilling	API call
Amazon Route 53 Domains	viewBilling	API call
Amazon CloudWatch RUM	BatchGetRumMetricDefinitions	API call
Amazon Simple Storage Service	echo	API call
Amazon Simple Storage Service	GenerateInventory	Service event
Amazon SageMaker AI	BatchDescribeModelPackage	API call
Amazon SageMaker AI	DeleteModelCard	API call
Amazon SageMaker AI	QueryLineage	API call
Amazon SageMaker AI	RenderUiTemplate	API call
Amazon SageMaker AI	Search	API call
Amazon EventBridge Schemas	ExportSchema	API call
Amazon EventBridge Schemas	SearchSchemas	API call
Amazon SimpleDB	DomainMetadata	API call
AWS Secrets Manager	ValidateResourcePolicy	API call

Service	Event name	Event type
AWS Service Catalog	ScanProvisionedProducts	API call
AWS Service Catalog	SearchProducts	API call
AWS Service Catalog	SearchProductsAsAdmin	API call
AWS Service Catalog	SearchProvisionedProducts	API call
Amazon SES	BatchGetMetricData	API call
Amazon SES	TestRenderEmailTemplate	API call
Amazon SES	TestRenderTemplate	API call
Amazon Simple Notification Service	CheckIfPhoneNumber IsOptedOut	API call
AWS SQL Workbench	BatchGetNotebookCell	API call
AWS SQL Workbench	ExportNotebook	API call
Amazon EC2 Systems Manager	ExecuteApi	API call
AWS Systems Manager Incident Manager	DeleteContactChannel	API call
AWS IAM Identity Center	IsMemberInGroup	API call
AWS IAM Identity Center	SearchGroups	API call
AWS IAM Identity Center	SearchUsers	API call
AWS STS	AssumeRole	API call
AWS STS	AssumeRoleWithSAML	API call
AWS STS	AssumeRoleWithWebIdentity	API call
AWS STS	DecodeAuthorizationMessage	API call

Service	Event name	Event type
AWS Tax Settings	BatchGetTaxExemptions	API call
AWS WAFV2	CheckCapacity	API call
AWS WAFV2	GenerateMobileSdkReleaseUrl	API call
AWS Well-Architected Tool	ExportLens	API call
AWS Well-Architected Tool	TagResource	API call
AWS Well-Architected Tool	UntagResource	API call
AWS Well-Architected Tool	UpdateGlobalSettings	API call
Amazon Connect Customer Wisdom	QueryAssistant	API call
Amazon Connect Customer Wisdom	SearchContent	API call
Amazon Connect Customer Wisdom	SearchSessions	API call
Amazon WorkDocs	AbortDocumentVersionUpload	API call
Amazon WorkDocs	AddUsersToGroup	API call
Amazon WorkDocs	BatchGetUsers	API call
Amazon WorkDocs	CheckAlias	API call
Amazon WorkDocs	CompleteDocumentVersionUpload	API call
Amazon WorkDocs	CreateAnnotation	API call
Amazon WorkDocs	CreateComment	API call

Service	Event name	Event type
Amazon WorkDocs	CreateFeedbackRequest	API call
Amazon WorkDocs	CreateFolder	API call
Amazon WorkDocs	CreateGroup	API call
Amazon WorkDocs	CreateShare	API call
Amazon WorkDocs	CreateUser	API call
Amazon WorkDocs	DeleteAnnotation	API call
Amazon WorkDocs	DeleteComment	API call
Amazon WorkDocs	DeleteDocument	API call
Amazon WorkDocs	DeleteFeedbackRequest	API call
Amazon WorkDocs	DeleteFolder	API call
Amazon WorkDocs	DeleteFolderContents	API call
Amazon WorkDocs	DeleteGroup	API call
Amazon WorkDocs	DeleteOrganizationShare	API call
Amazon WorkDocs	DeleteUser	API call
Amazon WorkDocs	DownloadDocumentVersion	API call
Amazon WorkDocs	DownloadDocumentVersionUnderlays	API call
Amazon WorkDocs	InitiateDocumentVersionUpload	API call
Amazon WorkDocs	LogoutUser	API call
Amazon WorkDocs	PaginatedOrganizationActivity	API call

Service	Event name	Event type
Amazon WorkDocs	PublishAnnotations	API call
Amazon WorkDocs	PublishComments	API call
Amazon WorkDocs	RestoreDocument	API call
Amazon WorkDocs	RestoreFolder	API call
Amazon WorkDocs	SearchGroups	API call
Amazon WorkDocs	SearchOrganizationUsers	API call
Amazon WorkDocs	TransferUserResources	API call
Amazon WorkDocs	UpdateAnnotation	API call
Amazon WorkDocs	UpdateComment	API call
Amazon WorkDocs	UpdateDocument	API call
Amazon WorkDocs	UpdateDocumentVersion	API call
Amazon WorkDocs	UpdateFolder	API call
Amazon WorkDocs	UpdateGroup	API call
Amazon WorkDocs	UpdateOrganization	API call
Amazon WorkDocs	UpdateUser	API call
Amazon WorkMail	AssumeImpersonationRole	API call
Amazon WorkMail	QueryDnsRecords	API call
Amazon WorkMail	SearchMembers	API call
Amazon WorkMail	TestAvailabilityConfiguration	API call
Amazon WorkMail	TestInboundMailFlowRules	API call

Service	Event name	Event type
Amazon WorkMail	TestOutboundMailFlowRules	API call

Amazon EventBridge events detail reference

EventBridge itself emits the following events. These events are automatically sent to the default event bus as with any other AWS service.

For definitions of the metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

Event detail type	Description
Scheduled Event	Represents a scheduled event.
Schema Created	Represents the creation of a new event schema.
Schema Version Created	Represents the creation of a new version of a new or existing event schema.
Connection state events	Represents a change in the state of a connection.
API Destination state events	Represents a change in the state of an API destination.

Schedule events

EventBridge sends the following schedule events to the default event bus. For more information, see Scheduler.

Scheduled Event

Represents a scheduled event.

The `source` and `detail-type` fields are included because they contain specific values for EventBridge events. For definitions of the other metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

```
{
```

```
  . . .,
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  . . .,
  "detail": {}
}
```

`detail-type`

Identifies the type of event.

For this event, this value is `Scheduled Event`.

Required: Yes

`source`

Identifies the service that generated the event. For EventBridge events, this value is `aws.events`.

Required: Yes

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

Required: Yes

There are no required fields in this object for `Scheduled Event` events.

Example Scheduled Event event

```
{
  "version": "0",
  "id": "89d1a02d-5ec7-412e-82f5-13505f849b41",
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  "account": "123456789012",
  "time": "2016-12-30T18:44:49Z",
  "region": "us-east-1",
  "resources": ["arn:aws:events:us-east-1:123456789012:rule/SampleRule"],
  "detail": {}
}
```

Schema registry events

EventBridge sends the following schema registry events to the default event bus. For more information, see ???.

Schema Created

Represents the creation of a new schema.

When a schema is created, EventBridge sends both a `Schema Created` and a `Schema Version Created` event.

The `source` and `detail-type` fields are included because they contain specific values for EventBridge events. For definitions of the other metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

```
{
  . . .,
  "detail-type": "Schema Created",
  "source": "aws.schemas",
  . . .,
  "detail": {
    "SchemaName" : "String",
    "SchemaType" : "String",
    "RegistryName" : "String",
    "CreationDate" : "DateTime",
    "Version" : "Number"
  }
}
```

`detail-type`

Identifies the type of event.

For this event, this value is `Schema Created`.

Required: Yes

`source`

Identifies the service that generated the event. For EventBridge events, this value is `aws.schemas`.

Required: Yes

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

Required: Yes

For this event, this data includes:

`SchemaName`

The name of the schema.

Required: Yes

`SchemaType`

The type of schema.

Valid values: `OpenApi3` | `JSONSchemaDraft4`

Required: Yes

`RegistryName`

The name of the registry that contains the schema.

Required: Yes

`CreationDate`

The date the schema was created.

Required: Yes

`Version`

The version of the schema.

For `Schema Created` events, this value will always be 1.

Required: Yes

Example Schema Created event

```
{
  "version": "0",
  "id": "01234567-0123-0123-0123-012345678901",
```

```
  "detail-type": "Schema Created",
  "source": "aws.schemas",
  "account": "123456789012",
  "time": "2019-05-31T21:49:54Z",
  "region": "us-east-1",
  "resources": ["arn:aws:schemas:us-east-1::schema/myRegistry/mySchema"],
  "detail": {
    "SchemaName": "mySchema",
    "SchemaType": "OpenApi3",
    "RegistryName": "myRegistry",
    "CreationDate": "2019-11-29T20:08:55Z",
    "Version": "1"
  }
}
```

Schema Version Created

Represents the creation of a new version of a new or existing event schema.

When a schema is created, EventBridge sends both a `Schema Created` and a `Schema Version Created` event.

The `source` and `detail-type` fields are included because they contain specific values for EventBridge events. For definitions of the other metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

```
{
  . . .,
  "detail-type": "Schema Version Created",
  "source": "aws.schemas",
  . . .,
  "detail": {
    "SchemaName" : "String",
    "SchemaType" : "String",
    "RegistryName" : "String",
    "CreationDate" : "DateTime",
    "Version" : "Number"
  }
}
```

`detail-type`

Identifies the type of event.

For this event, this value is `Schema Version Created`.

Required: Yes

`source`

Identifies the service that generated the event. For EventBridge events, this value is `aws.schemas`.

Required: Yes

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

Required: Yes

For this event, this data includes:

`SchemaName`

The name of the schema.

Required: Yes

`SchemaType`

The type of schema.

Valid values: `OpenApi3` | `JSONSchemaDraft4`

Required: Yes

`RegistryName`

The name of the registry that contains the schema.

Required: Yes

`CreationDate`

The date the schema version was created.

Required: Yes

`Version`

The version of the schema.

Required: Yes

Example Schema Version Created event

```
{
  "version": "0",
  "id": "01234567-0123-0123-0123-012345678901",
  "detail-type": "Schema Version Created",
  "source": "aws.schemas",
  "account": "123456789012",
  "time": "2019-05-31T21:49:54Z",
  "region": "us-east-1",
  "resources": ["arn:aws:schemas:us-east-1::schema/myRegistry/mySchema"],
  "detail": {
    "SchemaName": "mySchema",
    "SchemaType": "OpenApi3",
    "RegistryName": "myRegistry",
    "CreationDate": "2019-11-29T20:08:55Z",
    "Version": "5"
  }
}
```

Connection events

EventBridge sends the following connection events to the default event bus. For more information, see Connections.

Connection state events

These events each represent a change in the state of a new or existing connection.

The `source` and `detail-type` fields are included because they contain specific values for EventBridge events. For definitions of the other metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

```
{
  . . .,
  "detail-type": "Connection status",
  "source": "aws.events",
  . . .,
  "detail": {
    "ConnectionName" : "String",
```

```
    "StateReason" : "String",
    "Timestamp" : "DateTime"
  }
}
```

`detail-type`

Identifies the type of event.

For this event, this value is one of the following:

- `Connection Creation Started`
- `Connection Update Started`
- `Connection Deletion Started`
- `Connection Activated`
- `Connection Authorized`
- `Connection Authorization Started`
- `Connection Deauthorization Started`
- `Connection Deauthorized`
- `Connection Failed Connectivity`

Required: Yes

`source`

Identifies the service that generated the event. For EventBridge events, this value is `aws.events`.

Required: Yes

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

Required: Yes

For this event, this data includes:

`ConnectionName`

The name of the connection.

Required: Yes

`StateReason`

The reason the connection state has changed.

Required: No

`Timestamp`

The time and date the connection state changed.

Required: Yes

Example Connection state event

```
{
    "version": "0",
    "id": "1d7a4ac6-a50a-745f-a331-a0d802f7badb",
    "detail-type": "Connection Creation Started",
    "source": "aws.events",
    "account": "123456789012",
    "time": "2024-10-28T09:08:20Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:events:us-east-1:123456789012:connection/sample-connection/
ee7e4d52-8df0-4bed-a0d5-fa7dea43fcf8"
    ],
    "detail": {
        "ConnectionName": "sample-connection",
        "Timestamp": "2024-10-24 09:26:35 +0000 UTC"
    }
}
```

API Destination events

EventBridge sends the following API destination events to the default event bus. For more information, see API destinations.

API Destination state events

These events each represents a change in the state of an API destination.

The `source` and `detail-type` fields are included because they contain specific values for EventBridge events. For definitions of the other metadata fields that are included in all events, see AWS service event metadata in the *Events Reference*.

```
{
  . . .,
  "detail-type": "API Destination status",
  "source": "aws.events",
  . . .,
  "detail": {
    "ApiDestinationName" : "String",
    "Timestamp" : "DateTime"
  }
}
```

`detail-type`

Identifies the type of event.

For this event, this value is one of the following:

- `API Destination Activated`
- `API Destination Deactivated`

Required: Yes

`source`

Identifies the service that generated the event. For EventBridge events, this value is `aws.events`.

Required: Yes

`detail`

A JSON object that contains information about the event. The service generating the event determines the content of this field.

Required: Yes

For this event, this data includes:

`ApiDestinationName`

The name of the API destination.

Required: Yes

`Timestamp`

The time and date the API destination state changed.

Required: Yes

Example API destination state event

```
{
    "version": "0",
    "id": "1d7a4ac6-a50a-745f-a331-a0d802f7badb",
    "detail-type": "API Destination Deactivated",
    "source": "aws.events",
    "account": "123456789012",
    "time": "2024-10-28T09:08:20Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:events:us-east-1:123456789012:api-destination/sample-api-destination/
ee7e4d52-8df0-4bed-a0d5-fa7dea43fcf8"
    ],
    "detail": {
        "ApiDestinationName": "sample-api-destination",
        "Timestamp": "2024-10-24 09:26:35 +0000 UTC"
    }
}
```

Rules in Amazon EventBridge

You specify what EventBridge does with the events delivered to each event bus. To do this, you create *rules*. A rule specifies which events to send to which targets for processing. A single rule can send an event to multiple targets, which then run in parallel.

You can create two types of rules: rules that match on event data as events are delivered, and rules that run on a defined schedule. In addition, certain AWS services may create and manage rules in your account as well.

Rules that match on event data

You can create rules that match against incoming events based on event data criteria (called an *event pattern*). An event pattern defines the event structure and the fields that a rule matches. If

an event matches the criteria defined in the event pattern, EventBridge sends it to the target(s) you specify.

EventBridge provides two rule builder modes for creating event-pattern rules in the console:

- **Enhanced Builder** – A visual canvas for building rules and targets. Drag and drop events and targets to construct your rule without writing JSON directly. For more information, see Creating rules (Enhanced Builder).
- **Advanced Builder** – JSON patterns and configuration for rules and targets. Provides direct access to the full event pattern syntax and target configuration options. For more information, see Creating rules (Advanced Builder).

Rules that run on a schedule

> **Note**
>
> Scheduled rules are a legacy feature of EventBridge.
> EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.
> Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
> For more information, see ??? or the *EventBridge Scheduler User Guide*.

You can also create rules that sends events to the specified targets at specified intervals. For example, to periodically run an Lambda function, you can create a rule to run on a schedule.

For more information, see ???.

Rules managed by AWS services

In addition to the rules you create, AWS services can create and manage EventBridge rules in your AWS account that are needed for certain functions in those services. These are called *managed rules*.

When a service creates a managed rule, it can also create an IAM policy that grants permission to that service to create the rule. IAM policies created this way are scoped narrowly with resource-level permissions to allow the creation of only the necessary rules.

You can delete managed rules by using the **Force delete** option, but you should only delete them if you're sure that the other service no longer needs the rule. Otherwise, deleting a managed rule causes the features that rely on it to stop working.

Creating rules using the Enhanced Builder

Prerequisites

Before you begin, ensure you have:

- Access to the Amazon EventBridge console
- Appropriate IAM permissions to create EventBridge rules
- Basic understanding of event-driven architectures

Overview

To take action on events received by EventBridge, you can create rules. When an event matches the event pattern defined in your rule, EventBridge sends the event to the specified target.

This topic walks you through creating a rule that EventBridge uses to match events as they are sent to the specified event bus.

Creating EventBridge rules involves four main steps:

1. Choose the events you want to process
2. Configure event filtering and testing
3. Select and configure targets
4. Configure the rule settings

Choose the events

First, choose the events you want to send to the target.

To select an event

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**, and then choose **Create rule**.
3. On the rule builder selection page, choose **Enhanced Builder**.

The Enhanced Builder provides a visual canvas for building rules and targets by dragging and dropping events and targets. To use JSON patterns and configuration instead, see Creating rules (Advanced Builder).

4. In the **Events** tab, find the events you want to send to the target:

   a. Choose **AWS Service Events** or **Custom Events**.

   b. Use the Search box or browse the event list to find the events you want to send to the target.

   c. Drag one or more events into the **Build** canvas and drop it on **Triggering Events**.

   EventBridge displays the **Triggering Events** section. This section includes:

   - **Events**, which lists the event you chose.
   - **Schema**, which displays the schema of the selected event, if available.

     Schema contains three tabs:

     - **Tree**: A tree view of the event schema.
     - **Code**: The event schema in Open API spec or JSON format.
     - **Info**: Overview information about the schema.
   - **Sample event**, which displays a sample of a selected event, if available.
   - **Event pattern (filter)**, which contains an event pattern that selects all events you've chosen.

Filter events to send only what you want

You probably won't want to send *all* events of a specific type to a target. You can make your event pattern more specific so that it only selects events that contain the attributes and values you're interested in.

Edit the event pattern (optional)

In the **Triggering events** pane, you can build the event pattern two ways:

- Visually, using the **Schema** tree view
- By directly editing the JSON in the **Event pattern** pane

To edit the event pattern using the Schema tree view

1. Choose the **Tree** tab of the **Schema** pane.
2. Add and edit filters to build out the event pattern.

   a. Choose the filter icon next to the attribute you want to add as a filter to the event pattern.

   b. Choose a comparison operator from the drop-down list.

      For more information on comparison operators, see Creating event patterns.

   c. Enter the value you want to match.

   d. Click the check mark to finish.

   EventBridge adds the filter you've created to the JSON in the **Event pattern** pane.
3. Add additional filters until you've constructed an event pattern that matches on all the event attributes you want.

Test the event pattern

After you construct an event pattern that matches all the event attributes you want, test to ensure the event pattern performs as expected.

Test the event pattern (optional)

1. Choose a sample event for testing, or use your own.

   - **Use sample events provided**
     1. Choose **Use sample events provided**.
     2. Under **Sample triggering event**, select the event to use for testing from the drop-down list.
   - **Use your own event**
     1. Choose **Use your own event**.

        EventBridge displays a generic event with an empty `details` element.
     2. Edit the event JSON to create the event against which you want to test your event pattern.

> **Tip**
>
> You can use the sample events provided as starting points for creating your own custom events to use in your testing:
>
> a. Choose **Use sample events provided**, choose an event from the **Sample triggering event**, and then choose **Copy**.
>
> b. Choose **Use your own event**.
>
> c. Select the entire event in the window, and paste the sample event over it.
>
> d. Edit the event as desired.

2. Choose **Run Test**.

   EventBridge displays a message stating whether the test event matches the event pattern.

Select targets

Drag one (and up to five) targets into the **Build** canvas and drop it on **Targets** to receive events that match the specified event pattern. Targets can include:

- Other EventBridge event buses, in the same or a different AWS account
- EventBridge API destinations, including SaaS partners such as Salesforce
- A range of AWS service resources, such as Amazon SQS queues or Amazon SNS topics

To select targets

1. Select the target type from the **Targets** pane, and drop it onto the **Targets** shape in the Build canvas.

   EventBridge displays the **Target** section in the Build, with the applicable configuration options based on the type of target you selected.

> **Tip**
>
> If you have specified more than one target, selecting the target shape displays the configuration section for that target.

2. Configure the target based on your target type. For detailed configuration steps, see EventBridge Targets.

Customize the event data sent to the target

If you select a target other than an EventBridge event bus or EventBridge API destination, you have the option of customizing what data actually gets delivered to the target.

Transform or replace the event data sent (optional)

1. In the Build design canvas, choose the **Input transformation** icon in the **Targets** shape.

   EventBridge displays the **Input transformation** section.
2. Under **Input transformation configuration**, choose **Enable**.
3. Select the Transformation method to choose how you want to customize the text sent to the target for matching events:

   - **Part of the matched events** – EventBridge only sends the specified portion of the original source event to the target.
   - **Constant (JSON text)** – EventBridge sends only the specified JSON text to the target. No part of the original source event is sent.
   - **Input transformer** – Configure an input transformer to customize the text you want EventBridge to send to the target. For more information, see Transforming target input.

Configure the rule

Finally, configure and create the rule.

To configure the rule (console)

1. Choose **Configure**.
2. Enter a **Name** and, optionally, a **Description** for the rule.

   A rule can't have the same name as another rule in the same AWS Region and on the same event bus.
3. For **Event bus**, choose the event bus to associate with this rule.

If you want this rule to match events that come from your account, select **AWS default event bus**. When an AWS service in your account sends an event, it always goes to your account's default event bus.

4. To enable the rule as soon as it is created, under **Activation** enable **Active**.
5. (Optional) Enter one or more tags for the rule. For more information, see Tagging resources in Amazon EventBridge.
6. Choose **Create**.

Next steps

After creating your rule, you can:

- Monitor rule performance using metrics
- Test your rule with sample events
- Modify event patterns as needed
- Add additional targets to existing rules

Additional resources

For more information about EventBridge rules, see:

- EventBridge rules overview
- Creating event patterns
- EventBridge targets
- Troubleshooting EventBridge rules

Creating rules using the Advanced Builder

To take action on events received by Amazon EventBridge, you can create rules. When an event matches the event pattern defined in your rule, EventBridge sends the event to the specified target and triggers the action defined in the rule.

Use the following procedure to create an Amazon EventBridge rule that responds to events.

The following steps walk you through how to create a rule that EventBridge uses to match events as they are sent to the specified event bus.

Steps

- Define the rule
- Build the event pattern
- Select targets
- Configure tags and review rule

Define the rule

First, enter a name and description for your rule to identify it. You must also define the event bus where your rule looks for events to match to an event pattern.

To define the rule detail

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. On the rule builder selection page, choose **Advanced Builder**.

   The Advanced Builder uses JSON patterns and configuration for rules and targets. To use the visual canvas instead, see Creating rules (Enhanced Builder).
5. Enter a **Name** and, optionally, a **Description** for the rule.

   A rule can't have the same name as another rule in the same AWS Region and on the same event bus.
6. For **Event bus**, choose the event bus to associate with this rule. If you want this rule to match events that come from your account, select **AWS default event bus**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
7. For **Rule type**, choose **Rule with an event pattern**.
8. Choose **Next**.

Build the event pattern

Next, build the event pattern. To do this, specify the event source, choose the basis for the event pattern, and define the attributes and values to match on. You can also generate the event pattern in JSON and test it against a sample event.

To build the event pattern

1. For **Event source**, choose **AWS events or EventBridge partner events**.
2. (Optional) In the **Sample events** section, choose a **Sample event type** against which you want to test your event pattern.

   The following sample event types are available:

   - **AWS events** – Select from events emitted from supported AWS services.
   - **EventBridge partner events** – Select from events emitted from third-party services that support EventBridge, such as Salesforce.
   - **Enter my own** – Enter your own event in JSON text.

     You can also use an AWS or partner event as the starting point for creating your own custom event.

     1. Select **AWS events** or **EventBridge partner events**.
     2. Use the **Sample events** dropdown to select the event you want to use as a starting point for your custom event.

        EventBridge displays the sample event.
     3. Select **Copy**.
     4. Select **Enter my own** for **Event type.**
     5. Delete the sample event structure in the JSON editing pane, and paste the AWS or partner event in its place.
     6. Edit the event JSON to create your own sample event.
3. Choose a **Creation method**. You can create an event pattern from an EventBridge schema or template, or you can create a custom event pattern.

   Existing schema

   To use an existing EventBridge schema to create the event pattern, do the following:

1. In the **Creation method** section, for **Method**, select **Use schema**.
2. In the **Event pattern** section, for **Schema type**, select **Select schema from Schema registry**.
3. For **Schema registry**, choose the dropdown box and enter the name of a schema registry, such as `aws.events`. You can also select an option from the dropdown list that appears.
4. For **Schema**, choose the dropdown box and enter the name of the schema to use. For example, `aws.s3@ObjectDeleted`. You can also select an option from the dropdown list that appears.
5. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

6. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
7. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Custom schema

To write a custom schema and convert it to an event pattern, do the following:

1. In the **Creation method** section, for **Method**, choose **Use schema**.
2. In the **Event pattern** section, for **Schema type**, choose **Enter schema**.
3. Enter your schema into the text box. You must format the schema as valid JSON text.
4. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

5. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
6. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Event pattern

To write a custom event pattern in JSON format, do the following:

1. In the **Creation method** section, for **Method**, choose **Custom pattern (JSON editor)**.
2. For **Event pattern**, enter your custom event pattern in JSON-formatted text.
3. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

You can also choose any of the following options:

- **Copy** – Copy the event pattern to your device's clipboard.
- **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.
- **Event pattern form** – Opens the event pattern in Pattern Builder. If the pattern can't be rendered in Pattern Builder as-is, EventBridge warns you before it opens Pattern Builder.

4. Choose **Next**.

Select targets

Choose one or more targets to receive events that match the specified pattern. Targets can include an EventBridge event bus, EventBridge API destinations, including SaaS partners such as Salesforce, or another AWS service.

To select targets

1. For **Target type**, choose one of the following target types:

   Event bus

   1. Select **EventBridge event bus**.
   2. Choose the event bus to use as the target.
      - To use an event bus in the same AWS Region as this rule:
        a. Select **Event bus in the same account and Region**.
        b. For **Event bus for target**, choose the dropdown box and enter the name of the event bus. You can also select the event bus from the dropdown list.

           For more information, see ???.
      - To use an event bus in a different AWS Region or account as this rule:
        a. Select **Event bus in a different account or Region**.
        b. For **Event bus as target**, enter the ARN of the event bus you want to use.

           For more information, see:
           - ???
           - ???

3. For many target types, EventBridge needs permissions to send events to the target. In these cases, EventBridge can create the IAM role needed for your rule to run.

   For **Execution role**, do one of the following:

   - To create a new execution role for this rule:
     a. Select **Create a new role for this specific resource**.
     b. Either enter a name for this execution role, or use the name generated by EventBridge.
   - To use an existing execution role for this rule:
     a. Select **Use existing role**.
     b. Enter or select the name of the execution role to use from the dropdown list.

4. (Optional) For **Additional settings**, specify any of the optional settings available for your target type:

   (Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:

   - Choose **None** to not use a dead-letter queue.
   - Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.
   - Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

     For more information, see Granting permissions to the dead-letter queue.

API destination

1. Select **EventBridge API destination**.
2. Choose a new or existing API destination:
   - To use an existing API destination, select **Use an existing API destination**. Then select an API destination from the dropdown list.
   - To create a new API destination, select **Create a new API destination**. Then, provide the following details for the destination:

- **Name** – Enter a name for the destination.

  Names must be unique within your AWS account. Names can have up to 64 characters. Valid characters are **A-Z**, **a-z**, **0-9**, and **. _ -** (hyphen).

- (Optional) **Description** – Enter a description for the destination.

  Descriptions can have up to 512 characters.

- **API destination endpoint** – The URL endpoint for the target.

  The endpoint URL must start with **`https`**. You can include the ***** as a path parameter wildcard. You can set path parameters from the target's `HttpParameters` attribute.

- **HTTP method** – Select the HTTP method used when you invoke the endpoint.

- (Optional) **Invocation rate limit per second** – Enter the maximum number of invocations accepted for each second for this destination.

  This value must be greater than zero. By default, this value is set to 300.

- **Connection** – Choose to use a new or existing connection:

  - To use an existing connection, select **Use an existing connection** and select the connection from the dropdown list.

  - To create a new connection for this destination select **Create a new connection**, then define the connection's **Name**, **Destination type**, and **Authorization type**. You can also add an optional **Description** for this connection.

3. For many target types, EventBridge needs permissions to send events to the target. In these cases, EventBridge can create the IAM role needed for your rule to run.

   For **Execution role**, do one of the following:

   - To create a new execution role for this rule:

     a. Select **Create a new role for this specific resource**.

     b. Either enter a name for this execution role, or use the name generated by EventBridge.

   - To use an existing execution role for this rule:

     a. Select **Use existing role**.

     b. Enter or select the name of the execution role to use from the dropdown list.

4. (Optional) For **Additional settings**, specify any of the optional settings available for your target type:

   Note that EventBridge may not display all of the following fields for a given AWS service.

   a. (Optional) For Configure target input, choose how you want to customize the text sent to the target for matching events. Choose one of the following:

      - **Matched events** – EventBridge sends the entire original source event to the target. This is the default.
      - **Part of the matched events** – EventBridge only sends the specified portion of the original source event to the target.

        Under **Specify the part of the matched event**, specify a JSON path that defines the part of the event you want EventBridge to send to the target.
      - **Constant (JSON text)** – EventBridge sends only the specified JSON text to the target. No part of the original source event is sent.

        Under **Specify the constant in JSON**, specify the JSON text that you want EventBridge to send to the target instead of the event.
      - **Input transformer** – Configure an input transformer to customize the text you want EventBridge send to the target. For more information, see ???.

        i. Select **Configure input transformer**.

        ii. Configure the input transformer following the steps in ???.

   b. (Optional) Under **Retry policy**, specify how EventBridge should retry sending an event to a target after an error occurs.

      - **Maximum age of event** – Enter the maximum amount of time (in hours, minutes, and seconds) for EventBridge to retain unprocessed events. The default is 24 hours.
      - **Retry attempts** – Enter the maximum number of times EventBridge should retry sending an event to the target after an error occurs. The default is 185 times.

   c. (Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:

      - Choose **None** to not use a dead-letter queue.
      - Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.

- Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

  For more information, see Granting permissions to the dead-letter queue.

For more information, see ???.

AWS service

1. Select **AWS service**.
2. For **Select a target**, select an AWS service to use as the target.
3. If you choose an AWS service that supports cross-account targets, you can select a target in the same account as the event bus, or a different account.
   - For a target in the same account, for **Target type** select **Target in this account**.
     a. Provide the information requested for the service you select.

     > **Note**
     >
     > The fields displayed vary depending on the service selected. For more information about available targets, see Event bus targets available in the EventBridge console.

     b. For many target types, EventBridge needs permissions to send events to the target. In these cases, EventBridge can create the IAM role needed for your rule to run.

     For **Execution role**, do one of the following:

     - To create a new execution role for this rule:
       i. Select **Create a new role for this specific resource**.
       ii. Either enter a name for this execution role, or use the name generated by EventBridge.
     - To use an existing execution role for this rule:
       i. Select **Use existing role**.
       ii. Enter or select the name of the execution role to use from the dropdown list.

> **Role manager enabled**
>
> If role manager is enabled in your account, EventBridge attaches the role for you. EventBridge replaces the **Create a new role for this specific resource** and **Use existing role** options with a **Customize** option. To use a different role, choose **Customize**. For more information, see IAM role creation in the *IAM User Guide*.

- For a target in a different account, for **Target type** select **Target in another AWS account**.

  a. Enter the ARN of the target resource to which you want to send events.

  b. Provide any additional information requested for the service you select.

  c. Select the name of the execution role to use from the dropdown list.

4. (Optional) For **Additional settings**, specify any of the optional settings available for your target type:

   a. (Optional) For Configure target input, choose how you want to customize the text sent to the target for matching events. Choose one of the following:

   - **Matched events** – EventBridge sends the entire original source event to the target. This is the default.
   - **Part of the matched events** – EventBridge only sends the specified portion of the original source event to the target.

     Under **Specify the part of the matched event**, specify a JSON path that defines the part of the event you want EventBridge to send to the target.
   - **Constant (JSON text)** – EventBridge sends only the specified JSON text to the target. No part of the original source event is sent.

     Under **Specify the constant in JSON**, specify the JSON text that you want EventBridge to send to the target instead of the event.
   - **Input transformer** – Configure an input transformer to customize the text you want EventBridge send to the target. For more information, see ???.

     i. Select **Configure input transformer**.

     ii. Configure the input transformer following the steps in ???.

b. (Optional) Under **Retry policy**, specify how EventBridge should retry sending an event to a target after an error occurs.

- **Maximum age of event** – Enter the maximum amount of time (in hours, minutes, and seconds) for EventBridge to retain unprocessed events. The default is 24 hours.
- **Retry attempts** – Enter the maximum number of times EventBridge should retry sending an event to the target after an error occurs. The default is 185 times.

c. (Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:

- Choose **None** to not use a dead-letter queue.
- Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.
- Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

  For more information, see Granting permissions to the dead-letter queue.

2. (Optional) Choose **Add another target** to add another target for this rule.
3. Choose **Next**.

Configure tags and review rule

Finally, enter any desired tags for the rule, then review and create the rule.

To configure tags, and review and create the rule

1. (Optional) Enter one or more tags for the rule. For more information, see Tagging resources in Amazon EventBridge.
2. Choose **Next**.
3. Review the details for the new rule. To make changes to any section, choose the **Edit** button next to that section.

   When satisfied with the rule details, choose **Create rule**.

Disabling or deleting a rule in Amazon EventBridge

To stop a rule from processing events or running on a schedule, you can delete or disable the rule. The following steps walk you through how to delete or disable an EventBridge rule.

To delete or disable a rule

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.

   Under **Event bus**, select the event bus that is associated with the rule.
3. Do one of the following:

   a. To delete a rule, select the button next to the rule and choose **Actions**, **Delete**, **Delete**.

      If the rule is a managed rule, enter the name of the rule to acknowledge that it is a managed rule and that deleting it may stop functionality in the service that created the rule. To continue, enter the rule name and choose **Force delete**.

   b. To temporarily disable a rule, select the button next to the rule and choose **Disable**, **Disable**.

      You can't disable a managed rule.

Best practices when defining rules in Amazon EventBridge

Below are some best practices to consider when you create rules for your event buses.

Set a single target for each rule

While you can specify up to five targets for a given rule, managing rules is easier when you specify a single target for each rule. If more than one target needs to receive the same set of events, we recommend duplicating the rule to deliver the same events to different targets. This encapsulation simplifies maintaining rules: if the needs of the event targets diverge over time, you can update each rule and its event pattern independently of the others.

Set rule permissions

You can enable event-consuming application components or services to be in control of managing their own rules. A common architectural approach adopted by customers is to isolate these application components or services by using separate AWS accounts. To enable the flow of events from one account to another, you must create a rule on one event bus that routes events to an event bus in another account. You can enable event-consuming teams or services to be in control of managing their own rules. You do this by specifying the appropriate permissions to their accounts through resource policies. This works across accounts and Regions.

For more information, see ???.

For example of resource policies, see Multi-account design patterns with Amazon EventBridge on GitHub.

Monitor rule performance

Monitor your rules to make sure they are performing as you expect:

- Monitor the `TriggeredRules` metric for missing data-points or anomalies can assist you in detecting discrepancies for a publisher that made a breaking change. For more information, see ???.
- Alarm on anomalies or maximum expected count can also help detecting when a rule is matching against new events. This can happen when event publishers, including AWS services and SaaS partners, introduce new events when enabling new use-cases and features. When these new events are unexpected and lead to a higher volume than the processing rate of the downstream target, they can lead to an event backlog.

  Such processing of unexpected events can also lead to unwanted billing charges.

  It can also trigger throttling of rules when the account goes over its aggregate target invocations per second service quota. EventBridge will still attempt to deliver events matched by throttled rules and retry up to 24 hours or as described within the target's custom retry policy. You can detect and alarm throttled rules using the `ThrottledRules` metric
- For low-latency use cases, you can also monitor latency using `IngestionToInvocationStartLatency`, which provides an indication of health of your event bus. Any extended periods of high latency over 30 seconds may indicate a service disruption or rule throttling.

Using AWS Serverless Application Model templates to deploy Amazon EventBridge resources

You can build and test rules manually in the EventBridge console, which can help in the development process as you refine event patterns. However, once you are ready to deploy your application, it's easier to use a framework like AWS SAM to launch all your serverless resources consistently.

We'll use this example application to look into the ways you can use AWS SAM templates to build EventBridge resources. The template.yaml file in this example is a AWS SAM template that defines four AWS Lambda functions and shows two different ways to integrate the Lambda functions with EventBridge.

For a walkthrough of this example application, see ???.

There are two approaches to using EventBridge and AWS SAM templates. For simple integrations where one Lambda function is invoked by one rule, the the **Combined template** approach is recommended. If you have complex routing logic, or you are connecting to resources outside of your AWS SAM template, the **Separated template** approach is the better choice.

Approaches:

- Combined template
- Separated template

Combined template

The first approach uses the `Events` property to configure the EventBridge rule. The following example code defines an event that invokes your Lambda function.

> **Note**
>
> This example automatically creates the rule on the default event bus, which exists in every AWS account. To associate the rule with a , you can add the `EventBusName` to the template.

```
atmConsumerCase3Fn:
  Type: AWS::Serverless::Function
```

```
  Properties:
    CodeUri: atmConsumer/
    Handler: handler.case3Handler
    Runtime: nodejs12.x
    Events:
      Trigger:
        Type: CloudWatchEvent
        Properties:
          Pattern:
            source:
              - custom.myATMapp
            detail-type:
              - transaction
            detail:
              result:
                - "anything-but": "approved"
```

This YAML code is equivalent to an event pattern in the EventBridge console. In YAML, you only need to define the event pattern, and AWS SAM automatically creates an IAM role with the required permissions.

Separated template

In the second approach to defining an EventBridge configuration in AWS SAM, the resources are separated more clearly in the template.

1. First, you define the Lambda function:

   ```
   atmConsumerCase1Fn:
     Type: AWS::Serverless::Function
     Properties:
       CodeUri: atmConsumer/
       Handler: handler.case1Handler
       Runtime: nodejs12.x
   ```

2. Next, define the rule using an `AWS::Events::Rule` resource. The properties define the event pattern and can also specify targets. You can explicitly define multiple targets.

   ```
   EventRuleCase1:
     Type: AWS::Events::Rule
     Properties:
       Description: "Approved transactions"
   ```

```
    EventPattern:
      source:
        - "custom.myATMapp"
      detail-type:
        - transaction
      detail:
        result:
          - "approved"
    State: "ENABLED"
    Targets:
      -
        Arn:
          Fn::GetAtt:
            - "atmConsumerCase1Fn"
            - "Arn"
        Id: "atmConsumerTarget1"
```

3. Finally, define an `AWS::Lambda::Permission` resource that grants permission to EventBridge to invoke the target.

```
PermissionForEventsToInvokeLambda:
  Type: AWS::Lambda::Permission
  Properties:
    FunctionName:
      Ref: "atmConsumerCase1Fn"
    Action: "lambda:InvokeFunction"
    Principal: "events.amazonaws.com"
    SourceArn:
      Fn::GetAtt:
        - "EventRuleCase1"
        - "Arn"
```

Generating an AWS CloudFormation template from an existing EventBridge rule

AWS CloudFormation enables you to configure and manage your AWS resources across accounts and regions in a centralized and repeatable manner by treating infrastructure as code. CloudFormation does this by letting you create *templates*, which define the resources you want to provision and manage.

EventBridge enables you to generate templates from the existing rules in your account, as an aid to help you jumpstart developing CloudFormation templates. You can select a single rule, or multiple rules to include in the template. You can then use these templates as the basis for creating stacks of resources under CloudFormation management.

For more information on CloudFormation see *The CloudFormation User Guide.*

> **Note**
>
> EventBridge does not include managed rules in the generated template.

You can also generate a template from an existing event bus, including the rules that event bus contains.

To generate an CloudFormation template from one or more rules

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Rules**.
3. Under **Select event bus**, choose the event bus that contains the rules you want to include in the template.
4. Under **Rules**, choose the rules you want to include in the generated CloudFormation template.

   For a single rule, you can also choose the rule name to display the rule's details page.
5. Choose **CloudFormation Template**, and then choose which format you want EventBridge to generate the template in: **JSON** or **YAML**.

   EventBridge displays the template, generated in the selected format.
6. EventBridge gives you the option of downloading the template file, or copying the template to the clipboard.

   - To download the template file, choose **Download**.
   - To copy the template to the clipboard, choose **Copy**.
7. To exit the template, choose **Cancel**.

Once you've customized your CloudFormation template as necessary for your use case, you can use it to create stacks in CloudFormation.

Considerations when using CloudFormation templates generated from Amazon EventBridge

Consider the following factors when using a CloudFormation template you generated from EventBridge:

- EventBridge does not include any passwords in the generate template.

  You can edit the template to include template parameters that enable users to specify passwords or other sensitive information when using the template to create or update a CloudFormation stack.

  In addition, users can use Secrets Manager to create a secret in the desired region and then edit the generated template to employ dynamic parameters.
- Targets in the generated template remain exactly as they were specified in the original event bus. This can lead to cross-region issues if you do not appropriately edit the template before using it to create stacks in other regions.

  Additionally, the generated template does not create the downstream targets automatically.

Event bus targets in Amazon EventBridge

A *target* is a resource or endpoint that EventBridge sends an event to when the event matches the event pattern defined for a rule. The rule processes the event data and sends the pertinent information to the target. To deliver event data to a target, EventBridge needs permission to access the target resource. You can define up to five targets for each rule.

When you add targets to a rule and that rule runs soon after, any new or updated targets might not be immediately invoked. Allow a short period of time for changes to take effect.

> **Note**
>
> This topic describes targets for Custom Event Bus - Classic rules. On the Custom Event Bus, a target belongs to a subscriber rather than a rule. Each subscriber delivers to exactly one target, configured in `InvokeConfiguration` with at most one parameter block that

matches the target type, and a subscriber can invoke any supported AWS API action directly through a universal target. For more information, see the section called "Subscribing to events".

Event bus targets available in the EventBridge console

You can configure the following target types for rules in the EventBridge console:

- API destinations

  API destinations are HTTPS endpoints that you can invoke as the target of an event bus rule. When you specify an API destination as a rule target, EventBridge invokes the HTTPS endpoint for any event that matches the event pattern specified in the rule, and then delivers the event information with the request. For more information, see API destinations.
- Event buses

  You can specify other event buses as targets for rules. This includes event buses in the same or different AWS accounts.
  - Cross-account event buses as targets
  - Same account event buses as targets
- AWS services

  You can have EventBridge send events to a number of AWS service resources. These include:
  - API Gateway
  - AWS AppSync
  - Batch job queue
  - CloudWatch log group
  - CodeBuild project
  - CodePipeline
  - Amazon EBS `CreateSnapshot` API call
  - EC2 Image Builder
  - EC2 `RebootInstances` API call
  - EC2 `StopInstances` API call
  - EC2 `TerminateInstances` API call

- ECS task
- Firehose delivery stream
- Glue workflow
- Incident Manager response plan
- Kinesis stream
- Lambda function (ASYNC)
- Amazon Redshift cluster data API queries
- Amazon Redshift Serverless workgroup data API queries
- SageMaker AI Pipeline
- Amazon SNS topic
- Amazon SQS standard, fair, and FIFO queues
- Step Functions state machine (ASYNC)
- Systems Manager Automation
- Systems Manager OpsItem
- Systems Manager Run Command

Target parameters

Some targets don't send the information in the event payload to the target, instead, they treat the event as a trigger for invoking a specific API. EventBridge uses the Target parameters to determine what happens with that target. These include the following:

- API destinations

  The data sent to an API destination must match the structure of the API. Use the `InputTransformer` object to make sure the data is structured correctly. If you want to include the original event payload, reference it in the `InputTransformer`.
- API Gateway

  The data sent to API Gateway must match the structure of the API. Use the `InputTransformer` object to make sure the data is structured correctly. If you want to include the original event payload, reference it in the `InputTransformer`.
- Amazon EC2 Image Builder
- Amazon Redshift Data API clusters

Use `RedshiftDataParameters`.

- Amazon SageMaker Runtime Model Building Pipelines

  Use `SageMakerPipelineParameters`.

- Amazon SQS fair and FIFO queues

  Use `SqsParameters` to specify the message group to use as the target.

- Systems Manager Run Command

  Use `RunCommandParameters` to specify the SSM document and target instances. For more information, see Systems Manager Run Command as a target.

> **Note**
>
> EventBridge does not support all JSON Path syntax and evaluate it at runtime. Supported syntax includes:
>
> - dot notation (for example,`$.detail`)
> - dashes
> - underscores
> - alphanumeric characters
> - array indices
> - wildcards (*)
> - forward slashes

Dynamic path parameters

Dynamic path parameters let you use JSON path syntax to reference event data at runtime instead of static values.

You can use dynamic JSON path syntax with target parameters to specify JSON paths instead of static values (for example, `$.detail.state`).

Requirements

The entire value must be a JSON path, not just part of it. For example:

- ✓ Correct: `RedshiftParameters.Sql` can be `$.detail.state`
- ✗ Incorrect: `RedshiftParameters.Sql` cannot be `"SELECT * FROM $.detail.state"`

EventBridge replaces these paths at runtime with data from the event payload at the specified path.

Limitations

Dynamic path parameters cannot reference new or transformed values from input transformation. The JSON path syntax is the same as input transformation syntax. For more information, see Amazon EventBridge input transformation.

Supported parameters

You can use dynamic syntax on all string, non-enum fields of these parameters:

- `EcsParameters`
- `HttpParameters` (except `HeaderParameters` keys)
- `RedshiftDataParameters`
- `SageMakerPipelineParameters`
- `SqsParameters`

Permissions

To make API calls on the resources that you own, EventBridge needs appropriate permissions. Specify an IAM execution role using the EventBridge console, or by setting the `RoleARN` parameter in `PutTargets`.

For example, the following policy defines permission to send messages to an Amazon SQS queue:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "sqs:SendMessage"
            ],
            "Resource": [
                "arn:aws-eusc:sqs:eusc-de-east-1:111122223333:sqs-queue-name"
            ]
        }
    ]
}
```

And the following trust policy enables EventBridge to assume the role:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

You can invoke an API Gateway endpoint with configured IAM authorization, but the role is optional if you haven't configured authorization. For more information, see Amazon EventBridge and AWS Identity and Access Management.

If another account is in the same Region and has granted you permission, then you can send events to that account.

For more information, see Sending and receiving events between AWS accounts in Amazon EventBridge.

If your target, such as an Amazon SQS queue, uses AWS Key Management Service (AWS KMS) encryption, you must include the following section in your KMS key policy:

```
{
  "Sid": "Allow EventBridge to use the key",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*"
}
```

AWS Batch job queues as targets

Certain parameters to AWS Batch `submitJob` can be configured via BatchParameters.

Others can be specified in the event payload. If the event payload (passed through or via InputTransformers) contains the following keys, they are mapped to `submitJob` request parameters:

- `ContainerOverrides: containerOverrides`

  > **Note**
  >
  > This includes only command, environment, memory, and vcpus

- `DependsOn: dependsOn`

  > **Note**
  >
  > This includes only jobId

- `Parameters: parameters`

CloudWatch Logs groups as targets

If you don't use an InputTransformer with a CloudWatch Logs target, the event payload is used as the log message, and the source of the event as the timestamp. If you do use an InputTransformer, the template must be:

```
{"timestamp":<timestamp>,"message":<message>}
```

EventBridge batches the entries sent to a log stream; therefore, EventBridge may deliver a single or multiple events to a log stream, depending on traffic.

CodeBuild projects as targets

EventBridge supports both standard and batch builds as targets.

If you use an input transformer to shape the source event to match the StartBuildRequest structure before it is delivered to a CodeBuild target, the parameters will be mapped 1-to-1 and passed through to `codeBuild.StartBuild` by default.

To pass the parameters to `codeBuild.StartBuildBatch` instead, transform the source event to match the StartBuildBatchRequest structure, and add the following key/value pair to the root of the transformed event:

```
"buildType": "BATCH"
```

Amazon ECS tasks as targets

If you use InputTransformers to shape the input event to a Target to match the Amazon ECS RunTask TaskOverride structure, the parameters will be mapped 1-to-1 and passed through to `ecs.RunTask`.

Incident Manager response plans as targets

If the matched event came from CloudWatch Alarms, the alarm state change details are populated into the trigger details of the StartIncidentRequest call to Incident Manager.

Systems Manager Run Command as a target

When you specify Systems Manager Run Command as a target, EventBridge calls the `SendCommand` API on your behalf. Configure the target using `RunCommandParameters`, which specifies the SSM document to run and the target instances or tags.

`RunCommandParameters` contains the following fields:

- `RunCommandTargets` — (Required) A list of key-value pairs that specify the target instances. Use `Key` set to `InstanceIds` with a list of instance IDs, or `Key` set to `tag:`*`tag-name`* with tag values to target instances by tag. You can specify between 1 and 5 run command targets.

To specify which SSM document to run and pass parameters to it, use the `Input` field on the `Target` object. The `Input` value must be a JSON object with the following structure:

```
{
  "DocumentName": "document-name",
  "DocumentVersion": "version",
  "Parameters": {
    "parameter-key": ["parameter-value"]
  }
}
```

Where:

- `DocumentName` — The name or ARN of the SSM document to run.
- `DocumentVersion` — (Optional) The version of the document. If omitted, the default version is used.
- `Parameters` — (Optional) A map of parameter names to value arrays, matching the parameters defined in the SSM document.

For example, the following AWS CLI command creates a rule that runs the `AWS-RunShellScript` document on a specific instance when an EventBridge event matches:

```
aws events put-targets --rule "my-rule" --targets '[{
  "Id": "ssm-target-1",
  "Arn": "arn:aws:ssm:region:account-id:document/AWS-RunShellScript",
  "RoleArn": "arn:aws:iam::account-id:role/EventBridgeSSMRole",
  "Input": "{\\"Parameters\\":{\\"commands\\":[\\"echo Hello from EventBridge\\"]}}",
  "RunCommandParameters": {
    "RunCommandTargets": [{
      "Key": "InstanceIds",
      "Values": ["i-0123456789abcdef0"]
    }]
  }
}]'
```

Note

The `Input` field on the `Target` object is used to pass the document name and parameters to Systems Manager Run Command. This is different from the `InputTransformer`,

which transforms the event payload. When using Systems Manager Run Command as a target, set the document parameters in `Input` and specify the target instances in `RunCommandParameters`.

Amazon SQS queues as targets

EventBridge does not support using Amazon SQS queues that are encrypted with an AWS owned key. This includes targets, as well as Amazon SQS queues specified as dead-letter queues for targets. For more information on AWS owned keys, see AWS owned keys in the *AWS Key Management Service Developer Guide*.

Amazon API Gateway targets for rules in Amazon EventBridge

You can use Amazon API Gateway to create, publish, maintain, and monitor APIs. Amazon EventBridge supports sending events to an API Gateway endpoint. When you specify an API Gateway endpoint as a target, each event sent to the target maps to a request sent to the endpoint.

Important

EventBridge supports using API Gateway *Edge-optimized* and *Regional* endpoints as targets. *Private* endpoints are not currently supported. To learn more about endpoints, see https://docs.aws.eu/apigateway/latest/developerguide/api-gateway-api-endpoint-types.html.

You can use an API Gateway target for the following use cases:

- To invoke a customer-specified API hosted in API Gateway based on AWS or third-party events.
- To invoke an endpoint periodically on a schedule.

The EventBridge JSON event information is sent as the body of the HTTP request to your endpoint. You can specify the other request attributes in the target's `HttpParameters` field as follows:

- `PathParameterValues` lists the values that correspond sequentially to any path variables in your endpoint ARN, for example `"arn:aws:execute-api:us-east-1:112233445566:myapi/dev/POST/pets/*/*"`.
- `QueryStringParameters` represents the query string parameters that EventBridge appends to the invoked endpoint.
- `HeaderParameters` defines HTTP headers to add to the request.

Note

For security considerations, the following HTTP header keys aren't permitted:

- Anything prefixed with `X-Amz` or `X-Amzn`
- `Authorization`
- `Connection`

- `Content-Encoding`
- `Content-Length`
- `Host`
- `Max-Forwards`
- `TE`
- `Transfer-Encoding`
- `Trailer`
- `Upgrade`
- `Via`
- `WWW-Authenticate`
- `X-Forwarded-For`

Dynamic Parameters

When invoking an API Gateway target, you can dynamically add data to events that are sent to the target. For more information, see the section called “Target parameters”.

Invocation Retries

As with all targets, EventBridge retries some failed invocations. For API Gateway, EventBridge retries responses sent with a 5xx or 429 HTTP status code for up to 24 hours with exponential back off and jitter. After that, EventBridge publishes a `FailedInvocations` metric in Amazon CloudWatch. EventBridge doesn't retry other 4xx HTTP errors.

Timeout

EventBridge rule API Gateway requests must have a maximum client execution timeout of 5 seconds. If API Gateway takes longer than 5 seconds to respond, EventBridge times out the request and then retries.

EventBridge Pipes API Gateway requests have a maximum timeout of 29 seconds, the API Gateway maximum.

AWS AppSync targets for rules in Amazon EventBridge

AWS AppSync enables developers to connect their applications and services to data and events with secure, serverless and high-performing GraphQL and Pub/Sub APIs. With AWS AppSync, you can publish real-time data updates to your applications with GraphQL mutations. EventBridge supports calling a valid GraphQL mutation operation for matched events. When you specify an AWS AppSync API mutation as a target, AWS AppSync processes the event via a mutation operation, which can then trigger subscriptions linked to the mutation.

> **Note**
>
> EventBridge supports AWS AppSync public GraphQL APIs. EventBridge does not currently support AWS AppSync Private APIs.

You can use an AWS AppSync GraphQL API target for the following use cases:

- To push, transform, and store event data into your configured data sources.
- To send real-time notifications to connected application clients.

> **Note**
>
> AWS AppSync targets only support calling AWS AppSync GraphQL APIs using the `AWS_IAM` authorization type.

For more information on AWS AppSync GraphQL APIs, see the GraphQL and AWS AppSync architecture in the *AWS AppSync Developer Guide*.

To specify an AWS AppSync target for an EventBridge rule using the console

1. Create or edit the rule.
2. Under **Target**, specify the target by choosing **AWS service** and then **AWS AppSync**.
3. Specify the mutation operation to be parsed and executed, along with the selection set.
   - Choose the AWS AppSync API, and then the GraphQL API mutation to invoke.
   - Under **Configure parameters and selection set**, choose to create a selection set using key-value mapping or an input transformer.

Key-value mapping

To use key-value mapping to create your selection set:

- Specify variables for the API parameters. Each variables can be either a static values or a dynamic JSON path expression to the event payload.
- Under **Selection set**, choose the variables you want included in the response.

Input transformer

To use an input transformer to create your selection set:

- Specify an input path that defines the variables to use.
- Specify an input template to define and format the information you want passed to the target.

For more information, see ???.

4. For **Execution role**, choose whether to create a new role or use an existing role.
5. Complete creating or editing the rule.

Example: AWS AppSync targets for Amazon EventBridge

In the following example, we'll walk through how to specifying an AWS AppSync target for an EventBridge rule, including defining an input transformation to format events for delivery.

Suppose you have an AWS AppSync GraphQL API, `Ec2EventAPI`, defined by the following schema:

```
type Event {
    id: ID!
    statusCode: String
    instanceId: String
}

type Mutation {
    pushEvent(id: ID!, statusCode: String!, instanceId: String): Event
}

type Query {
    listEvents: [Event]
}

type Subscription {
```

```
    subscribeToEvent(id: ID, statusCode: String, instanceId: String): Event
        @aws_subscribe(mutations: ["pushEvent"])
}
```

Applications clients that use this API can subscribe to the `subscribeToEvent` subscription, which is triggered by the `pushEvent` mutation.

You can create an EventBridge rule with a target that sends events to the AppSync API via the `pushEvent` mutation. When the mutation is invoked, any client that is subscribed will receive the event.

To specifying this API as the target for an EventBridge rule, you would do the following:

1. Set the Amazon Resource Name (ARN) of the rule target to the GraphQL endpoint ARN of the `Ec2EventAPI` API.
2. Specify the mutation GraphQL Operation as a target parameter:

   ```
   mutation CreatePushEvent($id: ID!, $statusCode: String, $instanceId: String) {
     pushEvent(id: $input, statusCode: $statusCode, instanceId: $instanceId) {
       id
       statusCode
       instanceId
     }
   }
   ```

   Your mutation selection set must include all the fields you wish to subscribe to in your GraphQL subscription.
3. Configure an input transformer to specify how data from matched events is used in your operation.

   Suppose you selected the "`EC2 Instance Launch Successful`" sample event:

   ```
   {
     "version": "0",
     "id": "3e3c153a-8339-4e30-8c35-687ebef853fe",
     "detail-type": "EC2 Instance Launch Successful",
     "source": "aws.autoscaling",
     "account": "123456789012",
     "time": "2015-11-11T21:31:47Z",
     "region": "us-east-1",
   ```

```
  "resources": ["arn:aws:autoscaling:us-
east-1:123456789012:autoScalingGroup:eb56d16b-bbf0-401d-b893-
d5978ed4a025:autoScalingGroupName/sampleLuanchSucASG", "arn:aws:ec2:us-
east-1:123456789012:instance/i-b188560f"],
  "detail": {
    "StatusCode": "InProgress",
    "AutoScalingGroupName": "sampleLuanchSucASG",
    "ActivityId": "9cabb81f-42de-417d-8aa7-ce16bf026590",
    "Details": {
      "Availability Zone": "us-east-1b",
      "Subnet ID": "subnet-95bfcebe"
    },
    "RequestId": "9cabb81f-42de-417d-8aa7-ce16bf026590",
    "EndTime": "2015-11-11T21:31:47.208Z",
    "EC2InstanceId": "i-b188560f",
    "StartTime": "2015-11-11T21:31:13.671Z",
    "Cause": "At 2015-11-11T21:31:10Z a user request created an AutoScalingGroup
 changing the desired capacity from 0 to 1.  At 2015-11-11T21:31:11Z an instance was
 started in response to a difference between desired and actual capacity, increasing
 the capacity from 0 to 1."
  }
}
```

You can define the following variables for use in your template, using the target input transformer's input path:

```
{
  "id": "$.id",
  "statusCode": "$.detail.StatusCode",
  "EC2InstanceId": "$.detail.EC2InstanceId"
}
```

Compose the input transformer template to define the variables that EventBridge passes to the AWS AppSync mutation operation. The template must evaluate to JSON. Given our input path, you can compose the following template:

```
{
  "id": <id>,
  "statusCode": <statusCode>,
  "instanceId": <EC2InstanceId>
}
```

Sending events to an AWS service in another account in EventBridge

EventBridge can send events from an event bus in one AWS account to supported AWS services in another account, thereby simplifying the architecture of your event-driven solutions and reducing latency.

For example, suppose you have a set of event buses, hosted in multiple accounts, that you require to send security-related events to an Amazon SQS queue in a centralized account for further asynchronous processing and analysis.

EventBridge supports sending events to cross-account targets in the same Region.

Supported services

EventBridge supports sending events to the following targets in other AWS accounts:

- Amazon API Gateway APIs
- Amazon Kinesis Data Streams streams
- Lambda functions
- Amazon SNS topics
- Amazon SQS queues

For pricing, see Amazon EventBridge pricing.

Permissions

Enabling access for cross-account event delivery to AWS services as targets involves the following steps:

- Specify an execution role
- Attach a resource policy to the target

Specify an execution role

Specify an execution role for EventBridge to use when sending events to the target when the rule is triggered.

This execution role must be in the same account as the event bus. EventBridge assumes this role when attempting to invoke the target, and any Service Control Policies (SCPs) affecting this account are applied.

SCPs are a type of organization policy that you can use to manage permissions in your organization. For more information, see Service control policies in the *AWS Organizations User Guide*.

For example, the following policy allows the EventBridge service to assume the execution role:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

And the following policy allows the role to send messages to Amazon SQS queues:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sqs:SendMessage",
      "Resource": "arn:aws-eusc:sqs:eusc-de-east-1:123456789012:queue-name"
    }
  ]
}
```

For accounts using AWS Organizations, you can apply an SCP to prevent invoking resources that do not belong to your organization, as shown in the following example:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
          {
            "Action": [
                "*"
            ],
            "Resource": "*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEquals": {
                    "aws:ResourceOrgID": "o-1234567890"
                }
            }
        }
    ]
}
```

Note

For cross-account targets other than event buses, calling `PutTarget` from a different account than the event bus, even if supplying an execution role from the calling account, is not supported.

Attach a resource access policy to the target

The AWS services that can receive cross-account events support IAM resource-based policies. This enables you to attach a resource access policy to the target, so you can specify which account has access to it.

Building on our earlier example, the following policy allows the event bus account access to the Amazon SQS queue in the target account:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "SQS:SendMessage"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws-eusc:sqs:eusc-de-east-1:444455556666:queue-name",
      "Principal": {
        "AWS": "111122223333"
      }
    }
  ]
}
```

For more information, see Identity-based policies and resource-based policies in the *AWS Identity and Access Management User Guide*.

Creating rules that send events to AWS services in other accounts

Specifying an AWS service in another account as a target is part of creating the event bus rule.

To create a rule that sends events to an AWS service in a different AWS account using the console

1. Follow the steps in the ??? procedure.
2. In the ??? step, when prompted to choose a target type:
   a. Select **AWS service**.
   b. Select an AWS service that supports cross-account targets.

      For more information, see ???.
   c. For **Target location** choose **Target in another AWS account**.
   d. Enter the ARN of the target resource to which you want to send events.
   e. Select the name of the execution role to use from the dropdown list.

f. Provide any additional information requested for the service you select. The fields displayed vary depending on the service selected.

3. Complete creating the rule following the procedure steps.

Sending and receiving events between AWS accounts in Amazon EventBridge

You can configure EventBridge to send and receive events between event buses in AWS accounts. When you configure EventBridge to send or receive events between accounts, you can specify which AWS accounts can send events to or receive events from the event bus in your account. You can also allow or deny events from specific rules associated with the event bus, or events from specific sources. For more information, see Simplifying cross-account access with Amazon EventBridge resource policies

> **Note**
>
> If you use AWS Organizations, you can specify an organization and grant access to all accounts in that organization. In addition, the sending event bus must have IAM roles attached to them when sending events to another account. For more information, see What is AWS Organizations in the *AWS Organizations User Guide*.

> **Note**
>
> If you're using an Incident Manager response plan as a target, all the response plans that are shared with your account are available by default.

You can send and receive events between event buses in AWS accounts within the same Region in all Regions and between accounts in different Regions as long as the destination Region is a supported cross-Region destination Region.

The steps to configure EventBridge to send events to or receive events from an event bus in a different account include the following:

- On the *receiver* account, edit the permissions on an event bus to allow specified AWS accounts, an organization, or all AWS accounts to send events to the receiver account.

- On the *sender* account, set up one or more rules that have the receiver account's event bus as the target.

  If the sender account inherits permissions to send events from an AWS Organization, the sender account also must have an IAM role with policies that enable it to send events to the receiver account. If you use the AWS Management Console to create the rule that targets the event bus in the receiver account, the role is created automatically. If you use the AWS CLI, you must create the role manually.
- On the *receiver* account, set up one or more rules that match events that come from the sender account.

Events sent from one account to another are charged to the sending account as custom events. The receiving account is not charged. For more information, see Amazon EventBridge Pricing.

If a receiver account sets up a rule that sends events received from a sender account on to a third account, these events are not sent to the third account.

If you have three event buses in the same account, and set up a rule on the first event bus to forward events from the second event bus to a third event bus, those events are not sent to the third event bus.

The following video covers routing events between accounts: Routing events to buses in other AWS accounts

Grant permissions to allow events from other AWS accounts

To receive events from other accounts or organizations, you must first edit the permissions on the event bus where you intend to receive events. The default event bus accepts events from AWS services, other authorized AWS accounts, and `PutEvents` calls. The permissions for an event bus are granted or denied using a resource-based policy attached to the event bus. In the policy, you can grant permissions to other AWS accounts using the account ID, or to an AWS organization using the organization ID. To learn more about event bus permissions, including example policies, see Permissions for event buses in Amazon EventBridge.

Note

EventBridge now requires all new cross account event bus targets to add IAM roles. This only applies to event bus targets created after March 2, 2023. Applications created without an IAM role before that date are not affected. However, we recommend adding IAM roles to grant users access to resources in another account, as this ensures organization boundaries using Service Control Policies (SCPs) are applied to determine who can send and receive events from accounts in your organization.

Important

If you choose to receive events from all AWS accounts, be careful to create rules that match only the events to receive from others. To create more secure rules, make sure that the event pattern for each rule contains an `Account` field with the account IDs of one or more accounts from which to receive events. Rules that have an event pattern containing an Account field do not match events sent from accounts that are not listed in the `Account` field. For more information, see Events in Amazon EventBridge.

Rules for events between AWS accounts

If your account is set up to receive events from event buses in other AWS accounts, you can write rules that match those events. Set the event pattern of the rule to match the events you are receiving from event buses in the other account.

Unless you specify `account` in the event pattern of a rule, any of your account's rules, both new and existing, that match events you receive from event buses in other accounts trigger based on

those events. If you are receiving events from event buses in another account, and you want a rule to trigger only on that event pattern when it is generated from your own account, you must add `account` and specify your own account ID to the event pattern of the rule.

If you set up your AWS account to accept events from event buses in all AWS accounts, we strongly recommend that you add `account` to every EventBridge rule in your account. This prevents rules in your account from triggering on events from unknown AWS accounts. When you specify the `account` field in the rule, you can specify the account IDs of more than one AWS account in the field.

To have a rule trigger on a matching event from any event buses in AWS account that you have granted permissions to, do not specify * in the `account` field of the rule. Doing so would not match any events, because * never appears in the `account` field of an event. Instead, just omit the `account` field from the rule.

Creating rules that send events between AWS accounts

Specifying an event bus in another account as a target is part of creating the rule.

To create a rule that sends events to a different AWS account using the console

1. Follow the steps in the ??? procedure.
2. In the ??? step, when prompted to choose a target type:
    a. Select **EventBridge event bus**.
    b. Select **Event bus in a different account or Region**.
    c. For **Event bus as target**, enter the ARN of the event bus you want to use.
3. Complete creating the rule following the procedure steps.

Sending and receiving events between AWS Regions in Amazon EventBridge

You can configure EventBridge to send and receive events between AWS Regions. You can also allow or deny events from specific Regions, specific rules associated with the event bus, or events from specific sources. For more information, see Introducing cross-Region event routing with Amazon EventBridge

The following video covers routing events between Regions using the https://eusc-de-east-1.console.amazonaws-eusc.eu/events/, AWS CloudFormation, and AWS Serverless Application Model: Cross-Region event routing

Creating rules that send events to a different AWS Region

Specifying an event bus in another AWS Region as a target is part of creating the rule.

To create a rule that sends events to a different AWS account using the console

1. Follow the steps in the ??? procedure.
2. In the ??? step, when prompted to choose a target type:
   a. Select **EventBridge event bus**.
   b. Select **Event bus in a different account or Region**.
   c. For **Event bus as target**, enter the ARN of the event bus you want to use.
3. Complete creating the rule following the procedure steps.

Sending events between event buses in the same account and Region in Amazon EventBridge

You can configure EventBridge to send and receive events between event buses in the same AWS account and Region.

When you configure EventBridge to send or receive events between event buses, you use IAM roles on the *sender* event bus to give the *sender* event bus permission to send events to the *receiver* event bus. You use Resource-based policies on the *receiver* event bus to give the *receiver* event bus permission to receive events from the *sender* event bus. You can also allow or deny events from certain event buses, specific rules associated with the event bus, or events from specific sources. For more information about event bus permissions, including example policies, see Permissions for event buses in Amazon EventBridge.

The steps to configure EventBridge to send events to or receive events between event buses in your account include the following:

- To use an existing IAM role, you need to give either the sender event bus permissions to the receiver event bus or the receiver event bus permissions to the sender event bus.
- On the *sender* event bus, set up one or more rules that have the receiver event bus as the target and create an IAM role. For an example of the policy that should be attached to the role, see ???.
- On the *receiver* event bus, edit the permissions to allow events to be passed from the other event bus.
- On the *receiver* event, set up one or more rules that match events that come from the sender event bus.

> **Note**
>
> EventBridge can't route events received from a sender event bus to a third event bus.

Events sent from one event bus to another are charged as custom events. For more information, see Amazon EventBridge Pricing.

Creating rules that send events to a different event bus in the same AWS account and Region

To send events to another event bus, you create a rule with an event bus as a target. Specifying an event bus in the same AWS account and Region as a target is part of creating the rule.

To create a rule that sends events to a different event bus in the same AWS account and Region using the console

1. Follow the steps in the ??? procedure.

2. In the ??? step, when prompted to choose a target type:

   a. Select **EventBridge event bus**.

   b. Select **Event bus in the same AWS account and Region**.

   c. For **Event bus as a target**, select an event bus from the drop-down list.

3. Complete creating the rule following the procedure steps.

Amazon EventBridge input transformation

You can customize the text from an event before EventBridge passes the information to the target of a rule. Using the input transformer in the console or the API, you define variables that use JSON path to reference values in the original event source. The transformed event is sent to a target instead of the original event. However, dynamic path parameters must reference the original event, not the transformed event. You can define up to 100 variables, assigning each a value from the input. Then you can use those variables in the *Input Template* as <*variable-name*>.

For a tutorial on using input transformer, see ???.

> **Note**
>
> EventBridge does not support all JSON Path syntax and evaluate it at runtime. Supported syntax includes:
>
> - dot notation (for example,`$.detail`)
> - dashes
> - underscores
> - alphanumeric characters
> - array indices
> - wildcards (*)
> - forward slashes

In this topic:

- Predefined variables
- Input transform examples
- Transforming input by using the EventBridge API
- Transforming input by using AWS CloudFormation
- Common Issues with transforming input
- Configuring an input transformer when creating a rule in EventBridge
- Testing a target input transformer using the EventBridge Sandbox

Predefined variables

There are pre-defined variables you can use without defining a JSON path. These variables are reserved, and you can't create variables with these names:

- `aws.events.rule-arn` — The Amazon Resource Name (ARN) of the EventBridge rule.
- `aws.events.rule-name` — The Name of the EventBridge rule.
- `aws.events.event.ingestion-time` — The time at which the event was received by EventBridge. This is an ISO 8601 timestamp. This variable is generated by EventBridge and can't be overwritten.
- `aws.events.event` — The original event payload as JSON (without the `detail` field). Can only be used as a value for a JSON field, as it's contents are not escaped.
- `aws.events.event.json` — The full original event payload as JSON. (with the `detail` field). Can only be used as a value for a JSON field, as it's contents are not escaped.

Input transform examples

The following is an example Amazon EC2 event.

```
{
  "version": "0",
  "id": "7bf73129-1428-4cd3-a780-95db273d1602",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "123456789012",
  "time": "2015-11-11T21:29:54Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:ec2:us-east-1:123456789012:instance/i-abcd1111"
  ],
  "detail": {
    "instance-id": "i-0123456789",
    "state": "RUNNING"
  }
}
```

When defining a rule in the console, select the **Input Transformer** option under **Configure input**. This option displays two text boxes: one for the *Input Path* and one for the *Input Template*.

Input path

Input Path is used to define variables. Use JSON path to reference items in your event and store those values in variables. For instance, you could create an *Input Path* to reference values in the example event by entering the following in the first text box. You can also use brackets and indices to get items from arrays.

> **Note**
>
> EventBridge replaces input transformers at runtime to ensure a valid JSON output. Because of this, put quotes around variables that refer to JSON path parameters, but do not put quotes around variables that refer to JSON objects or arrays.

```
{
  "timestamp" : "$.time",
  "instance" : "$.detail.instance-id",
  "state" : "$.detail.state",
  "resource" : "$.resources[0]"
}
```

This defines four variables, `<timestamp>`, `<instance>`, `<state>`, and `<resource>`. You can reference these variables as you create your *Input Template*.

> **Note**
>
> Amazon CloudWatch Logs and SSM targets do not support the use of `Input` and `InputPath` in their input transformers.

Input template

The *Input Template* is a template for the information you want to pass to your target. You can create a template that passes either a string or JSON to the target. Using the previous event and *Input Path*, the following *Input Template* examples will transform the event to the example output before routing it to a target.

Description	Template	Output
Simple string	`"instance <instance> is in <state>"`	`"instance i-0123456789 is in RUNNING"`
String with escaped quotes	`"instance \"<instance>\" is in <state>"`	`"instance \"i-0123456789\" is in RUNNING"` Note that this is the behavior in the EventBridge console. The AWS CLI escapes the slash characters and the result is `"instance "i-0123456789" is in RUNNING"`.
Simple JSON	`{ "instance" : <instance>, "state": <state> }`	`{ "instance" : "i-0123456789", "state": "RUNNING" }`
JSON with strings and variables	`{ "instance" : <instance>, "state": "<state>", "instanceStatus": "instance \"<instance>\" is in <state>" }`	`{ "instance" : "i-0123456789", "state": "RUNNING", "instanceStatus": "instance \"i-0123456789\" is in RUNNING" }`
JSON with a mix of variables and static information	`{ "instance" : <instance>, "state": [ 9, <state>, true ], "Transformed" : "Yes"`	`{ "instance" : "i-0123456789", "state": [ 9, "RUNNING", true`

Description	Template	Output

	`}`	`],`
		`"Transformed" : "Yes"`
		`}`

Including reserved variables in JSON	`{`
	`"instance" : <instance>,`
	`"state": <state>,`
	`"ruleArn" : <aws.events.rule-arn>,`
	`"ruleName" : <aws.events.rule-name>,`
	`"originalEvent" : <aws.events.event.json>`
	`}`
		`{`
		`"instance" : "i-0123456789",`
		`"state": "RUNNING",`
		`"ruleArn" : "arn:aws:events:us-east-2:123456789012:rule/example",`
		`"ruleName" : "example",`
		`"originalEvent" : {`
		`... // commented for brevity`
		`}`
		`}`

Including reserved variables in a string	`"<aws.events.rule-name> triggered"`	`"example triggered"`

Amazon CloudWatch log group	`{`
	`"timestamp" : <timestamp>,`
	`"message": "instance \"<instance>\" is in <state>"`
	`}`
		`{`
		`"timestamp" : 2015-11-11T21:29:54Z,`
		`"message": "instance "i-0123456789" is in RUNNING`
		`}`

Transforming input by using the EventBridge API

For information about using the EventBridge API to transform input, see Use Input Transformer to extract data from an event and input that data to the target.

Transforming input by using AWS CloudFormation

For information about using AWS CloudFormation to transform input, see AWS::Events::Rule InputTransformer.

Common Issues with transforming input

These are some common issues when transforming input in EventBridge:

- For Strings, quotes are required.
- There is no validation when creating JSON path for your template.
- If you specify a variable to match a JSON path that doesn't exist in the event, that variable isn't created and won't appear in the output.
- JSON properties like `aws.events.event.json` can only be used as the value of a JSON field, not inline in other strings.
- EventBridge doesn't escape values extracted by *Input Path*, when populating the *Input Template* for a target.
- If a JSON path references a JSON object or array, but the variable is referenced in a string, EventBridge removes any internal quotes to ensure a valid string. For example, for a variable `<detail>` pointed at `$.detail`, "Detail is <detail>" would result in EventBridge removing quotes from the object.

  Therefore, if you want to output a JSON object based on a single JSON path variable, you must place it as a key. In this example, `{"detail": <detail>}`.
- Quotes are not required for variables that represent strings. They are permitted, but EventBridge automatically adds quotes to string variable values during transformation, to ensure the transformation output is valid JSON. EventBridge does not add quotes to variables that represent JSON objects or arrays. Do not add quotes for variables that represent JSON objects or arrays.

  For example, the following input template includes variables that represent both strings and JSON objects:

```
{
  "ruleArn" : <aws.events.rule-arn>,
  "ruleName" : <aws.events.rule-name>,
  "originalEvent" : <aws.events.event.json>
}
```

Resulting in valid JSON with proper quotation:

```
{
  "ruleArn" : "arn:aws:events:us-east-2:123456789012:rule/example",
  "ruleName" : "example",
  "originalEvent" : {
    ... // commented for brevity
  }
}
```

- For (non-JSON) text output as multi-line strings, wrap each separate line in your input template in double quotes.

  For example, if you were matching Amazon Inspector Finding events against the following event pattern:

```
{
  "detail": {
    "severity": ["HIGH"],
    "status": ["ACTIVE"]
  },
  "detail-type": ["Inspector2 Finding"],
  "source": ["inspector2"]
}
```

  And using the following input path:

```
{
  "account": "$.detail.awsAccountId",
  "ami": "$.detail.resources[0].details.awsEc2Instance.imageId",
  "arn": "$.detail.findingArn",
  "description": "$.detail.description",
  "instance": "$.detail.resources[0].id",
  "platform": "$.detail.resources[0].details.awsEc2Instance.platform",
  "region": "$.detail.resources[0].region",
  "severity": "$.detail.severity",
  "time": "$.time",
  "title": "$.detail.title",
  "type": "$.detail.type"
}
```

You could use the input template below to generate multi-line string output:

```
"<severity> severity finding <title>"
"Description: <description>"
"ARN: \"<arn>\""
"Type: <type>"
"AWS Account: <account>"
"Region: <region>"
"EC2 Instance: <instance>"
"Platform: <platform>"
"AMI: <ami>"
```

Configuring an input transformer when creating a rule in EventBridge

As part of creating a rule, you can specify an input transformer for EventBridge to use to process matching events prior to sending those event to the specified target. You can configure input transformers for targets that are AWS services or API destinations.

To create a target input transformer as part of a rule

1. Follow the steps for creating a rule as detailed in ???.
2. In **Step 3 - Select target(s)**, expand **Additional settings**.
3. For **Configure target input**, choose **Input transformer** in the dropdown.

   Click **Configure input transformer**.

   EventBridge displays the **Configure input transformer** dialog box.
4. In the **Sample event** section, choose a **Sample event type** against which you want to test your event pattern. You can choose an AWS event, a partner event, or enter your own custom event.

   AWS events

   Select from events emitted from supported AWS services.

   1. Select **AWS events**.
   2. Under **Sample events**, choose the desired AWS event. Events are organized by AWS service.

      When you select an event, EventBridge populates the sample event.

For example, if you choose **S3 Object Created**, EventBridge displays a sample S3 Object Created event.

3. (Optional) You can also select **Copy** to copy the sample event to your device's clipboard.

Partner events

Select from events emitted from third-party services that support EventBridge, such as Salesforce.

1. Select **EventBridge partner events**.
2. Under **Sample events**, choose the desired partner event. Events are organized by partner.

   When you select an event, EventBridge populates the sample event.
3. (Optional) You can also select **Copy** to copy the sample event to your device's clipboard.

Enter your own

Enter your own event in JSON text.

1. Select **Enter your own**.
2. EventBridge populates the sample event with a template of required event attributes.
3. Edit and add to the sample event as desired. The sample event must be valid JSON.
4. (Optional) You can also choose any of the following options:
   - **Copy** – Copy the sample event to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

5. (Optional) Expand the **Example input paths, Templates and Outputs** section to see examples of:

- How JSON paths are used to define variables that represent event data
- How those variables can be used in an input transformer template
- The resulting output that EventBridge sends to the target

For more detailed examples of input transformations, see ???.

6. In the **Target input transformer** section, define any variables you want to use in the input template.

   Variables use JSON path to reference values in the original event source. You can then reference those variables in the input template in order to include data from the original source event in the transformed event that EventBridge passes to the target. You can define up to 100 variables. The input transformer must be valid JSON.

   For example, suppose you had chosen AWS event **S3 Object Created** as your sample event for this input transformer. You could then define the following variables for use in your template:

   ```
   {
     "requester": "$.detail.requester",
     "key": "$.detail.object.key",
     "bucket": "$.detail.bucket.name"
   }
   ```

   (Optional) You can also choose **Copy** to copy the input transformer to your device's clipboard.

7. In the **Template** section, compose the template you want to use to determine what EventBridge passes to the target.

   You can use JSON, strings, static information, variables you've defined as well as reserved variables. For more detailed examples of input transformations, see ???.

   For example, suppose you had defined the variables in the previous example. You could then compose the following template, which references those variables, as well as reserved variables, and static information.

   ```
   {
       "message": "<requester> has created the object \"<key>\" in the bucket
    \"<bucket>\"",
       "RuleName": <aws.events.rule-name>,
       "ruleArn" : <aws.events.rule-arn>,
       "Transformed": "Yes"
   }
   ```

   (Optional) You can also choose **Copy** to copy the template to your device's clipboard.

8. To test your template, select **Generate output**.

EventBridge processes the sample event based on the input template, and displays the transformed output generated under **Output**. This is the information that EventBridge will pass to the target in place of the original source event.

The generated output for the example input template described above would be the following:

```
{
    "message": "123456789012 has created the object "example-key" in the bucket
 "amzn-s3-demo-bucket"",
    "RuleName": rule-name,
    "ruleArn" : arn:aws:events:us-east-1:123456789012:rule/rule-name,
    "Transformed": "Yes"
}
```

(Optional) You can also choose **Copy** to copy the generated output to your device's clipboard.

9. Select **Confirm**
10. Follow the rest of the steps for creating a rule as detailed in ???.

Testing a target input transformer using the EventBridge Sandbox

You can use input transformers to customize the text from an event before EventBridge passes the information to the target of a rule.

Configuring an input transformer is typically part of the larger process of specifying a target while creating a new rule or editing an existing one. Using the Sandbox in EventBridge, however, you can quickly configure an input transformer and use a sample event to confirm you are getting the desired output, without having to create or edit a rule.

For more information about input transformations, see ???.

To test a target input transformer

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Under **Developer resources**, choose **Sandbox**, and on the **Sandbox** page choose the **Target input transformer** tab.

3. In the **Sample event** section, choose a **Sample event type** against which you want to test your event pattern. You can choose an AWS event, a partner event, or enter your own custom event.

   AWS events

   Select from events emitted from supported AWS services.

   1. Select **AWS events**.
   2. Under **Sample events**, choose the desired AWS event. Events are organized by AWS service.

      When you select an event, EventBridge populates the sample event.

      For example, if you choose **S3 Object Created**, EventBridge displays a sample S3 Object Created event.
   3. (Optional) You can also select **Copy** to copy the sample event to your device's clipboard.

   Partner events

   Select from events emitted from third-party services that support EventBridge, such as Salesforce.

   1. Select **EventBridge partner events**.
   2. Under **Sample events**, choose the desired partner event. Events are organized by partner.

      When you select an event, EventBridge populates the sample event.
   3. (Optional) You can also select **Copy** to copy the sample event to your device's clipboard.

   Enter your own

   Enter your own event in JSON text.

   1. Select **Enter your own**.
   2. EventBridge populates the sample event with a template of required event attributes.
   3. Edit and add to the sample event as desired. The sample event must be valid JSON.
   4. (Optional) You can also choose any of the following options:
      - **Copy** – Copy the sample event to your device's clipboard.

- **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

4. (Optional) Expand the **Example input paths, Templates and Outputs** section to see examples of:

   - How JSON paths are used to define variables that represent event data
   - How those variables can be used in an input transformer template
   - The resulting output that EventBridge sends to the target

   For more detailed examples of input transformations, see ???.

5. In the **Target input transformer** section, define any variables you want to use in the input template.

   Variables use JSON path to reference values in the original event source. You can then reference those variables in the input template in order to include data from the original source event in the transformed event that EventBridge passes to the target. You can define up to 100 variables. The input transformer must be valid JSON.

   For example, suppose you had chosen AWS event **S3 Object Created** as your sample event for this input transformer. You could then define the following variables for use in your template:

   ```
   {
     "requester": "$.detail.requester",
     "key": "$.detail.object.key",
     "bucket": "$.detail.bucket.name"
   }
   ```

   (Optional) You can also choose **Copy** to copy the input transformer to your device's clipboard.

6. In the **Template** section, compose the template you want to use to determine what EventBridge passes to the target.

   You can use JSON, strings, static information, variables you've defined as well as reserved variables. For more detailed examples of input transformations, see ???.

   For example, suppose you had defined the variables in the previous example. You could then compose the following template, which references those variables, as well as reserved variables, and static information.

```
{
    "message": "<requester> has created the object \"<key>\" in the bucket
 \"<bucket>\"",
    "RuleName": <aws.events.rule-name>,
    "ruleArn" : <aws.events.rule-arn>,
    "Transformed": "Yes"
}
```

(Optional) You can also choose **Copy** to copy the template to your device's clipboard.

7. To test your template, select **Generate output**.

   EventBridge processes the sample event based on the input template, and displays the transformed output generated under **Output**. This is the information that EventBridge will pass to the target in place of the original source event.

   The generated output for the example input template described above would be the following:

```
{
    "message": "123456789012 has created the object "example-key" in the bucket
 "amzn-s3-demo-bucket"",
    "RuleName": rule-name,
    "ruleArn" : arn:aws:events:us-east-1:123456789012:rule/rule-name,
    "Transformed": "Yes"
}
```

   (Optional) You can also choose **Copy** to copy the generated output to your device's clipboard.

Archiving and replaying events in Amazon EventBridge

In EventBridge, you can create an archive of events so that you can easily *replay*, or resend them to the event bus that originally received them, at a later time. For example, you might want to replay events to recover from errors, or to validate new functionality in your application.

Archiving events

When you create an archive, you can specify:

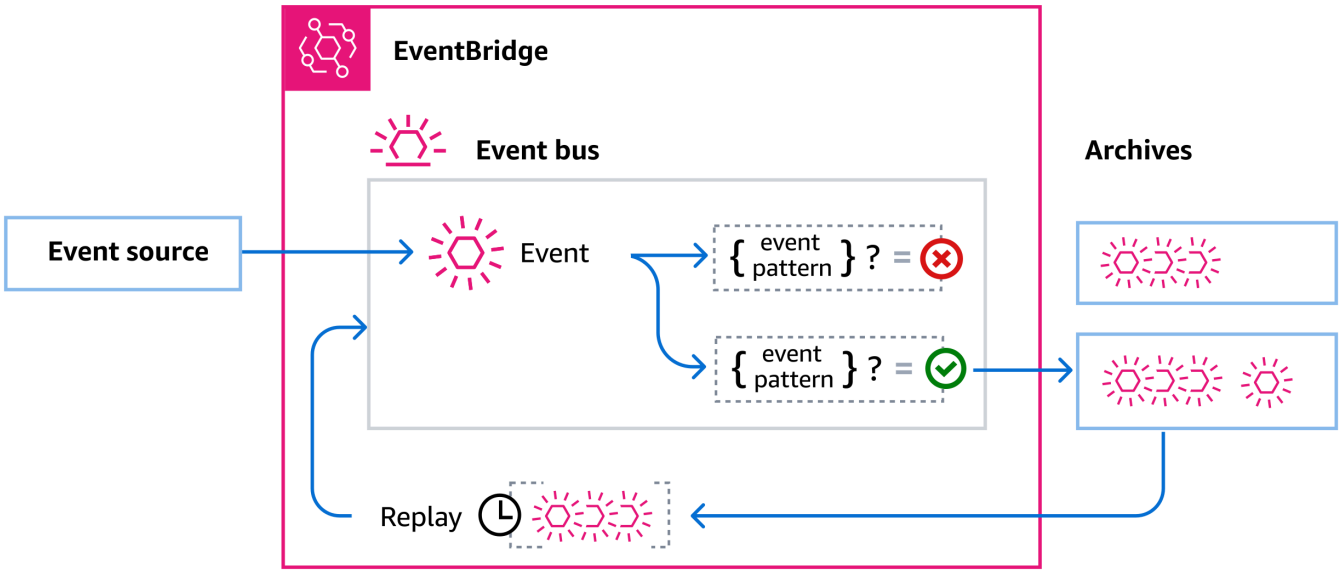
- Which events to send to the archive.

You can specify an event pattern for EventBridge to use when filtering the events it sends to the archive.

- How long to retain events in the archive.

  You can specify the number of days to retain events in the archive. By default, EventBridge stores events in an archive indefinitely.

Each archive receives events from a single *source* event bus. You cannot change the source event bus once an archive is created. You can create multiple archives for a given event bus.



EventBridge charges apply to archives. Please refer to Amazon EventBridge Pricing for details.

Encrypting archive events

By default, EventBridge encrypts event data in an archive using 256-bit Advanced Encryption Standard (AES-256) under an AWS owned CMK, which helps secure your data from unauthorized access.

Event delivery

Keep in the following considerations in mind about how EventBridge delivers events to archives:

- There may be a delay between an event being received on an event bus and the event arriving in the archive. We recommend you delay replaying archived events for 10 minutes to make sure all events are replayed.
- The `EventCount` and `SizeBytes` values of the `DescribeArchive` operation have a reconciliation period of 24 hours. Therefore, any recently-expired or newly-archived events may not be immediately reflected in these values.

Preventing replayed events from being delivered to an archive

When you create an archive, EventBridge generates a managed rule on the source event bus that prevents replayed events from being sent to the archive. The managed rule adds the following event pattern, which filters events based on whether it contains a `replay-name` field. (EventBridge adds this field to events when it replays them.)

```
{
  "replay-name": [{
    "exists": false
  }]
}
```

Replaying events from an archive

After you create an archive, you can then replay events from the archive. For example, if you update an application with additional functionality, you can replay historical events to ensure that the events are reprocessed to keep the application consistent. You can also use an archive to replay events for new functionality.

When you replay events from an archive, you specify:

- The time frame from which to select events to replay.
- Optionally, specific rules on the event bus to which EventBridge should replay the selected events.

Archive events can only be replayed to the source event bus.

You can have a maximum of ten active concurrent replays per account per AWS Region.

Replaying events does not remove them from the archive. You can replay events in multiple replays. EventBridge only removes events when they exceed the archive retention period, or you delete the archive itself.

EventBridge deletes replays after 90 days.

You can cancel replays while their status is `Starting` or `Running`. For more information, see Canceling event replays.

Identifying events that have been replayed

When EventBridge sends an event from an archive to the source event bus during a replay, it adds a metadata field to the event, `replay-name`, which contains the name of the replay. You can use this field to identify replayed events when they are delivered to a target.

EventBridge also uses this field to make sure that replayed events aren't sent to archives.

Considerations when replaying events from an archive

Keep the following considerations in mind when replaying events from an archive:

- There may be a delay between an event being received on an event bus and the event arriving in the archive. We recommend you delay replaying archived events for 10 minutes to make sure all events are replayed.
- Events aren't necessarily replayed in the same order that they were added to the archive. A replay processes events to replay based on the time in the event, and replays them on one minute intervals. If you specify an event start time and an event end time that covers a 20 minute time range, the events are replayed from the first minute of that 20 minute range first. Then the events from the second minute are replayed.
- You can use the `DescribeReplay` operation of the EventBridge API to determine the progress of a replay. `EventLastReplayedTime` returns the time stamp of the last event replayed.
- Events are replayed based on, but separate from, the `PutEvents` transactions per second limit for the AWS account. You can request an increase to the limit for PutEvents. For more information, see Amazon EventBridge Quotas.

Amazon EventBridge: Creating archives and replays

Creating event archives in Amazon EventBridge

When you create an archive in EventBridge, you can determine which events are sent to the archive by specifying an event pattern. EventBridge sends events that match the event pattern to the archive. You also set the retention period to store events in the archive before they are discarded.

You can also create archives as part of creating an event bus. These archives have an indefinite retention policy and no event filter, although this can be updated once the archive is created.

Steps

- Define the archive
- Build the event pattern (optional)

Define the archive

First, enter a name and description for archive, and specify the event bus from which it receives events. Optionally, you can also set how long to retain events in the archive.

To define the archive

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Navigate to the source event bus, or create the archive directly:
   - In the navigation pane, choose **Event buses**.

     On the events bus details page, choose the **Archives** tab.
   - In the navigation pane, choose **Archives**.
3. Choose **Create archive**.
4. Under **Archive detail**, enter a name and optionally, a description for the archive.

   The name must be unique to your account in the selected Region. You can't change the name after you create the archive.
5. For **Source**, select the event bus you want to send events to the archive

   If you navigated from an existing event bus details page, the name of that event bus appears by default.

   You cannot change the source event bus once you have created the archive.

6. For **Retention period**, specify how long to retain the events in the archive:
   - Choose **Indefinite** to retain the events in the archive and not ever delete them.
   - For a set retention period, enter the number of days after which EventBridge should delete the events from the archive.
7. For **Encryption**, choose the KMS key for EventBridge to use when encrypting the events stored in the archive.

   > **Important**
   >
   > If you have specify that EventBridge use a customer managed key for encrypting the source event bus, we strongly recommend you also specify a customer managed key for any archives for the event bus as well.

   - Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

     This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

     This is the default.
   - Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

     Customer managed key are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

     - Specify an existing customer managed key, or choose **Create a new KMS key />**.

       EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.
8. Choose **Next**.

Build the event pattern (optional)

Next, as an optional step, you can build an event pattern to filter which events EventBridge sends to the archive. To do this, specify the event source, choose the basis for the event pattern, and define the attributes and values to match on. You can also generate the event pattern in JSON and test it against a sample event.

For more information on event patterns, see ???.

To build the event pattern

1. For **Event source**, choose **AWS events or EventBridge partner events**.
2. (Optional) In the **Sample events** section, choose a **Sample event type** against which you want to test your event pattern.

   The following sample event types are available:

   - **AWS events** – Select from events emitted from supported AWS services.
   - **EventBridge partner events** – Select from events emitted from third-party services that support EventBridge, such as Salesforce.
   - **Enter my own** – Enter your own event in JSON text.

     You can also use an AWS or partner event as the starting point for creating your own custom event.

     1. Select **AWS events** or **EventBridge partner events**.
     2. Use the **Sample events** dropdown to select the event you want to use as a starting point for your custom event.

        EventBridge displays the sample event.
     3. Select **Copy**.
     4. Select **Enter my own** for **Event type.**
     5. Delete the sample event structure in the JSON editing pane, and paste the AWS or partner event in its place.
     6. Edit the event JSON to create your own sample event.
3. Choose a **Creation method**. You can create an event pattern from an EventBridge schema or template, or you can create a custom event pattern.

Existing schema

To use an existing EventBridge schema to create the event pattern, do the following:

1. In the **Creation method** section, for **Method**, select **Use schema**.
2. In the **Event pattern** section, for **Schema type**, select **Select schema from Schema registry**.
3. For **Schema registry**, choose the dropdown box and enter the name of a schema registry, such as `aws.events`. You can also select an option from the dropdown list that appears.
4. For **Schema**, choose the dropdown box and enter the name of the schema to use. For example, `aws.s3@0bjectDeleted`. You can also select an option from the dropdown list that appears.
5. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

6. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
7. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Custom schema

To write a custom schema and convert it to an event pattern, do the following:

1. In the **Creation method** section, for **Method**, choose **Use schema**.
2. In the **Event pattern** section, for **Schema type**, choose **Enter schema**.
3. Enter your schema into the text box. You must format the schema as valid JSON text.
4. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

5. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
6. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Event pattern

To write a custom event pattern in JSON format, do the following:

1. In the **Creation method** section, for **Method**, choose **Custom pattern (JSON editor)**.
2. For **Event pattern**, enter your custom event pattern in JSON-formatted text.

3. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.
   - **Event pattern form** – Opens the event pattern in Pattern Builder. If the pattern can't be rendered in Pattern Builder as-is, EventBridge warns you before it opens Pattern Builder.

4. Choose **Create archive**.

To confirm that events are successfully sent to the archive, you can use the `DescribeArchive` operation of the EventBridge API to see if the `EventCount` reflects the number of events in the archive. If it is 0, there are no events in the archive.

Updating archives on Amazon EventBridge event buses

You can update the following:

- Archive description
- The event pattern used to filter which events are sent to the archive.
- The retention period for events.
- The AWS KMS key used for event encryption.

  For more information, see Encrypting archives.

You cannot change the name or source event bus for an archive once it has been created.

> **Note**
>
> Schema discovery is not supported for event buses encrypted using a customer managed key. To enable schema discovery on an event bus, choose to use an AWS owned key. For more information, see KMS key options.

To update an archive (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Navigate to the archive directly, or from the source event bus:
   - In the navigation pane, choose **Event buses**.

     On the events bus details page, choose the **Archives** tab.
   - In the navigation pane, choose **Archives**.
3. Select the archive, and then select **Edit**.
4. Update the archive.

To update an archive for an event bus (AWS CLI)

- Use update-archive.

Deleting event archives in Amazon EventBridge

When you delete an archive, EventBridge deletes the following resources:

- The archive and any events it contains.
- The event pattern, if any, specified for the archive.
- The managed rule EventBridge generated for the archive.

To delete an archive from an event bus (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Navigate to the archive directly, or from the source event bus:
   - In the navigation pane, choose **Event buses**.

     On the events bus details page, choose the **Archives** tab.
   - In the navigation pane, choose **Archives**.
3. Choose the event bus that contains the archive you want to delete.
4. On the event bus details page, select the **Archives** tab.
5. Select the archive, and then select **Delete**.

To delete an archive (AWS CLI)

- Use delete-archive.

Creating replays of archived events in Amazon EventBridge

When you start a new replay, you specify a time period for the event you want EventBridge to resend to the source event bus. You can also specify for EventBridge to send the events to specific rules.

To start an event replay (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Navigate to the replays directly, or from the archive you want to replay:

- In the navigation pane, choose **Archives**.

  On the **Archives** page, choose the archive and then choose**Replay**.
- In the navigation pane, choose **Replays**.

  Choose **Start new replay**.

3. Enter a **Name** for the replay and, optionally, a **Description**.
4. For **Source**, select the archive to replay events from.
5. For destination, you can replay events only to the same event bus that emitted the events.
6. For **Specify rules**, do one of the following:
   - Choose **All rules** to replay events to all rules.
   - Choose **Specify rules**, and then select the rule or rules to replay the events to.
7. Under **Replay time frame**, specify the **Date**, **Time**, and **Time zone** for the **Start time** and the **End time**.

   Only events that occurred between the **Start time** and **End time** are replayed.
8. Choose **Start replay**.

To start a replay (AWS CLI)

- Use start-replay.

Canceling replays of archived events in Amazon EventBridge

If you start a replay and then want to stop it, you can cancel it while its status is `Starting` or `Running`.

To cancel a replay (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the left navigation pane, choose **Replays**.
3. Choose the replay to cancel.
4. Choose **Cancel**.

To cancel a replay (AWS CLI)

- Use cancel-replay.

Making applications Regional-fault tolerant with global endpoints in EventBridge

You can improve your application's availability with Amazon EventBridge global endpoints. Global endpoints help make your application regional-fault tolerant at no additional cost. To start, you assign an Amazon Route 53 health check to the endpoint. When failover is initiated, the health check reports an “unhealthy” state. Within minutes of failover initiation, all custom events are routed to an event bus in the secondary Region and are processed by that event bus. Once the health check reports a “healthy” state, events are processed by the event bus in the primary Region.

When you use global endpoints, you can enable event replication. Event replication sends all custom events to the event buses in the primary and secondary Regions using managed rules.

> **Note**
>
> If you're using custom buses, you'll need a custom bus in each Region with the same name and in the same account for failover to work properly.

Recovery Time & Recovery Point Objectives

The Recovery Time Objective (RTO) is the time that it takes for the secondary Region to start receiving events after a failure. For RTO, the time includes time period for triggering CloudWatch alarms and updating statuses for Route 53 health checks. The Recovery Point Objective (RPO) is the measure of the data that will be left unprocessed during a failure. For RPO, the time includes events that are not replicated to the secondary Region and are stuck in the primary Region until the service or Region recovers. With global endpoints, if you follow our prescriptive guidance for alarm configuration, you can expect the RTO and RPO to be 360 seconds with a maximum of 420 seconds.

Event replication

Events are processed in the secondary Region asynchronously. This means that events are not guaranteed to be processed at the same time in both Regions. When failover is triggered, the events are processed by the secondary Region and will be processed by the primary Region when it's available. Enabling event replication will increase your monthly costs. For more information, see Amazon EventBridge pricing

We recommend enabling event replication when setting up global endpoints for the following reasons:

- Event replication helps you verify that your global endpoints are configured correctly. This helps to ensure that you'll be covered in the event of failover.
- Event replication is required to automatically recover from a failover event. If you don't have event replication enabled, you'll have to manually reset the Route 53 health check to "healthy" before events will go back to the primary Region.

Replicated event payload

The following is an example of a replicated event payload:

> **Note**
>
> For `region`, the Region that the event was replicated from is listed.

```
{
    "version": "0",
    "id": "a908baa3-65e5-ab77-367e-527c0e71bbc2",
    "detail-type": "Test",
    "source": "test.service.com",
    "account": "0123456789",
    "time": "1900-01-01T00:00:00Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:events:us-east-1:0123456789:endpoint/MyEndpoint"
    ],
    "detail": {
        "a": "b"
    }
```

```
}
```

Working with global endpoints by using an AWS SDK

> **Note**
>
> Support for C++ is coming soon.

When using an AWS SDK to work with global endpoints, keep the following in mind:

- You'll need to have the AWS Common Runtime (CRT) library installed for your specific SDK. If you don't have the CRT installed, you'll get an exception message indicating what needs to be installed. For more information, see the following:
  - AWS Common Runtime (CRT) libraries
  - awslabs/aws-crt-java
  - awslabs/aws-crt-nodejs
  - awslabs/aws-crt-python
- Once you have created a global endpoint, you'll need to add the `endpointId` and `EventBusName` to any `PutEvents` calls that you use.
- Global endpoints support Signature Version 4A. This version of SigV4 allows requests to be signed for multiple AWS Regions. This is useful in API operations that might result in data access from one of several Regions. When using the AWS SDK, you supply your credentials and the requests to global endpoints will use Signature Version 4A without additional configuration. For more information about SigV4A, see Signing AWS API requests in the *AWS General Reference*.

  If you request temporary credentials from the global AWS STS endpoint (sts.amazonaws.com), AWS STS vends credentials which, by default, do not support SigV4A. See Managing AWS STS in an AWS Region in the *AWS Identity and Access Management User Guide* for further information.

Available Regions

The following Regions support global endpoints:

- US East (N. Virginia)
- US East (Ohio)

- US West (N. California)
- US West (Oregon)
- Canada (Central)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Milan)
- Europe (Paris)
- Europe (Stockholm)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- South America (São Paulo)

Creating a global endpoint in Amazon EventBridge

Complete the following steps to set up a global endpoint:

1. Make sure that you have matching event buses and rules in both the primary and secondary Region.
2. Create a Route 53 health check to monitor your event buses. For assistance in creating your health check, choose **New Health Check** when creating your global endpoint.
3. Create your global endpoint.

Once you have set up the Route 53 health check, you can create a global endpoint.

To create a global endpoint by using the console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.

2. In the navigation pane, choose **Global endpoints**.
3. Choose **Create Endpoint**.
4. Enter a name and description for the endpoint.
5. For **Event bus in primary Region**, choose the event bus you'd like the endpoint associated with.
6. For **Secondary Region**, choose the Region you'd like to direct events to in the event of a failover.

> **Note**
>
> The **Event bus in secondary Region** is auto-filled and not editable.

7. For **Route 53 health check for triggering failover and recovery**, choose the health check that the endpoint will monitor. If you don't already have a health check, choose **New Health check** to open the CloudFormation console and create a health check using a CloudFormation template.

> **Note**
>
> Missing data will cause the health check to fail. If you only need to send events intermittently, consider using a longer **MinimumEvaluationPeriod**, or treat missing data as 'missing' instead of 'breaching'.

8. (Optional) For **Event replication** do the following:
   a. Select **Event replication enabled**.
   b. For **Execution role**, choose whether to create a new AWS Identity and Access Management role or use an existing one. Do the following:
      - Choose **Create a new role for this specific resource**. Optionally, you can update the **Role name** to create a new role.
      - Choose **Use existing role**. Then, for **Execution role**, choose the desired role to use.
9. Choose **Create**.

To create a global endpoint by using the API

To create a global endpoint using the EventBridge API, see CreateEndpoint in the Amazon EventBridge API Reference.

To create a global endpoint by using CloudFormation

To create a global endpoint using the AWS CloudFormation API, see AWS::Events::Endpoints in the AWS CloudFormation User Guide.

Best practices for Amazon EventBridge global endpoints

The following best practices are recommended when you set up global endpoints.

Enabling event replication

We strongly recommend that you turn on replication and process your events in the secondary Region that you assign to your global endpoint. This ensures that your application in the secondary Region is configured correctly. You should also turn on replication to ensure automatic recovery to the primary Region after an issue has been mitigated.

Event IDs can change across API calls so correlating events across Regions requires you to have an immutable, unique identifier. Consumers should also be designed with idempotency in mind. That way, if you're replicating events, or replaying them from archives, there are no side effects from the events being processed in both Regions.

Preventing event throttling

To prevent events from being throttled, we recommend updating your `PutEvents` and targets limits so they're consistent across Regions.

Using subscriber metrics in Amazon Route 53 health checks

Avoid including subscriber metrics in your Amazon Route 53 health checks. Including these metrics may cause your publisher to failover to the secondary Regions if a subscriber encounters an issue despite all other subscribers remaining healthy in the primary Region. If one of your subcribers is failing to process events in the primary Region, you should turn on replication to ensure that your subscriber in the secondary Region can process events successfully.

Setting up the Route 53 health check for EventBridge global endpoints

When using global endpoints you have to have a Route 53 health check to monitor the status of your Regions. The following template defines a Amazon CloudWatch alarm and uses it to define a Route 53 health check.

Topics

- CloudFormation template for defining a Route 53 health check
- CloudWatch alarm template properties
- Route 53 health check template properties

CloudFormation template for defining a Route 53 health check

Use the following template to define your Route 53 health check.

```
Description: |-
  Global endpoints health check that will fail when the average Amazon EventBridge
  latency is above 30 seconds for a duration of 5 minutes. Note, missing data will
  cause the health check to fail, so if you only send events intermittently, consider
  changing the heath check to use a longer evaluation period or instead treat missing
  data as 'missing' instead of 'breaching'.

Metadata:
  AWS::CloudFormation::Interface:
    ParameterGroups:
      - Label:
          default: "Global endpoint health check alarm configuration"
        Parameters:
          - HealthCheckName
          - HighLatencyAlarmPeriod
          - MinimumEvaluationPeriod
          - MinimumThreshold
          - TreatMissingDataAs
    ParameterLabels:
      HealthCheckName:
        default: Health check name
      HighLatencyAlarmPeriod:
        default: High latency alarm period
      MinimumEvaluationPeriod:
        default: Minimum evaluation period
      MinimumThreshold:
```

```
        default: Minimum threshold
      TreatMissingDataAs:
        default: Treat missing data as

Parameters:
  HealthCheckName:
    Description: Name of the health check
    Type: String
    Default: LatencyFailuresHealthCheck
  HighLatencyAlarmPeriod:
    Description: The period, in seconds, over which the statistic is applied. Valid
 values are 10, 30, 60, and any multiple of 60.
    MinValue: 10
    Type: Number
    Default: 60
  MinimumEvaluationPeriod:
    Description: The number of periods over which data is compared to the specified
 threshold. You must have at least one evaluation period.
    MinValue: 1
    Type: Number
    Default: 5
  MinimumThreshold:
    Description: The value to compare with the specified statistic.
    Type: Number
    Default: 30000
  TreatMissingDataAs:
    Description: Sets how this alarm is to handle missing data points.
    Type: String
    AllowedValues:
      - breaching
      - notBreaching
      - ignore
      - missing
    Default: breaching

Mappings:
  "InsufficientDataMap":
    "missing":
      "HCConfig": "LastKnownStatus"
    "breaching":
      "HCConfig": "Unhealthy"

Resources:
  HighLatencyAlarm:
```

```
      Type: AWS::CloudWatch::Alarm
      Properties:
        AlarmDescription: High Latency in Amazon EventBridge
        MetricName: IngestionToInvocationStartLatency
        Namespace: AWS/Events
        Statistic: Average
        Period: !Ref HighLatencyAlarmPeriod
        EvaluationPeriods: !Ref MinimumEvaluationPeriod
        Threshold: !Ref MinimumThreshold
        ComparisonOperator: GreaterThanThreshold
        TreatMissingData: !Ref TreatMissingDataAs

  LatencyHealthCheck:
      Type: AWS::Route53::HealthCheck
      Properties:
        HealthCheckTags:
          - Key: Name
            Value: !Ref HealthCheckName
        HealthCheckConfig:
          Type: CLOUDWATCH_METRIC
          AlarmIdentifier:
            Name:
              Ref: HighLatencyAlarm
            Region: !Ref AWS::Region
          InsufficientDataHealthStatus: !FindInMap [InsufficientDataMap, !Ref
 TreatMissingDataAs, HCConfig]

Outputs:
  HealthCheckId:
    Description: The identifier that Amazon Route 53 assigned to the health check when
 you created it.
    Value: !GetAtt LatencyHealthCheck.HealthCheckId
```

Event IDs can change across API calls so correlating events across Regions requires you to have an immutable, unique identifier. Consumers should also be designed with idempotency in mind. That way, if you're replicating events, or replaying them from archives, there are no side effects from the events being processed in both Regions.

CloudWatch alarm template properties

> **Note**
>
> For all **`editable`** fields, consider your throughput per second. If you only send events intermittently, consider changing the heath check to use a longer evaluation period or instead treat missing data as `missing` instead of `breaching`.

The following properties are used in th CloudWatch alarm section of the template:

Metric	Description
`AlarmDescription`	The description of the alarm. Default: **`High Latency in Amazon EventBridge`**
`MetricName`	The name of the metric associated with the alarm. This is required for an alarm based on a metric. For an alarm based on a math expression, you use `Metrics` instead and you can't specify `MetricName` . Default: IngestionToInvocationStartLatency
`Namespace`	The namespace of the metric associated with the alarm. This is required for an alarm based on a metric. For an alarm based on a math expression, you can't specify `Namespace` and you use `Metrics` instead. Default: `AWS/Events`
`Statistic`	The statistic for the metric associated with the alarm, other than percentile. Default: Average
`Period`	The period, in seconds, over which the statistic is applied. This is required for an alarm based on a metric. Valid values are 10, 30, 60, and any multiple of 60. Default: **`60`**

Metric	Description
`EvaluationPeriods`	The number of periods over which data is compared to the specified threshold. If you are setting an alarm that requires that a number of consecutive data points be breaching to trigger the alarm, this value specifies that number. If you are setting an "M out of N" alarm, this value is the N, and `DatapointsToAlarm` is the M. Default: **5**
`Threshold`	The value to compare with the specified statistic. Default: **`30,000`**
`ComparisonOperator`	The arithmetic operation to use when comparing the specified statistic and threshold. The specified statistic value is used as the first operand. Default: `GreaterThanThreshold`
`TreatMissingData`	Sets how this alarm is to handle missing data points. Valid values: `breaching` , `notBreaching` , `ignore`, and `missing` Default: `breaching`

Route 53 health check template properties

> **Note**
>
> For all **`editable`** fields, consider your throughput per second. If you only send events intermittently, consider changing the heath check to use a longer evaluation period or instead treat missing data as `missing` instead of `breaching`.

The following properties are used in th Route 53 health check section of the template:

Metric	Description
`HealthCheckName`	The name of the health check.

Metric	Description
	Default: **`LatencyFailuresHealthCheck`**
`InsufficientDataHealthStatus`	When CloudWatch has insufficient data about the metric to determine the alarm state, the status that you want Amazon Route 53 to assign to the health check Valid values: • `Healthy`: Route 53 considers the health check to be healthy. • `Unhealthy` : Route 53 considers the health check to be unhealthy. • `LastKnownStatus` : Route 53 uses the status of the health check from the last time that CloudWatch had sufficient data to determine the alarm state. For new health checks that have no last known status, the default status for the health check is healthy. Default: Unhealthy **Note** This field is updated based on the input to the `TreatMissingData` field. If `TreatingMissingData` is set to `Missing`, it will be updated to `LastKnownStatus` .If `TreatingMissingData` is set to `Breaching` , it will be updated to `Unhealthy` .

Configuring logs for Amazon EventBridge event buses

You can configure EventBridge to send logs detailing how an event bus is processing events, to help with troubleshooting and debugging.

You can select the following AWS services as *log destinations* to which EventBridge delivers logs for the specified event bus:

- Amazon CloudWatch Logs

EventBridge delivers logs to the specified CloudWatch Logs log group.

Use CloudWatch Logs to centralize the logs from all of your systems, applications, and AWS services that you use, in a single, highly scalable service. For more information, see Working with log groups and log streams in the *Amazon CloudWatch Logs User Guide*.

- Amazon Data Firehose

  EventBridge delivers logs to a Firehose delivery stream.

  Amazon Data Firehose is a fully-managed service for delivering real-time streaming data to destinations such as certain AWS services, as well as any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers. For more information, see Creating an Amazon Data Firehose delivery stream in the *Amazon Data Firehose User Guide*.

- Amazon S3

  EventBridge delivers logs as Amazon S3 objects to the specified bucket.

  Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. For more information, see Uploading, downloading, and working with objects in Amazon S3 in the *Amazon Simple Storage Service User Guide*.

How logging works for event buses

EventBridge generates logs for:

- Any AWS service events that matches a rule on the event bus
- Any events delivered by the following methods, whether or not the event is ingested successfully or matches any rule:
  - Events from partner event sources
  - Events replayed from an archive
  - Events sent to the bus via `PutEvents`

EventBridge does not log events that only match managed rules.

The log data sent to each selected log destination is the same.

You can customize the logs EventBridge sends to the selected destinations in the following way:

- You can specify the *log level*, which determines the steps for which EventBridge sends logs to the selected destinations. For more information, see ???.
- You can specify whether EventBridge includes more granular information when relevant, including:
  - Event details
  - Target input information
  - Target request information

  For more information, see ???.

Log delivery considerations

Keep the following considerations in mind as you configure logging for event buses:

- Event bus log records are delivered on a best effort basis. Most requests for an event bus that is properly configured for logging result in a delivered log record. The completeness and timeliness of event bus logging is not guaranteed.
- In some circumstances, delivering event bus log records itself generates events that are then sent to EventBridge, which can lead to disruption in log record delivery. For this reason, EventBridge does not log the following events:
  - AWS KMS `Decrypt` and `GenerateDataKey` events generated when log records encrypted using a customer managed key are delivered to a log destination.
  - `PutRecordBatch` events in Firehose generated by the delivery of event bus logs.
- For S3 log destinations, specifying a destination bucket with event notification for EventBridge enabled is not recommended, as this can result in disruption in the delivery of your logs.

Logging encryption

When sending logs, EventBridge encrypts the `detail` and `error` sections of each log record with the KMS key specified for the event bus. Once delivered, the record is decrypted and then re-encrypted with the KMS key specified for the log destination.

For more information, see Encrypting event bus logs.

Specifying event bus logging permissions

To enable logging from an event bus, attach the following identity-based policy to your AWS Identity and Access Management (IAM) identity (user or role). This policy grants the `events:AllowVendedLogDeliveryForResource` action for the event bus.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ServiceLevelAccessForLogDelivery",
            "Effect": "Allow",
            "Action": [
                "events:AllowVendedLogDeliveryForResource"
            ],
            "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:event-
bus/my-event-bus*"
        }
    ]
}
```

For more information, see Service-specific permissions in the *CloudWatch Logs User Guide*.

Specifying event bus log level

You can specify the types of event processing steps which EventBridge logs to the selected log destinations.

Choose from the following levels of detail to include in logs. The log level applies to all log destinations specified for the event bus. Each log level includes the steps of the previous log levels.

- **OFF** – EventBridge does not send any logs. This is the default.
- **ERROR** – EventBridge sends any logs related to errors generated during event processing and target delivery.
- **INFO** – EventBridge sends any logs related to errors, as well as major steps performed during event processing.
- **TRACE** – EventBridge sends any logs generated during all steps in the event processing.

The following table lists the event processing steps included in each log level.

Step	TRACE	INFO	ERROR	OFF
Event Ingested	x	x		
Event Ingestion Failed	x	x	x	
Event Received	x			
Invocation Attempt Started	x			
Invocation Attempt Permanent Failure	x	x	x	
Invocation Attempt Retry-able Failure	x	x	x	
Invocation Attempt Succeeded	x			
Invocation Attempt Throttled	x	x	x	
Invocation DLQ	x	x	x	
Invocation Failed	x	x	x	
Invocation Started	x	x		
Invocation Succeeded	x	x		
Invocation Throttle Started	x	x	x	
No Rules Matched	x	x		
Rule Matched	x	x		
Rule Matching Started	x			

Including detail data in event bus logs

You can specify for EventBridge to include more granular information in the logs it generates. This data can be useful for troubleshooting and debugging. If you select this option, EventBridge includes this data in the relevant records for all the specified log destinations.

Detail information includes the following fields:

- `event_detail`: The details of the event itself.
- `target_input`: The request EventBridge sends to the target.
- `target_properties`:

Truncating data in event bus logs

Due to log destination constraints, EventBridge limits log records to 1 MB. If a log record exceeds this limit, EventBridge truncates the record by removing the following fields in the following order:

- `target_input`
- `target_properties`
- `target_response_body`

EventBridge removes the `event_detail` field from the following log record types if necessary:

- `EVENT_RECEIVED`
- `EVENT_INGESTED`
- `EVENT_INGESTED_FAILED`
- `RULE_MATCH_STARTED`

If truncation is necessary, EventBridge removes the entire field.

If EventBridge does truncate fields in the event, the `dropped_fields` field includes a list of the excised data fields.

Error reporting in event bus logs

EventBridge also includes error data, where available, in steps that represent failure states. These steps include:

- EVENT_INGEST_FAILURE
- INVOCATION_THROTTLE_START
- INVOCATION_ATTEMPT_THROTTLE
- INVOCATION_ATTEMPT_RETRYABLE_FAILURE
- INVOCATION_ATTEMPT_PERMANENT_FAILURE
- INVOCATION_FAILURE
- INVOCATION_DLQ

What Amazon EventBridge logs for event buses

Understanding how EventBridge processes events can aid you in using logs to troubleshoot or debug event bus issues.

If logging is enabled, EventBridge generates multiple log records as an event is processed.

Here are the main steps EventBridge performs when processing an event:

- An event is sent to an event bus

  EventBridge generates logs for events sent from partner sources, replayed from archives, or sent using `PutEvents`, whether or not they match any rules.
- EventBridge determines if the event matches any rules on the bus.

  If the event matches one or more rules, EventBridge proceeds to the next step.

  If an AWS event doesn't match any rules, EventBridge discards the event and does not generate any logs.
- EventBridge invokes the target.

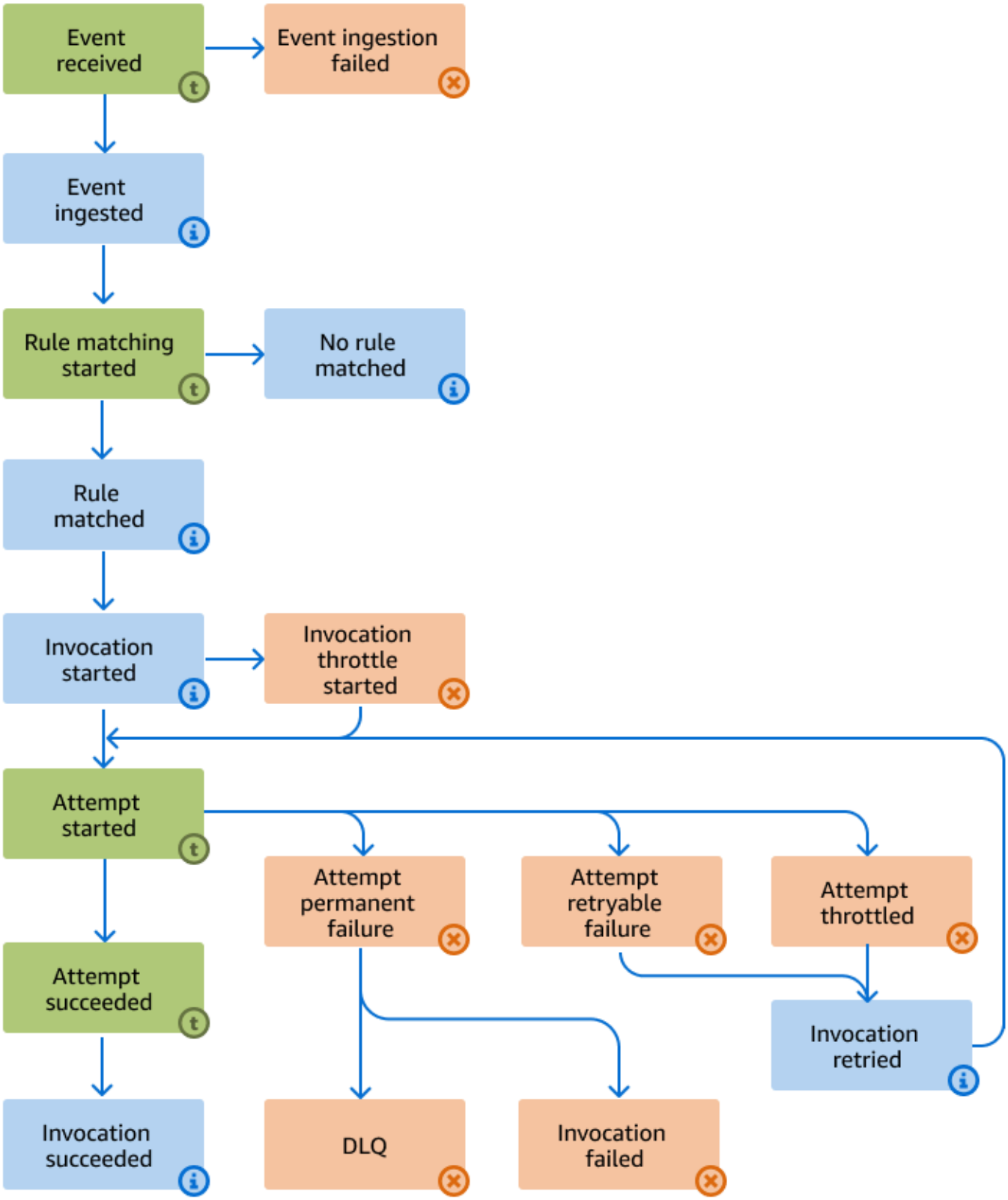
  EventBridge retries invoking the target as necessary until:

  - The event has been successfully delivered.
  - Event delivery fails, such as if the retry policy expires or a permanent failure occurs.

    If delivery fails, EventBridge sends the event to a dead-letter queue (DLQ) if one is specified, or drops the event if no DLQ is specified.

The diagram below presents a detailed view of the event processing flow, with all possible steps represented, along with the log level of each step.

For a complete list of steps, see Specifying log level.


Event received
Event ingestion failed
Event ingested
Rule matching started
No rule matched
Rule matched
Invocation started
Invocation throttle started
Attempt started
Attempt permanent failure
Attempt retryable failure
Attempt throttled
Attempt succeeded
Invocation retried
Invocation succeeded
DLQ
Invocation failed

Amazon EventBridge event bus log schema

The following reference details the schema for EventBridge event bus log records. Each record represent a step EventBridge performs processing a specific event.

For more information, see Logging event buses .

```
{
    "resource_arn": "arn:aws:events:region:account:event-bus/bus-name",
    "request_id": "guid",
    "event_id": "guid",
    "invocation_id": "guid",
    "message_timestamp_ms": "date_time",
    "message_type": "step",
    "log_level": "TRACE | INFO | ERROR",
    "details": {
      },
    "error": {
        "http_status_code": code,
        "error_message": "error_message",
        "aws_service": "service_name",
        "request_id": "service_request_id"
    }
}
```

resource_arn

The Amazon Resource Name (ARN) for the event bus.

request_id

The ID of the request.

event_id

The ID of the event being processed.

invocation_id

The ID of the invocation for the event.

message_timestamp_ms

The date and time the log event was emitted.

Unit: millisecond

message_type

The event processing step for which the log record was generated.

For more information on the steps EventBridge performs when processing an event, see ???.

Valid values:

- `EVENT_INGEST_FAILURE`
- `EVENT_INGEST_SUCCESS`
- `EVENT_RECEIPT`
- `INVOCATION_ATTEMPT_PERMANENT_FAILURE`
- `INVOCATION_ATTEMPT_RETRYABLE_FAILURE`
- `INVOCATION_ATTEMPT_START`
- `INVOCATION_ATTEMPT_SUCCESS`
- `INVOCATION_ATTEMPT_THROTTLE`
- `INVOCATION_DLQ`
- `INVOCATION_FAILURE`
- `INVOCATION_START`
- `INVOCATION_SUCCESS`
- `INVOCATION_THROTTLE_START`
- `NO_STANDARD_RULES_MATCHED`
- `RULE_MATCH`
- `RULE_MATCH_START`

log_level

The level of detail specified for the event bus log.

Valid values: `ERROR` | `INFO` | `TRACE`

For more information, see ???.

details

Contains step details, based on the step detail type.

The fields listed below are returned for the following message types:

- EVENT_INGEST_SUCCESS
- EVENT_INGEST_FAILURE
- EVENT_RECEIPT
- RULE_MATCH_START

```
{
  "caller_account_id": "account_id",
  "source_time_ms": date_time,
  "source": "source",
  "detail_type": " type",
  "resources": [],
  "event_detail": "{}"
}
```

The fields listed below are returned for the following message type:

- RULE_MATCH

```
{
  "rule_arn": "ARN",
  "target_arns": [
    "ARN"
  ],
  "invocation_ids": [
    "guid"
  ]
}
```

The fields listed below are returned for the following message types:

- INVOCATION_ATTEMPT_START
- INVOCATION_START
- INVOCATION_THROTTLE_START

```
{
  "rule_arn": "ARN",
  "role_arn": "ARN",
  "target_arn": "ARN",
  "attempt_count": Integer,
```

```
  "target_input": "string",
  "target_properties": "string"
}
```

The fields listed below are returned for the following message types:

- INVOCATION_DLQ
- INVOCATION_FAILURE
- INVOCATION_SUCCESS

```
{
  "rule_arn": "ARN",
  "role_arn": "ARN",
  "target_arn": "ARN",
  "target_input": "string",
  "target_properties": "string",
  "total_attempts": Integer,
  "final_invocation_status": "status",
  "ingestion_to_start_latency_ms": Integer,
  "ingestion_to_complete_latency_ms": Integer,
  "ingestion_to_success_latency_ms": Integer,
  "target_duration_ms": Integer,
  "target_response_body": "string"
}
```

The `ingestion_to_start_latency_ms` and `ingestion_to_complete_latency_ms` are only included in the first invocation attempt. The `ingestion_to_success_latency_ms` field is only included for successful invocations.

The fields listed below are returned for the following message types:

- INVOCATION_ATTEMPT_PERMANENT_FAILURE
- INVOCATION_ATTEMPT_RETRYABLE_FAILURE
- INVOCATION_ATTEMPT_SUCCESS
- INVOCATION_ATTEMPT_THROTTLE

```
{
  "rule_arn": "ARN",
  "role_arn": "ARN",
  "target_arn": "ARN",
  "attempt_type": "FIRST | THROTTLE | RETRY",
```

```
  "attempt_count": Integer,
  "invocation_status": "status",
  "target_duration_ms": Integer,
  "target_response_body": "string"
}
```

dropped_fields

A list of any data fields EventBridge has truncated to keep the record below the 1 MB size limitation.

EventBridge does not include this field if it has truncated any detail fields.

For more information, see ???.

error

Contains information for any error generated during this step. For errors, EV always includes the following fields:

- `error_message`
- `aws_service`

And the following fields if available:

- `request_id`
- `http_status_code`

If no error was generated during this step, EventBridge does not include this field in the log record.

http_status_code

The HTTP status code returned by the called service.

error_message

The error message returned by the called service.

aws_service

The name of the service called.

request_id

The request ID for this request from the called service.

AWS services and tools integrations with Amazon EventBridge

Amazon EventBridge works with other AWS services and tools to process events or invoke a resource as the target of a rule. For more information about EventBridge integrations with other AWS services and tools, see the following:

Topics

- Including Amazon EventBridge resources in AWS CloudFormation stacks
- Kafka connector for Amazon EventBridge
- Using Amazon EventBridge with Interface VPC endpoints
- Amazon EventBridge integration with AWS X-Ray

Including Amazon EventBridge resources in AWS CloudFormation stacks

CloudFormation enables you to configure and manage your AWS resources across accounts and regions in a centralized and repeatable manner by treating infrastructure as code. CloudFormation does this by letting you create *templates*, which define the resources you want to provision and manage. These resources can include EventBridge artifacts such as event buses and rules, pipes, schemas, and schedules, among others. Use these resources to include EventBridge functionality in the technology stacks you provision and manage through CloudFormation.

Amazon EventBridge resources available in AWS CloudFormation

EventBridge provides resources for use in CloudFormation templates in the following resource namespaces:

- AWS::Events

  Template examples include:

  - Create an API destination for PagerDuty
  - Create an API destinatio for Slack
  - Create a connection with ApiKey authorization parameters
  - Create a connection with OAuth authorization parameters
  - Create a global endpoint with event replication
  - Deny policy using multiple principals and actions
  - Grant permission to an organization using a custom event bus
  - Create a cross-Region rule
  - Create a rule that includes a dead-letter queue for a target
  - Regularly invoke a Lambda function
  - Invoke Lambda function in response to an event
  - Notify a topic in response to a log nntry
- AWS::EventSchemas
- AWS::Pipes

  Template examples include:

  - Create a pipe with an event filter

- AWS::Scheduler

Generating Amazon EventBridge resource definitions for AWS CloudFormation templates

As an aid to help you jumpstart developing CloudFormation templates, the EventBridge console enables you to create CloudFormation templates from the existing event buses, rules, and pipes in your account.

- ???
- ???
- ???

Bringing the default event bus under CloudFormation management

Because EventBridge provisions the default event bus into your account automatically, you cannot create it using a CloudFormation template, as you normally would for any resource you wanted to include in a CloudFormation stack. To include the default event bus in a CloudFormation stack, you must first *import* it into a stack. Once you have imported the default event bus into a stack, you can then update the event bus properties as desired.

For more information, see ???

Managing CloudFormation stack events using EventBridge

Beyond including EventBridge resources in your CloudFormation stacks, you can use EventBridge to manage the events generated by CloudFormation stacks themselves. CloudFormation sends events to EventBridge whenever a create, update, delete, or drift-detection operation is performed on a stack. CloudFormation also sends events to EventBridge for status changes to stack sets and stack set instances. You can use EventBridge rules to route events to your defined targets.

For more information, see Managing CloudFormation events using EventBridge in the *CloudFormation User Guide*.

Kafka connector for Amazon EventBridge

The Kafka sink connector for EventBridge allows you convert records from one or more Kafka topics into events, and send those events to the event bus of your choice.

> **Note**
>
> This connector publishes to a Custom Event Bus - Classic. To bring Kafka records onto a Custom Event Bus in Avro or Protobuf, publish them with `PutRawEvents` and a Confluent or Glue schema registry; see the section called “Supported content types”.

The connector includes the following capabilities:

- Customizable mapping of Kafka records to event types.

  You can customize the mapping of Kafka topic names to the event type, including using JsonPath expressions. This enables you to configure the connector to consume from multiple Kafka topics and filter the events sent to the specified event bus.
- Offload large event payloads to Amazon S3.

  Kafka topics can contain records exceeding the size limit of PutEvents. You can configure the connector to offload events to Amazon S3 prior to calling PutEvents.
- Support for dead-letter topics.
- Schema registry support for Avro and Protocol Buffers (Protobuf)

The Kafka Connector for Amazon EventBridge is available on GitHub. For detailed instruction on installing and configuring the connector using Amazon MSK Connect, see Set up EventBridge Kafka sink connector for Amazon MSK Connect in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Using Amazon EventBridge with Interface VPC endpoints

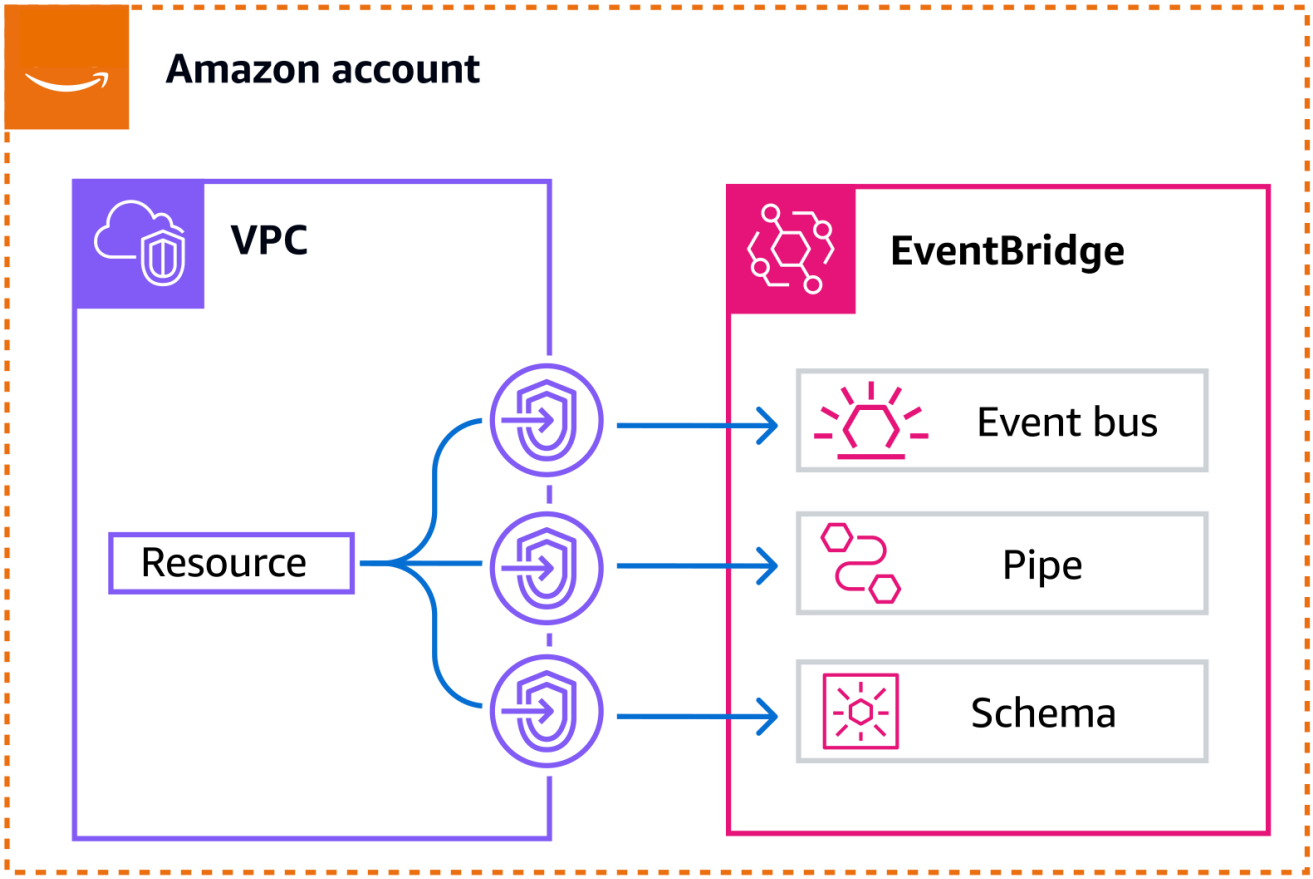
If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a private connection between your VPC and EventBridge. Your resources on your VPC can use this connection to communicate with EventBridge.

> **Note**
>
> The Custom Event Bus has its own endpoint service names: `com.amazonaws.region.eventsv2` and `com.amazonaws.region.eventsv2-fips`. Create a separate interface endpoint for them; the `events` endpoint on this page

serves Custom Event Bus - Classic only. See the section called "Names, endpoints, and permissions".

With a VPC, you have control over your network settings, such as the IP address range, subnets, route tables, and network gateways. To connect your VPC to EventBridge, you define an *interface VPC endpoint* for EventBridge. The endpoint provides reliable, scalable connectivity to EventBridge without requiring an internet gateway, network address translation (NAT) instance, or VPN connection. For more information, see What is Amazon VPC in the *Amazon VPC User Guide*.

Interface VPC endpoints are powered by AWS PrivateLink, which enables private communication between AWS services using an elastic network interface with private IP addresses. For more information, see AWS PrivateLink and VPC endpoints.



When you use a private interface VPC endpoint, custom events your VPC sends to EventBridge use that endpoint. EventBridge then sends those events to other AWS services based on the rules

and targets that you've configured. Once events are sent to another service you can receive them through either the public endpoint or a VPC endpoint for that service. For example, if you create a rule to send events to an Amazon SQS queue, you can configure an interface VPC endpoint for Amazon SQS to receive messages from that queue in your VPC without using the public endpoint.

Creating a VPC endpoint for EventBridge

To use EventBridge with your VPC, create an interface VPC endpoint for EventBridge and choose the appropriate EventBridge service name. For more information, see Creating an Interface Endpoint in the *Amazon VPC User Guide*.

- **Event buses**

  Service name: **com.amazonaws.*region*.events**

  Event bus FIPS endpoints also support VPC endpoints. For a complete list of FIPS endpoints, see EventBridge endpoints and quotas in the *AWS General Reference.*.

  Service name: **com.amazonaws.*region*.events-fips**

- **Pipes**

  Service name: **com.amazonaws.*region*.pipes**

  EventBridge Pipes supports endpoints for all pipe API operations.

  Pipes FIPS endpoints also support VPC endpoints. For a complete list of FIPS endpoints, see EventBridge Pipes endpoints and quotas in the *AWS General Reference.*.

  Service name: **com.amazonaws.*region*.pipes-fips**

  You can also use a VPC endpoint to fulfill networking requirements for Pipes Apache Kafka and Amazon MQ sources.

  Service name: **com.amazonaws.*region*.pipes-data**

  For more information, refer to the following:

  - Apache Kafka network configuration
  - Amazon MSK network configuration
  - Amazon MQ network configuration

> **Note**
>
> VPC endpoints to **pipes-data** do not support VPC Endpoint resource policies.
> VPC endpoints to **pipes** and **pipes-fips** do support VPC Endpoint resource policies that allow you to:
>
> - Deny access to specific Pipe APIs.
> - Limit access on some APIs to specific Pipes by ARN using the IAM **Resource** condition key.

- **Schemas**

  Service name: **com.amazonaws.*region*.schema**

  EventBridge supports endpoints for all schema API operations.

Amazon EventBridge integration with AWS X-Ray

You can use AWS X-Ray to trace events that pass through EventBridge. EventBridge passes the original trace header to the target so that target services can track, analyze, and debug.

> **Note**
>
> This page applies to Custom Event Bus - Classic. A Custom Event Bus subscriber propagates a trace header to Step Functions targets through `StepFunctionsParameters.TraceHeader` and to Amazon SQS targets through `MessageSystemAttributes`; see the section called "Targets".

EventBridge can pass a trace header for an event only if the event came from a `PutEvents` request that passed the trace context. X-Ray doesn't trace events that originate from third-party partners, scheduled events, or AWS services, and these event sources don't appear on your X-Ray service map.

X-Ray validates trace headers, and trace headers that aren't valid are dropped. However, the event is still processed.

> **Important**
>
> The trace header is **not** available on the event that's delivered to the invocation target.
>
> - If you have an event archive, the trace header isn't available on archived events. If you replay archived events, the trace header isn't included.
> - If you have a dead-letter queue (DLQ), the trace header is included in the `SendMessage` request that sends the event to the DLQ. If you retrieve events (messages) from the DLQ by using `ReceiveMessage`, the trace header associated with the event is included on the Amazon SQS message attribute, but it isn't included in the event message.

For information about how an EventBridge event node connects source and target services, see Viewing source and targets in the X-Ray service map in the *AWS X-Ray Developer Guide*.

You can pass the following trace header information through EventBridge:

- **Default HTTP header** – The X-Ray SDK automatically populates the trace header as the `X-Amzn-Trace-Id` HTTP header for all invocation targets. To learn more about the default HTTP header, see Tracing header in the *AWS X-Ray Developer Guide*..
- **`TraceHeader` system attribute** – `TraceHeader` is a PutEventsRequestEntry attribute reserved by EventBridge to carry the X-Ray trace header to a target. If you also use `PutEventsRequestEntry`, `PutEventsRequestEntry` overrides the HTTP trace header.

> **Note**
>
> The trace header doesn't count towards the `PutEventsRequestEntry` event size. For more information, see Calculating PutEvents event entry size.

The following video demonstrates the use X-Ray and EventBridge together: Using AWS X-Ray for tracing

Monitoring Amazon EventBridge

EventBridge sends metrics to Amazon CloudWatch every minute for everything from the number of matched events to the number of times a target is invoked by a rule.

The following video reviews monitoring and auditing EventBridge behavior through CloudWatch: Monitoring and auditing events

Best-effort CloudWatch metrics delivery

CloudWatch metrics are delivered on a best-effort basis. Most EventBridge operations result in a data point being sent to CloudWatch. The completeness and timeliness of metrics are not guaranteed. A data point for a given minute might be delayed before being available through CloudWatch, or it might not be delivered at all. CloudWatch metrics give you an idea of the nature of activity in near-real time. They are not meant to be a complete accounting of all operations.

Topics

- EventBridge metrics
- Dimensions for EventBridge metrics
- Best practices for monitoring event delivery in Amazon EventBridge

This section describes Custom Event Bus - Classic. For the Custom Event Bus, which publishes its metrics in the `AWS/EventsV2` namespace and its subscriber logs through CloudWatch vended logs, see the section called “Observability”.

EventBridge metrics

The AWS/Events namespace includes the following metrics.

For the metrics that use Count as a unit, Sum and SampleCount tend to be the most useful statistics.

Metrics that specify only the RuleName dimension refer to the default event bus. Metrics that specify both the EventBusName and RuleName dimensions refer to a Custom Event Bus - Classic.

Metric	Description	Dimensions	Units
DeadLetterInvocations	The number of times a rule's target isn't invoked in response to an event. This includes invocations that would result in running the same rule again, causing an infinite loop.	RuleName	Count
Events	The number of partner events ingested by EventBridge.	EventSourceName	Count
FailedInvocations	The number of invocations that failed permanently. This doesn't include invocations that are retried or invocations that succeeded after a retry attempt. It also doesn't count failed invocations that are counted in DeadLetterInvocations . **Note** EventBridge only sends this metric to CloudWatch if it isn't zero.	RuleName	Count
Invocations	The number of times a target is invoked by a rule in response to an event. This includes successful and failed invocations, but doesn't include throttled or retried attempts until they fail permanently. It doesn't include DeadLetterInvocations .	None, RuleName	Count

Metric	Description	Dimensions	Units
	Note EventBridge only sends this metric to CloudWatch if it isn't zero.		
`InvocationAttempts`	Number of times EventBridge attempted invoking a target.	EventBusName, None, RuleName	Count
`InvocationsCreated`	The total number of invocations created in response to each event. This metric is often used to monitor utilization of the **Invocations throttle limit in transactions per second** EventBridge service quota.	None	Count
`InvocationsFailedToBeSentToDlq`	The number of invocations that couldn't be moved to a dead-letter queue. Dead-letter queue errors occur due to permissions errors, unavailable resources, or size limits. **Note** EventBridge only sends this metric to CloudWatch if it isn't zero.	RuleName	Count
`IngestionToInvocationCompleteLatency`	The time taken from event ingestion to completion of the first invocation attempt.	EventBusName, None, RuleName	Milliseconds

Metric	Description	Dimensions	Units
IngestionToInvocationSuccessLatency	The time taken from event ingestion to successful target delivery, using the invocation end time as cutoff. This metric is only emitted after the first successful delivery attempt to the target.	EventBusName, None, RuleName	Milliseconds
IngestiontoInvocationStartLatency	The time to process events, measured from when an event is ingested by EventBridge to the first invocation of a target.	EventBusName, None, RuleName	Milliseconds
InvocationsSentToDlq	The number of invocations that are moved to a dead-letter queue. **Note** EventBridge only sends this metric to CloudWatch if it isn't zero.	RuleName	Count
MatchedEvents	If EventBusName or EventSourceName is specified, the number of events that matched with any rule. If RuleName is specified, the number of events that matched with a specific rule.	EventBusName, EventSourceName, RuleName	Count
RetryInvocationAttempts	Number of times target invocation has been retried. **Note** EventBridge only sends this metric to CloudWatch if it isn't zero.	EventBusName, None, RuleName	Count

Metric	Description	Dimensions	Units
SuccessfulInvocationAttempts	Number of times target was successfully invoked.	EventBusName, None, RuleName	Count
ThrottledRules	The number of times rule execution was throttled. Invocations for those rules may be delayed. For more information, see **Invocations throttle limit in transactions per second** in ???.	EventBusName, None, RuleName	Count
TriggeredRules	The number of rules that have run and matched with any event. You won't see this metric in CloudWatch until a rule is triggered.	EventBusName, None, RuleName	Count
EventBusEncryptionStarted	The number of times a re-encryption operation has started for an event bus. Updating the event bus configuration triggers a re-encryption of the static configuration stored by EventBridge.	EventBusName	Count
EventBusEncryptionCompleted	The number of times a re-encryption operation has completed successfully for an event bus. Updating the event bus configuration triggers a re-encryption of the static configuration stored by EventBridge.	EventBusName	Count

Metric	Description	Dimensions	Units
EventBusEncryptionFailed	The number of times a re-encryption operation has failed for an event bus. Updating the event bus configuration triggers a re-encryption of the static configuration stored by EventBridge. **Note** EventBridge only sends this metric to CloudWatch if it isn't zero.	EventBusName	Count

EventBridge PutEvents metrics

The AWS/Events namespace includes the following metrics pertaining to the PutEvents API requests.

For the metrics that use Count as a unit, Sum and SampleCount tend to be the most useful statistics.

Metric	Description	Dimensions	Units
PutEventsApproximateCallCount	Approximate number of received PutEvents requests.	None	Count
PutEventsApproximateFailedCount	Approximate number of failed PutEvents requests.	None	Count
PutEventsApproxima	Approximate number of successful PutEvents requests.	None	Count

Metric	Description	Dimensions	Units
teSuccess Count			
PutEvents ApproximateThrottledCount	Number of PutEvents requests rejected due to throttling.	None	Count
PutEvents EntriesCount	The number of event entries contained in a PutEvents request.	None	Count
PutEvents FailedEntriesCount	The number of event entries contained in a PutEvents request that failed to be ingested.	None	Count
PutEvents Latency	The time taken per PutEvents request.	None	Milliseconds
PutEvents RequestSize	The size of the PutEvents request.	None	Bytes

EventBridge PutPartnerEvents metrics

The AWS/Events namespace includes the following metrics pertaining to the PutPartnerEvents API requests.

> **Note**
>
> EventBridge only includes metrics pertaining to PutPartnerEvents requests in SaaS partner accounts that send events. For more information, see ???

For the metrics that use Count as a unit, Sum and SampleCount tend to be the most useful statistics.

Metric	Description	Dimensions	Units
PutPartnerEventsApproximateCallCount	Approximate number of received PutPartnerEvents requests.	None	Count
PutPartnerEventsApproximateFailedCount	Approximate number of failed PutPartnerEvents requests.	None	Count
PutPartnerEventsApproximateThrottledCount	Number of PutPartnerEvents requests rejected due to throttling.	None	Count
PutPartnerEventsApproximateSuccessCount	Approximate number of successful PutPartnerEvents requests.	None	Count
PutPartnerEventsEntriesCount	The number of event entries contained in a PutPartnerEvents request.	None	Count
PutPartnerEventsFa	The number of event entries contained in a PutPartnerEvents request that failed to be ingested.	None	Count

Metric	Description	Dimensions	Units
iledEntriesCount			
PutPartnerEventsLatency	The time taken per PutPartnerEvents request.	None	Milliseconds

Dimensions for EventBridge metrics

EventBridge metrics have *dimensions*, or sortable attributes, which are listed below.

Dimension	Description
EventBusName	Filters the available metrics by event bus name.
EventSourceName	Filters the available metrics by partner event source name.
RuleName	Filters the available metrics by rule name.

Best practices for monitoring event delivery in Amazon EventBridge

To ensure that the business logic in your event-driven applications executes reliably, it is essential to monitor your event delivery behavior. EventBridge provides metrics that enable you to monitor, detect, and mitigate issues early to ensure reliable event delivery. These metrics include:

- Counter-based metrics, such as InvocationAttempts, SuccessfulInvocationAttempts, RetryInvocationAttempts and FailedInvocations, to enable you to observe target throttling, and calculate error rates.
- Latency-based metrics, such as IngestionToInvocationSuccessLatency, to provide insights into event delivery and delays.

These metrics allow you to monitor the health of your event-driven architectures, and to understand and mitigate event delivery issues caused by underperforming, undersized, or

unresponsive targets. For example, a permanently under-scaled or throttled target can lead to excessive retries, delays in event delivery, and permanent delivery failures.

We recommend you combine multiple metrics to get a holistic overview, and closely monitor them. Setting up appropriate alarms and dashboards enables you to address persistent issues early.

For information on specific metrics, see EventBridge metrics.

Detecting event delivery failures

EventBridge includes metrics you can configure to report target invocations--that is, event delivery attempts--per rule.

We recommend you monitor the following metrics at the rule level:

- `InvocationAttempts` to observe the total number of times EventBridge attempts to invoke the target, including event delivery retries.
- `SuccessfulInvocationAttempts` for the number of invocation attempts where EventBridge successfully delivered the event to the target.
- `RetryInvocationAttempts` for the number attempts that represent event delivery retries.

  An increase in `RetryInvocationAttempts` may be an early indication of an undersized target.

In addition, since increased retry attempts can be a first sign of delivery issues, we also recommend creating a single metric that tracks the percentage of successful target invocations to all target invocations. For example, in CloudWatch you can use metric math to create such a metric, called `SuccessfulInvocationRate`, using the following formula:

`SuccessfulInvocationRate = SuccessfulInvocationAttempts / InvocationAttempts`

Then, depending on your requirements, you can configure CloudWatch Alarms to create notifications when a certain threshold is hit.

Although an occasional decrease of `SuccessfulInvocationRate` due to temporary traffic spikes or invocation errors can be considered normal, a constant mismatch is an indication of a misconfigured target and needs to be addressed as part of the shared responsibility model.

For more information on metric math, see Using math expressions with CloudWatch metrics in the *Amazon CloudWatch User Guide*.

By default, EventBridge retries delivering an event for 24 hours and up to 185 times. After EventBridge exhausts these retry attempts, EventBridge either drops the event, or sends it to a dead-letter queue if one has been specified. For more information, see Retrying event delivery. To avoid losing events if they fail to be delivered, we recommend you configure a dead-letter queue for each rule target. For more information, For more information, see Using dead-letter queues.

Events that EventBridge fails to deliver to the specified target are reported in the `FailedInvocations` metric and the `InvocationsSentToDlq` metric if you have configured a dead-letter queue for the target. If your application is experiencing a large number of `FailedInvocations` or `InvocationsSentToDlq` reports, we recommend you investigate if the target is properly scaled and able to receive the given traffic.

Detecting event delivery delays

EventBridge also provides a metric that lets you observe the end-to-end latency--the time it takes from event ingestion to successful delivery to the target. This can be achieved with the `IngestionToInvocationSuccessLatency` metric. This metric surfaces effects from retries and delayed delivery, for example due to timeouts and slow responses from targets. `IngestionToInvocationSuccessLatency` includes the time the target takes to successfully respond to event delivery. This allows you to monitor the end-to-end latency between EventBridge and your target, and detect performance variations and degradations of targets, even when there is no target throttling or errors.

Troubleshooting Amazon EventBridge

You can use the topics in this section to troubleshoot Amazon EventBridge issues.

> **Tip**
>
> To verify that your event patterns match the expected events, you can use the EventBridge Sandbox to test event patterns against sample events. For more information, see ???.

Topics

- My rule ran but my Lambda function wasn't invoked
- I just created or modified a rule, but it didn't match a test event
- My rule didn't run at the time I specified in the ScheduleExpression
- My rule didn't run at the time that I expected

- My rule matches AWS global service API calls but it didn't run
- The IAM role associated with my rule is being ignored when the rule runs
- My rule has an event pattern that is supposed to match a resource, but no events match
- My event's delivery to the target was delayed
- Some events were never delivered to my target
- My rule ran more than once in response to one event
- Preventing infinite loops
- My events are not delivered to the target Amazon SQS queue
- My rule runs, but I don't see any messages published into my Amazon SNS topic
- My Amazon SNS topic still has permissions for EventBridge even after I deleted the rule associated with the Amazon SNS topic
- Which IAM condition keys can I use with EventBridge?
- How can I tell when EventBridge rules are broken?

My rule ran but my Lambda function wasn't invoked

One reason your Lambda function might not run is if you don't have the right permissions.

To check your permissions for your Lambda function

1. Using the AWS CLI, run the following command with your function and your AWS Region:

   ```
   aws lambda get-policy --function-name MyFunction --region us-east-1
   ```

   You should see the following output.

   JSON

   ```
   {
   "Version":"2012-10-17",
   "Statement": [
   {
   "Condition": {
   "ArnLike": {
   "AWS:SourceArn": "arn:aws-eusc:events:eusc-de-east-1:123456789012:rule/rule-name"
   ```

```
}
},
"Action": "lambda:InvokeFunction",
"Resource": "arn:aws-eusc:lambda:eusc-de-
east-1:123456789012:function:function-name",
"Effect": "Allow",
"Principal": {
"Service": "events.amazonaws.com"
},
"Sid": "MyId"
}
],
"Id": "default"
}
```

2. If you see the following message.

```
A client error (ResourceNotFoundException) occurred when calling the GetPolicy
 operation: The resource you requested does not exist.
```

Or, you see the output but you can't locate events.amazonaws.com as a trusted entity in the policy, run the following command:

```
aws lambda add-permission \
--function-name MyFunction \
--statement-id MyId \
--action 'lambda:InvokeFunction' \
--principal events.amazonaws.com \
--source-arn arn:aws:events:us-east-1:123456789012:rule/MyRule
```

3. If the output contains a `SourceAccount` field, then you need to remove it. A `SourceAccount` setting prevents EventBridge from being able to invoke the function.

Note

If the policy is incorrect, you can edit the rule in the EventBridge console by removing and then adding it back to the rule. The EventBridge console then sets the correct permissions on the target.

If you're using a specific Lambda alias or version, add the `--qualifier` parameter in the `aws lambda get-policy` and `aws lambda add-permission` commands, as shown in the following command

```
aws lambda add-permission \
--function-name MyFunction \
--statement-id MyId \
--action 'lambda:InvokeFunction' \
--principal events.amazonaws.com \
--source-arn arn:aws:events:us-east-1:123456789012:rule/MyRule
--qualifier alias or version
```

I just created or modified a rule, but it didn't match a test event

When you make a change to a rule or to its targets, incoming events might not immediately start or stop matching to new or updated rules. Allow a short period of time for changes to take effect.

If events still don't match after a short period of time, check the CloudWatch metrics `TriggeredRules`, `Invocations`, and `FailedInvocations` for your rule. For more information about these metrics, see Monitoring Amazon EventBridge.

If the rule is intended to match an event from an AWS service, do one of these things:

- Use the `TestEventPattern` action to test the event pattern of your rule matches a test event. For more information, see TestEventPattern in the *Amazon EventBridge API Reference*.
- Use the **Sandbox** on the EventBridge console.

My rule didn't run at the time I specified in the `ScheduleExpression`

Make sure you have set the schedule for the rule in the UTC+0 time zone. If the `ScheduleExpression` is correct, then follow the steps under I just created or modified a rule, but it didn't match a test event.

My rule didn't run at the time that I expected

EventBridge runs rules within one minute of the start time you set. The count down to run time begins as soon as you create the rule.

> **Note**
>
> Scheduled rules have delivery type of `guaranteed` meaning events will be triggered for each expected time at least once.

You can use a cron expression to invoke targets at a specified time. To create a rule that runs every four hours on the 0th minute, you do one of the following:

- In the EventBridge console, you use the cron expression `0 0/4 * * ? *`.
- Using the AWS CLI, you use the expression `cron(0 0/4 * * ? *)`.

For example, to create a rule named `TestRule` that runs every 4 hours by using the AWS CLI, you use the following command.

```
aws events put-rule --name TestRule --schedule-expression 'cron(0 0/4 * * ? *)'
```

To run a rule every five minutes, you use the following cron expressio.

```
aws events put-rule --name TestRule --schedule-expression 'cron(0/5 * * * ? *)'
```

The finest resolution for an EventBridge rule that uses a cron expression is one minute. Your scheduled rule runs within that minute but not on the precise 0th second.

Because EventBridge and target services are distributed, there can be a delay of several seconds between the time the scheduled rule runs and the time the target service performs the action on the target resource.

My rule matches AWS global service API calls but it didn't run

AWS global services; such as, IAM and Amazon Route 53 are only available in the US East (N. Virginia) Region, so events from AWS API calls from global services are only available in that region. For more information, see Events from AWS services.

The IAM role associated with my rule is being ignored when the rule runs

EventBridge supports IAM execution roles for all target types. If your rule's IAM role appears to be ignored, verify that the role's trust policy allows `events.amazonaws.com` to assume it, and that the permission policy grants the required action for the target (for example, `lambda:InvokeFunction`, `sns:Publish`, or `sqs:SendMessage`).

For Lambda, Amazon SNS, and Amazon SQS targets, if no IAM execution role is configured on the target, EventBridge falls back to resource-based permissions on the target resource.

Make sure your regional AWS STS endpoints are enabled, so that EventBridge can use them when assuming the IAM role you provided. For more information, see Activating and Deactivating AWS STS in an AWS Region in the *IAM User Guide*.

My rule has an event pattern that is supposed to match a resource, but no events match

Most services in AWS treat a colon (:) or slash (/) as the same character in Amazon Resource Names (ARNs)., but EventBridge uses an exact match in event patterns and rules. Be sure to use the correct ARN characters when creating event patterns so that they match the ARN syntax in the event to match.

Some events, such as AWS API call events from CloudTrail, don't have anything in the resources field.

My event's delivery to the target was delayed

EventBridge tries to deliver an event to a target for up to 24 hours, except in scenarios where your target resource is constrained. The first attempt is made as soon as the event arrives in the event stream. If the target service is having problems, EventBridge automatically reschedules another delivery. If 24 hours has passed since the arrival of event, EventBridge stops trying to deliver the event and publishes the `FailedInvocations` metric in CloudWatch. We recommend that you set up a DLQ to store events that couldn't successfully be delivered to a target. For more information, see Using dead-letter queues to process undelivered events in EventBridge

Some events were never delivered to my target

If the target of an EventBridge rule is constrained for a prolonged time, EventBridge might not retry delivery. For example, if the target is not provisioned to handle the incoming event traffic and the target service is throttling requests that EventBridge makes on your behalf, then EventBridge might not retry delivery.

My rule ran more than once in response to one event

In rare cases, the same rule can run more than once for a single event or scheduled time, or the same target can be invoked more than once for a given triggered rule.

Preventing infinite loops

In EventBridge, it is possible to create a rule that leads to infinite loops, where the rule runs repeatedly. If you have a rule that causes an infinite loop, rewrite it so that the actions that the rule takes don't match the same rule.

For example, a rule that detects that ACLs have changed on an Amazon S3 bucket and then runs software to change them to a new state causes an infinite loop. One way to resolve it is to rewrite the rule so that it only matches ACLs that are in a bad state.

An infinite loop can quickly cause higher than expected charges. We recommend that you use budgeting, which alerts you when charges exceed your specified limit. For more information, see Managing Your Costs with Budgets.

My events are not delivered to the target Amazon SQS queue

If your Amazon SQS queue is encrypted, you must create a customer-managed KMS key and include the following permission section in your KMS key policy. For more information, see Configuring AWS KMS permissions.

```
{
  "Sid": "Allow EventBridge to use the key",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": [
```

```
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*"
}
```

My rule runs, but I don't see any messages published into my Amazon SNS topic

Scenario 1

You need permission for messages to be published into your Amazon SNS topic. Use the following command using the AWS CLI, replacing us-east-1 with your Region and using your topic ARN.

```
aws sns get-topic-attributes --region us-east-1 --topic-arn "arn:aws:sns:us-east-1:123456789012:MyTopic"
```

To have the correct permission, your policy attributes similar to the following.

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_policy_ID",
    "Statement": [
    {
    "Sid": "__default_statement_ID",
    "Effect": "Allow",
    "Principal": {
    "AWS": "*"
    },
    "Action": [
    "SNS:Subscribe",
    "SNS:ListSubscriptionsByTopic",
    "SNS:DeleteTopic",
    "SNS:GetTopicAttributes",
    "SNS:Publish",
    "SNS:RemovePermission",
    "SNS:AddPermission",
    "SNS:SetTopicAttributes"
    ],
```

```
    "Resource": "arn:aws-eusc:sns:eusc-de-east-1:123456789012:MyTopic",
    "Condition": {
    "StringEquals": {
    "AWS:SourceOwner": "123456789012"
    }
    }
    },
    {
    "Sid": "Allow_Publish_Events",
    "Effect": "Allow",
    "Principal": {
    "Service": "events.amazonaws.com"
    },
    "Action": "sns:Publish",
    "Resource": "arn:aws-eusc:sns:eusc-de-east-1:123456789012:MyTopic"
    }
    ]
    }
```

If you don't see `events.amazonaws.com` with `Publish` permission in your policy, first copy the current policy and add the following statement to the list of statements.

```
{\"Sid\":\"Allow_Publish_Events\",
\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"events.amazonaws.com\"},
\"Action\":\"sns:Publish\",
\"Resource\":\"arn:aws:sns:us-east-1:123456789012:MyTopic\"}
```

Then set the topic attributes by using the AWS CLI, use the following command.

```
aws sns set-topic-attributes --region us-east-1 --topic-arn "arn:aws:sns:us-
east-1:123456789012:MyTopic" --attribute-name Policy --attribute-
value NEW_POLICY_STRING
```

> **Note**
>
> If the policy is incorrect, you can also edit the rule in the EventBridge console by removing and then adding it back to the rule. EventBridge sets the correct permissions on the target.

Scenario 2

If your SNS topic is encrypted, you must include the following section in your KMS key policy.

```
{
  "Sid": "Allow EventBridge to use the key",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*"
}
```

My Amazon SNS topic still has permissions for EventBridge even after I deleted the rule associated with the Amazon SNS topic

When you create a rule with Amazon SNS as the target, EventBridge adds permission to your Amazon SNS topic on your behalf. If you delete the rule shortly after you create it, EventBridge might not remove the permission from your Amazon SNS topic. If this happens, you can remove the permission from the topic by using the `aws sns set-topic-attributes` command. For information about resource-based permissions for sending events, see Using resource-based policies for Amazon EventBridge.

Which IAM condition keys can I use with EventBridge?

EventBridge supports the AWS-wide condition keys (see IAM and AWS STS condition context keys in the *IAM User Guide*), plus the keys listed at Using IAM policy conditions in Amazon EventBridge.

How can I tell when EventBridge rules are broken?

You can use the following alarm to notify you when your EventBridge rules are broken.

To create an alarm to alert when rules are broken

1. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
2. Choose **Create Alarm**. In the **CloudWatch Metrics by Category** pane, choose **Events Metrics**.

3. In the list of metrics, select **FailedInvocations**.
4. Above the graph, choose **Statistic**, **Sum**.
5. For **Period**, choose a value, for example **5 minutes**. Choose **Next**.
6. Under **Alarm Threshold**, for **Name**, type a unique name for the alarm, for example **myFailedRules**. For **Description**, type a description of the alarm, for example **Rules aren't delivering events to targets**.
7. For **is**, choose **>=** and **1**. For **for**, enter **10**.
8. Under **Actions**, for **Whenever this alarm**, choose **State is ALARM**.
9. For **Send notification to**, select an existing Amazon SNS topic or create a new one. To create a new topic, choose **New list**. Type a name for the new Amazon SNS topic, for example: **myFailedRules**.
10. For **Email list**, type a comma-separated list of email addresses to be notified when the alarm changes to the **ALARM** state.
11. Choose **Create Alarm**.

Using EventBridge with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

For examples specific to EventBridge, see Code examples for EventBridge using AWS SDKs.

Amazon EventBridge Pipes

Amazon EventBridge Pipes connects sources to targets. Pipes are intended for point-to-point integrations between supported sources and targets, with support for advanced transformations and enrichment. It reduces the need for specialized knowledge and integration code when developing event-driven architectures, fostering consistency across your company's applications. To set up a pipe, you choose the source, add optional filtering, define optional enrichment, and choose the target for the event data.

> **Note**
>
> You can also route events using event buses. Event buses are well-suited for many-to-many routing of events between event-driven services. For more information, see ???.

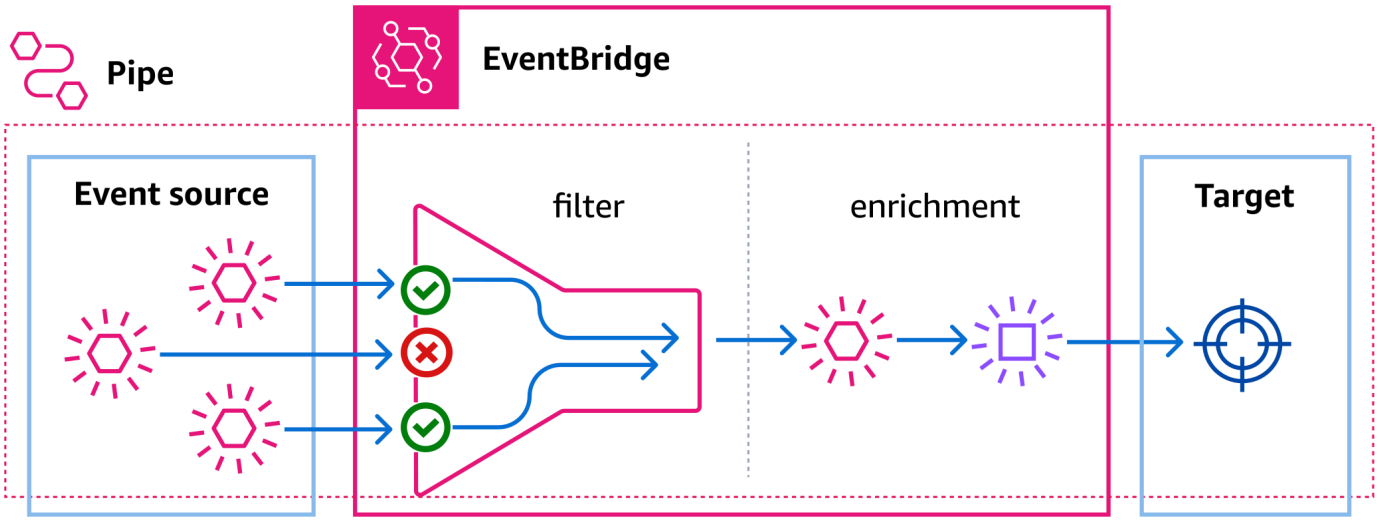
How EventBridge Pipes work

At a high level, here's how EventBridge Pipes works:

1. You create a pipe in your account. This includes:
   - Specifying one of the supported event sources from which you want your pipe to receive events.
   - Optionally, configuring a filter so that the pipe only processes a subset of the events it receives from the source.
   - Optionally, configuring an enrichment step that enhances the event data before sending it to the target.
   - Specifying one of the supported targets to which you want your pipe to send events.
2. The event source begins sending events to the pipe, and the pipe processes the event before sending it to the target.
   - If you have configured a filter, the pipe evaluates the event and only sends it to the target if it matches that filter.

     You are only charged for those events that match the filter.
   - If you have configured an enrichment, the pipe performs that enrichment on the event before sending it to the target.

If the events are batched, the enrichment maintains the ordering of the events in the batch.



For example, a pipe could be used to create an e-commerce system. Suppose you have an API that contains customer information, such as shipping addresses.

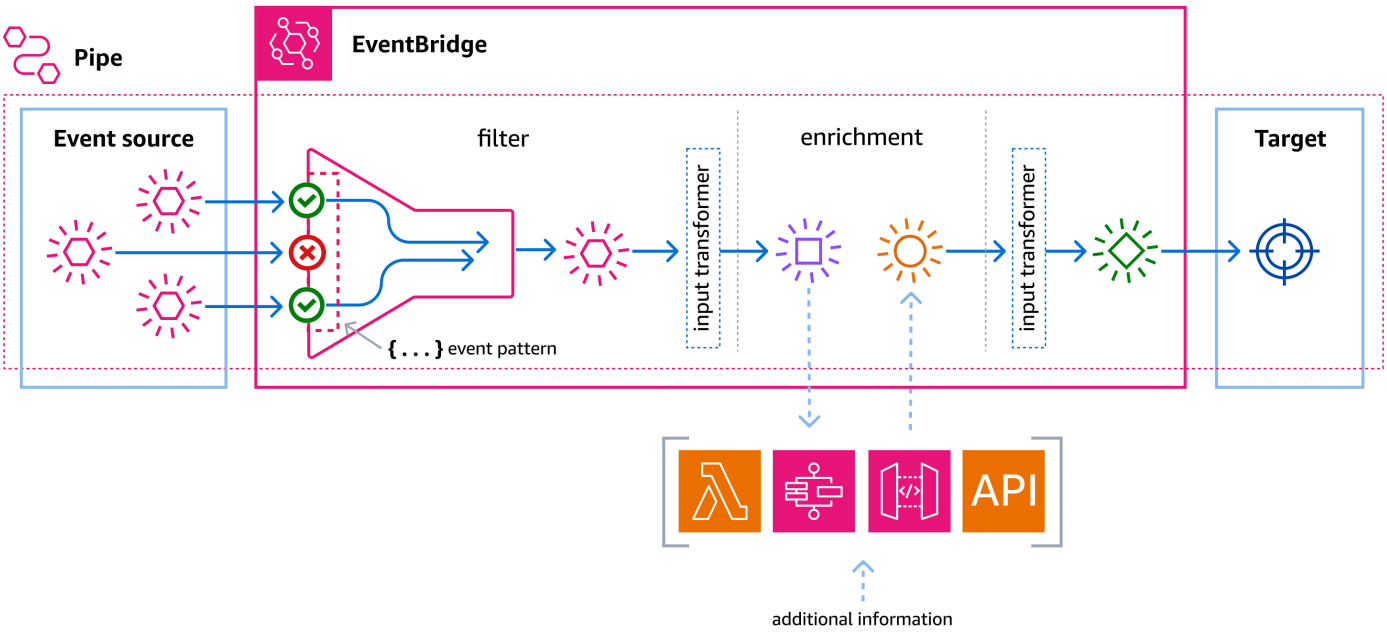
1. You then create a pipe with the following:
   - An Amazon SQS order received message queue as the event source.
   - An EventBridge API Destination as an enrichment
   - An AWS Step Functions state machine as the target
2. Then, when an Amazon SQS order received message appears in the queue, it is sent to your pipe.
3. The pipe then sends that data to the EventBridge API Destination enrichment, which returns the customer information for that order.
4. Lastly, the pipe sends the enriched data to the AWS Step Functions state machine, which processes the order.

Amazon EventBridge Pipes concepts

Here's a closer look at the basic components of EventBridge Pipes.

Pipe

A pipe routes events from a single source to a single target. The pipe also includes the ability to filter for specific events, and to perform enrichments on the event data before it is sent to the target.



Source

EventBridge Pipes receives event data from a variety of sources, applies optional filters and enrichment to that data, and sends it to a target. If a source enforces order to the events sent to pipes, that order is maintained throughout the entire process to the target.

For more information about sources, see ???.

Filters

A pipe can filter a given source's events and then process only a subset of those events. To configure filtering on a pipe, you define an event pattern the pipe uses to determine which events to send to the target.

You are only charged for those events that match the filter.

For more information, see ???.

Enrichment

With the enrichment step of EventBridge Pipes, you can enhance the data from the source before sending it to the target. For example, you might receive *Ticket created* events that don't include the full ticket data. Using enrichment, you can have a Lambda function call the `get-ticket` API for the full ticket details. The pipe can then send that information to a target.

For more information about enriching event data, see ???.

Target

After the event data has been filtered and enriched, you can send it to a specific target, such as an Amazon Kinesis stream or an Amazon CloudWatch log group. For a list of the available targets, see ???.

You can transform the data after it's enhanced and before it's sent by the pipe to the target. For more information, see ???.

Multiple pipes, each with a different source, can send events to the same target.

You can also use pipes and event buses together to send events to multiple targets. A common use case is to create a pipe with an event bus as its target; the pipe sends events to the event bus, which then sends those events on to multiple targets. For example, you could create a pipe with a DynamoDB stream for a source, and an event bus as the target. The pipe receives events from the DynamoDB stream and sends them to the event bus, which then sends them on to multiple targets according to the rules you've specified on the event bus.

Getting started: Create an Amazon EventBridge pipe

To get familiar with pipes and their capabilities, we'll use a CloudFormation template to set up an EventBridge pipe and associated components. Then we can explore various pipe features.

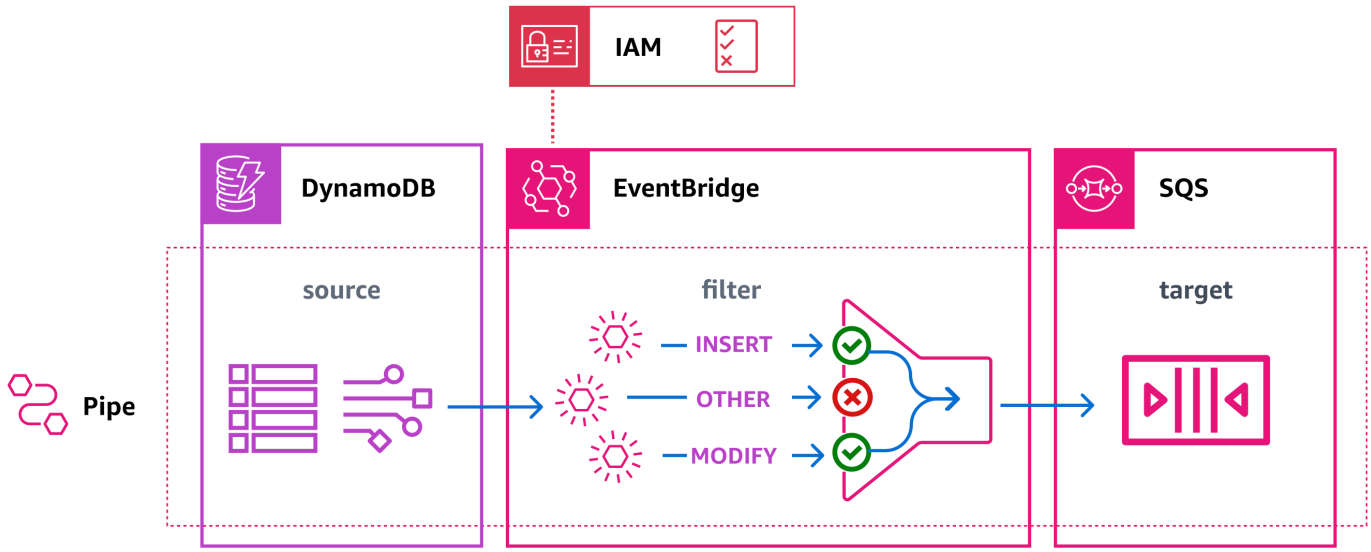
> **Tip**
>
> For a more comprehensive, hands-on learning experience, try the EventBridge Pipes Workshop. This interactive workshop walks you through building and troubleshooting a pipe that connects DynamoDB to API Gateway with a Lambda enrichment.

The template creates an EventBridge pipe that connects a stream from a DynamoDB table to an Amazon SQS queue. Every time a record is created or modified in the database table, the pipe sends the resulting event to the queue.

The deployed pipe consists of:

- A DynamoDB table (and stream) to act as the pipe source, and an Amazon SQS queue as the target.
- An execution role that grants EventBridge the necessary permissions to access the DynamoDB table and Amazon SQS queue.
- The pipe itself, which contains an event filter that selects only events generated when a table item is created (inserted) or modified.

For specific technical details of the template, see Template details.



Creating the pipe using CloudFormation

To create the pipe and its associated resources, we'll create a CloudFormation template and use it to create a stack containing a sample pipe, complete with source and target.

> **Important**
>
> You will be billed for the Amazon resources used if you create a stack from this template.

Creating the template

First, create the CloudFormation template.

1. In the Template section, click the copy icon on the **JSON** or **YAML** tab to copy the template contents.
2. Paste the template contents into a new file.
3. Save the file locally.

Creating the stack

Next, use the template you've saved to provision a CloudFormation stack.

1. Open the CloudFormation console at https://console.aws.amazon.com/cloudformation/.
2. On the **Stacks** page, from the **Create stack** menu, choose **with new resources (standard)**.
3. Specify the template:
   a. Under **Prerequisite**, choose **Choose an existing template**.
   b. Under **Specify template**, choose **Upload a template file**.
   c. Choose **Choose file**, navigate to the template file, and choose it.
   d. Choose **Next**.
4. Specify the stack details:
   a. Enter a stack name.
   b. For parameters, accept the default values or enter your own.
   c. Choose **Next**.
5. Configure the stack options:
   a. Under **Stack failure options**, choose **Delete all newly created resources**.

      > **Note**
      >
      > Choosing this option prevents you from possibly being billed for resources whose deletion policy specifies they be retained even if the stack creation fails. For more information, see `DeletionPolicy` attribute in the *CloudFormation User Guide*.

   b. Accept all other default values.

c. Under **Capabilities**, check the box to acknowledge that CloudFormation might create IAM resources in your account.

d. Choose **Next**.

6. Review the stack details and choose **Submit**.

Create the stack using CloudFormation (AWS CLI)

You can also use the AWS CLI to create the stack.

- Use the `create-stack` command.
  - Accept the default template parameter values, specifying the stack name. Use the `template-body` parameter to pass the template contents, or `template-url` to specify a URL location.

```
aws cloudformation create-stack \
  --stack-name eventbridge-rule-tutorial \
  --template-body template-contents \
  --capabilities CAPABILITY_IAM
```

  - Override the default value(s) of one or more template parameters. For example:

```
aws cloudformation create-stack \
  --stack-name eventbridge-rule-tutorial \
  --template-body template-contents \
  --parameters \
    ParameterKey=SourceTableName,ParameterValue=pipe-example-source \
    ParameterKey=TargetQueueName,ParameterValue=pipe-example-target \
    ParameterKey=PipeName,ParameterValue=pipe-with-filtering-example \
  --capabilities CAPABILITY_IAM
```

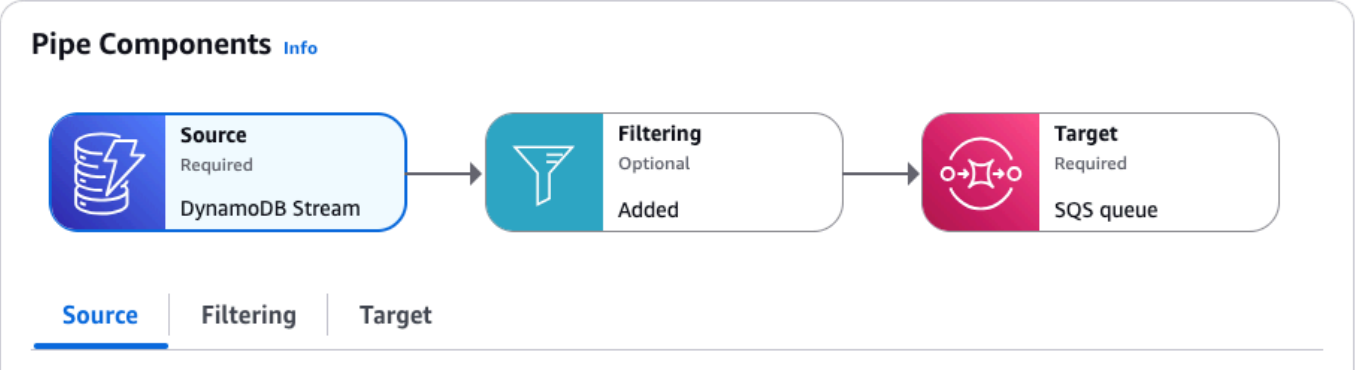
CloudFormation creates the stack. Once the stack creation is complete, the stack resources are ready to use. You can use the **Resources** tab on the stack detail page to view the resources that where provisioned in your account.

Exploring pipe capabilities

Once the pipe has been created, you can use the EventBridge console to observe pipe operation and test event delivery.

1. Open the EventBridge console at https://console.aws.amazon.com/events/home?#/pipes.
2. Choose the pipe you created.

   On the pipe detail page, the **Pipe Components** section displays the resources that make up the pipe, and contains tabs that provide more details on each component.



   You can find the execution role we created for the pipe on the **Settings** tab, in the **Permissions** section.

Examining the pipe filter

Before we test the pipe operation, let's examine the filter we've specified to control which events are sent to the target. The pipe will only send events that match the filter criteria to the target; all others are discarded. In this case, we only want events generated when table entries are created or modified being sent to the Amazon SQS queue.

- On the pipe detail page, under **Pipe Components**, choose the **Filtering** tab.

  We've included a filter that selects only events where the `eventName` is set to `INSERT` or `MODIFY`.

```
{
  "eventName": ["INSERT", "MODIFY"]
}
```

Sending events through the pipe

Next, we'll generate events in the pipe source to test that the pipe filtering and delivery is operating correctly. To do this, we'll create and edit an item in the DynamoDB table we specified as the pipe source.

1. On the pipe detail page, under **Pipe Components**, choose the **Source** tab.
2. Under **Source**, choose the DynamoDB stream name.

   This opens the DynamoDB console in a separate window, with the source table details displayed.
3. Choose **Explore items**.
4. Generate an `INSERT` event by creating an item in the table:

   a. Choose **Create item**.

   b. Add values for the **Album** and **Artist** attributes.

   c. Choose **Create item**.
5. Generate a `DELETE` and an `INSERT` event by editing the item:

   a. Choose the item from the list, and from the **Actions** menu, choose **Edit item**.

   b. Enter a new value for the **Album** or **Artist** attribute.

   c. Tick the box that confirms you are changing the value of the item keys, and then choose **Recreate item**.

      This results in the item being deleted and then recreated, generating a `DELETE` event, and then a new `INSERT` event.
6. Generate a `MODIFY` event by adding an attribute to the item:

   a. Choose the item from the list, and from the **Actions** menu, choose **Edit item**.

   b. From the **Add new attribute** menu, choose **Number**.

   c. For the attribute name, enter **Year**, and then enter a value for the attribute. Choose **Save and close**.

Confirming event delivery through the pipe

Finally, we'll confirm that the pipe successfully filtered and delivered the events we generated by creating and editing the table item in DynamoDB.

1. On the pipe detail page, under **Pipe Components**, choose the **Target** tab.
2. Under **Target**, choose the Amazon SQS queue name.

   This opens the Amazon SQS console in a separate window, with the target queue details displayed.
3. Choose **Send and receive messages**.
4. Under **Receive messages**, choose **Poll for messages**.

   Amazon SQS loads messages received into the queue. Click on an individual message to see its details.

   There should be three event messages in the queue:

   - Two of type `INSERT`, one generated when you first created the table item, and the other generated when you recreated the item by changing a key value.
   - One of type `MODIFY`, generated when you added an attribute to the table item.

   Notice that there isn't an event message of type `DELETE` in the queue, even though one was generated when you deleted and recreated the table item by changing a key value. The pipe filter we specified only selects on `INSERT` and `MODIFY`, so the pipe filtered out the `DELETE` event rather than deliver it to the queue.

Clean up: deleting resources

As a final step, we'll delete the stack and the resources it contains.

> **⚠ Important**
>
> You will be billed for the Amazon resources contained in the stack for as long as it exists.

1. Open the CloudFormation console at https://console.aws.amazon.com/cloudformation/.

2. On the **Stacks** page, choose the stack created from the template, and choose **Delete**, then confirm **Delete**.

   CloudFormation initiates deletion of the stack and all resources it includes.

CloudFormation template details

This template creates resources and grants permissions in your account.

Resources

The CloudFormation template for this tutorial will create the following resources in your account:

> **Important**
>
> You will be billed for the Amazon resources used if you create a stack from this template.

- `AWS::DynamoDB::Table`: A DynamoDB table that acts as the event source for the pipe.
- `AWS::SQS::Queue`: An Amazon SQS queue that acts as the target for the events flowing through the pipe.
- `AWS::IAM::Role`: An IAM execution role granting permissions to the EventBridge Pipes service in your account.
- `AWS::Pipes::Pipe`: The pipe connecting the DynamoDB table to the Amazon SQS queue.

Permissions

The template includes an `AWS::IAM::Role` resource that represents an execution role. This role grants the EventBridge Pipes service (`pipes.amazonaws.com`) the following permissions in your account.

The following permissions are scoped to the DynamoDB table and stream the template creates as the event source for the pipe:

- `dynamodb:DescribeStream`
- `dynamodb:GetRecords`

- dynamodb:GetShardIterator
- dynamodb:ListStreams

The following permission is scoped to the Amazon SQS queue the stack creates as the target of the pipe:

- sqs:SendMessage

CloudFormation template

Save the following JSON or YAML code as a separate file to use as the CloudFormation template for this tutorial.

JSON

```
{
  "AWSTemplateFormatVersion": "2010-09-09",

 "Description" : "[AWSDocs] EventBridge: pipes-get-started",

  "Parameters" : {
    "SourceTableName" : {
      "Type" : "String",
      "Default" : "pipe-example-source",
      "Description" : "Specify the name of the table to provision as the pipe
 source, or accept the default."
    },
  "TargetQueueName" : {
    "Type" : "String",
    "Default" : "pipe-example-target",
    "Description" : "Specify the name of the queue to provision as the pipe target,
 or accept the default."
  },
    "PipeName" : {
      "Type" : "String",
      "Default" : "pipe-with-filtering-example",
      "Description" : "Specify the name of the table to provision as the pipe
 source, or accept the default."
    }
},
  "Resources": {
```

```
    "PipeSourceDynamoDBTable": {
      "Type": "AWS::DynamoDB::Table",
      "Properties": {
        "AttributeDefinitions": [{
            "AttributeName": "Album",
            "AttributeType": "S"
          },
          {
            "AttributeName": "Artist",
            "AttributeType": "S"
          }

        ],
        "KeySchema": [{
            "AttributeName": "Album",
            "KeyType": "HASH"

          },
          {
            "AttributeName": "Artist",
            "KeyType": "RANGE"
          }
        ],
        "ProvisionedThroughput": {
          "ReadCapacityUnits": 10,
          "WriteCapacityUnits": 10
        },
        "StreamSpecification": {
          "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "TableName": { "Ref" : "SourceTableName" }
      }
    },
    "PipeTargetQueue": {
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": { "Ref" : "TargetQueueName" }
      }
    },
    "PipeTutorialPipeRole": {
      "Type": "AWS::IAM::Role",
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
```

```
          "Statement": [{
            "Effect": "Allow",
            "Principal": {
              "Service": "pipes.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
              "StringLike": {
                "aws:SourceArn": {
                  "Fn::Join": [
                    "",
                    [
                      "arn:",
                      { "Ref": "AWS::Partition" },
                      ":pipes:",
                      { "Ref": "AWS::Region" },
                      ":",
                      { "Ref": "AWS::AccountId" },
                      ":pipe/",
                      { "Ref": "PipeName" }
                    ]
                  ]
                },
                "aws:SourceAccount": { "Ref" : "AWS::AccountId" }
              }
            }
          }]
        },
        "Description" : "EventBridge Pipe template example. Execution role that
grants the pipe the permissions necessary to send events to the specified pipe.",
        "Path": "/",
        "Policies": [{
            "PolicyName": "SourcePermissions",
            "PolicyDocument": {
              "Version": "2012-10-17",
              "Statement": [{
                "Effect": "Allow",
                "Action": [
                  "dynamodb:DescribeStream",
                  "dynamodb:GetRecords",
                  "dynamodb:GetShardIterator",
                  "dynamodb:ListStreams"
                ],
                "Resource": [
```

```
                { "Fn::GetAtt" : [ "PipeSourceDynamoDBTable", "StreamArn" ] }
              ]
            }]
          }
        },
        {
          "PolicyName": "TargetPermissions",
          "PolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [{
              "Effect": "Allow",
              "Action": [
                "sqs:SendMessage"
              ],
              "Resource": [
                { "Fn::GetAtt" : [ "PipeTargetQueue", "Arn" ] }
              ]
            }]
          }
        }
      ]
    }
 },
   "PipeWithFiltering": {
     "Type": "AWS::Pipes::Pipe",
     "Properties": {
       "Description" : "EventBridge Pipe template example. Pipe that receives
events from a DynamoDB stream, applies a filter, and sends matching events on to an
SQS Queue.",
       "Name": { "Ref" : "PipeName" },
       "RoleArn": {"Fn::GetAtt" : ["PipeTutorialPipeRole", "Arn"] },
       "Source": { "Fn::GetAtt" : [ "PipeSourceDynamoDBTable", "StreamArn" ] },
       "SourceParameters": {
         "DynamoDBStreamParameters" : {
           "StartingPosition" : "LATEST"
        },
       "FilterCriteria" : {
         "Filters" : [ {
           "Pattern" : "{ \"eventName\": [\"INSERT\", \"MODIFY\"] }"
        }]
       }
       },
       "Target": { "Fn::GetAtt" : [ "PipeTargetQueue", "Arn" ] }
     }
```

```
    }
  }
}
```

YAML

```
AWSTemplateFormatVersion: '2010-09-09'
Description: '[AWSDocs] EventBridge: pipes-get-started'
Parameters:
  SourceTableName:
    Type: String
    Default: pipe-example-source
    Description: Specify the name of the table to provision as the pipe source, or
 accept the default.
  TargetQueueName:
    Type: String
    Default: pipe-example-target
    Description: Specify the name of the queue to provision as the pipe target, or
 accept the default.
  PipeName:
    Type: String
    Default: pipe-with-filtering-example
    Description: Specify the name of the table to provision as the pipe source, or
 accept the default.
Resources:
  PipeSourceDynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: Album
          AttributeType: S
        - AttributeName: Artist
          AttributeType: S
      KeySchema:
        - AttributeName: Album
          KeyType: HASH
        - AttributeName: Artist
          KeyType: RANGE
      ProvisionedThroughput:
        ReadCapacityUnits: 10
        WriteCapacityUnits: 10
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

```
      TableName: !Ref SourceTableName
  PipeTargetQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: !Ref TargetQueueName
  PipeTutorialPipeRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: pipes.amazonaws.com
            Action: sts:AssumeRole
            Condition:
              StringLike:
                aws:SourceArn: !Join
                  - ''
                  - - 'arn:'
                    - !Ref AWS::Partition
                    - ':pipes:'
                    - !Ref AWS::Region
                    - ':'
                    - !Ref AWS::AccountId
                    - ':pipe/'
                    - !Ref PipeName
                aws:SourceAccount: !Ref AWS::AccountId
      Description: EventBridge Pipe template example. Execution role that grants the
pipe the permissions necessary to send events to the specified pipe.
      Path: /
      Policies:
        - PolicyName: SourcePermissions
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - dynamodb:DescribeStream
                  - dynamodb:GetRecords
                  - dynamodb:GetShardIterator
                  - dynamodb:ListStreams
                Resource:
                  - !GetAtt PipeSourceDynamoDBTable.StreamArn
```

```
        - PolicyName: TargetPermissions
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - sqs:SendMessage
                Resource:
                  - !GetAtt PipeTargetQueue.Arn
  PipeWithFiltering:
    Type: AWS::Pipes::Pipe
    Properties:
      Description: EventBridge Pipe template example. Pipe that receives events from
a DynamoDB stream, applies a filter, and sends matching events on to an SQS Queue.
      Name: !Ref PipeName
      RoleArn: !GetAtt PipeTutorialPipeRole.Arn
      Source: !GetAtt PipeSourceDynamoDBTable.StreamArn
      SourceParameters:
        DynamoDBStreamParameters:
          StartingPosition: LATEST
        FilterCriteria:
          Filters:
            - Pattern: '{ "eventName": ["INSERT", "MODIFY"] }'
      Target: !GetAtt PipeTargetQueue.Arn
```

Event source permissions for Amazon EventBridge Pipes

When setting up a pipe, you can use an existing execution role, or have EventBridge create one for you with the needed permissions. The permissions EventBridge Pipes requires vary based on the source type, and are listed below. If you're setting up your own execution role, you must add these permissions yourself.

> **Note**
>
> If you're unsure of the exact well-scoped permissions required to access the source, use the EventBridge Pipes console to create a new role, then inspect the actions listed in the policy.

Topics

- DynamoDB execution role permissions

- Kinesis execution role permissions
- Amazon MQ execution role permissions
- Amazon MSK execution role permissions
- Self managed Apache Kafka execution role permissions
- Amazon SQS execution role permissions
- Enrichment and target permissions

DynamoDB execution role permissions

For DynamoDB Streams, EventBridge Pipes requires the following permissions to manage resources that are related to your DynamoDB data stream.

- `dynamodb:DescribeStream`
- `dynamodb:GetRecords`
- `dynamodb:GetShardIterator`
- `dynamodb:ListStreams`

To send records of failed batches to the pipe dead-letter queue, your pipe execution role needs the following permission:

- `sqs:SendMessage`

Kinesis execution role permissions

For Kinesis, EventBridge Pipes requires the following permissions to manage resources that are related to your Kinesis data stream.

- `kinesis:DescribeStream`
- `kinesis:DescribeStreamSummary`
- `kinesis:GetRecords`
- `kinesis:GetShardIterator`
- `kinesis:ListShards`
- `kinesis:ListStreams`

- kinesis:SubscribeToShard

To send records of failed batches to the pipe dead-letter queue, your pipe execution role needs the following permission:

- sqs:SendMessage

Amazon MQ execution role permissions

For Amazon MQ, EventBridge Pipes requires the following permissions to manage resources that are related to your Amazon MQ message broker.

- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Amazon MSK execution role permissions

For Amazon MSK, EventBridge requires the following permissions to manage resources that are related to your Amazon MSK topic.

> **Note**
>
> If you're using IAM role-based authentication, your execution role will need the permissions listed in ??? in addition the ones listed below.

- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers
- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Self managed Apache Kafka execution role permissions

For self managed Apache Kafka, EventBridge requires the following permissions to manage resources that are related to your self managed Apache Kafka stream.

Required permissions

To create and store logs in a log group in Amazon CloudWatch Logs, your pipe must have the following permissions in its execution role:

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Optional permissions

Your pipe might also need permissions to:

- Describe your Secrets Manager secret.
- Access your AWS Key Management Service (AWS KMS) customer managed key.
- Access your Amazon VPC.

Secrets Manager and AWS KMS permissions

Depending on the type of access control that you're configuring for your Apache Kafka brokers, your pipe might need permission to access your Secrets Manager secret or to decrypt your AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC permissions

If only users within a VPC can access your self managed Apache Kafka cluster, your pipe must have permission to access your Amazon VPC resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your pipe's execution role must have the following permissions:

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

Amazon SQS execution role permissions

For Amazon SQS, EventBridge requires the following permissions to manage resources that are related to your Amazon SQS queue.

- sqs:ReceiveMessage
- sqs:DeleteMessage
- sqs:GetQueueAttributes

Enrichment and target permissions

To make API calls on the resources that you own, EventBridge Pipes needs appropriate permission. EventBridge Pipes uses the IAM role that you specify on the pipe for enrichment and target calls using the IAM principal `pipes.amazonaws.com`.

Creating an Amazon EventBridge pipe

EventBridge Pipes enables you to create point-to-point integrations between sources and targets, including advanced event transformations and enrichment.

To create an EventBridge pipe, you perform the following steps:

1. ???
2. ???
3. ???
4. ???
5. ???

To quickly set up a sample pipe, see the section called “Get started with pipes”. This topic uses CloudFormation to deploy a pipe and its associated resources, and walks you through an overview of a pipe's capabilities.

For information on how to create a pipe using the AWS CLI, see create-pipe in the *AWS CLI Command Reference*.

Specifying a source

To start, specify the source from which you want the pipe to receive events.

To specify a pipe source by using the console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. On the navigation pane, choose **Pipes**.
3. Choose **Create pipe**.
4. Enter a name for the pipe.

5. (Optional) Add a description for the pipe.
6. On the **Build pipe** tab, for **Source**, choose the type of source you want to specify for this pipe, and configure the source.

   Configuration properties differ based on the type of source you choose:

   Confluent

   To configure a Confluent Cloud stream as a source, by using the console

   1. For **Source**, choose **Confluent Cloud**.
   2. For **Bootstrap servers**, enter the `host:port` pair addresses of your brokers.
   3. For **Topic name**, enter the name of topic that the pipe will read from.
   4. (Optional) For **VPC**, choose the VPC that you want. Then, for **VPC subnets**, choose the desired subnets. For **VPC security groups**, choose the security groups.
   5. For **Authentication - optional**, turn on **Use Authentication** and do the following:
      a. For **Authentication method**, choose the authentication type.
      b. For **Secret key**, choose the secret key.

      For more information, see Authenticate to Confluent Cloud resources in the Confluent documentation.
   6. (Optional) For **Additional setting - optional**, do the following:
      a. For **Starting position**, choose one of the following:
         - **Latest** – Start reading the stream with the most recent record in the shard.
         - **Trim horizon** – Start reading the stream with the last untrimmed record in the shard. This is the oldest record in the shard.
      b. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 100.
      c. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.

   DynamoDB

   1. For **Source**, choose **DynamoDB**.
   2. For **DynamoDB stream**, choose the stream you want to use as a source.

3. For **Starting position**, choose one of the following:
   - **Latest** – Start reading the stream with the most recent record in the shard.
   - **Trim horizon** – Start reading the stream with the last untrimmed record in the shard. This is the oldest record in the shard.
4. (Optional) For **Additional setting - optional**, do the following:
   a. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 10.
   b. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.
   c. For **Concurrent batches per shard - optional**, enter the number of batches from the same shard that can be read at the same time.
   d. For **On partial batch item failure**, choose the following:
      - **AUTOMATIC_BISECT** – Halve each batch and retry each half until all the records are processed or there is one failed message remaining in the batch.

> **Note**
>
> If you don't choose **AUTOMATIC_BISECT**, you can return specific failed records and only those get retried.

Kinesis

To configure a Kinesis source by using the console

1. For **Source**, choose **Kinesis**.
2. For **Kinesis stream**, choose the stream that you want to use as a source.
3. For **Starting position**, choose one of the following:
   - **Latest** – Start reading the stream with the most recent record in the shard.
   - **Trim horizon** – Start reading the stream with the last untrimmed record in the shard. This is the oldest record in the shard.
   - **At timestamp** – Start reading the stream from a specified time. Under **Timestamp**, enter a data and time using YYYY/MM/DD and hh:mm:ss format.
4. (Optional) For **Additional setting - optional**, do the following:

a. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 10.

b. (Optional) For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.

c. For **Concurrent batches per shard - optional**, enter the number of batches from the same shard that can be read at the same time.

d. For **On partial batch item failure**, choose the following:

- **AUTOMATIC_BISECT** – Halve each batch and retry each half until all the records are processed or there is one failed message remaining in the batch.

> **Note**
>
> If you don't choose **AUTOMATIC_BISECT**, you can return specific failed records and only those get retried.

Amazon MQ

To configure an Amazon MQ source by using the console

1. For **Source**, choose **Amazon MQ**.
2. For **Amazon MQ broker**, choose the stream you want to use as a source.
3. For **Queue name**, enter the name of the queue that the pipe will read from.
4. For **Authentication Method**, choose **BASIC_AUTH**.
5. For **Secret key**, choose the secret key.
6. (Optional) For **Additional setting - optional**, do the following:
   a. For **Batch size - optional**, enter a maximum number of messages for each batch. The default value is 100.
   b. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.

Amazon MSK

To configure an Amazon MSK source by using the console

1. For **Source**, choose **Amazon MSK**.
2. For **Amazon MSK cluster**, choose the cluster that you want to use.
3. For **Topic name**, enter the name of topic that the pipe will read from.
4. (Optional) For **Consumer Group ID - optional**, enter the ID of the consumer group you want the pipe to join.
5. (Optional) For **Authentication - optional**, turn on **Use Authentication** and do the following:
   a. For **Authentication method**, choose the type you want.
   b. For **Secret key**, choose the secret key.
6. (Optional) For **Additional setting - optional**, do the following:
   a. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 100.
   b. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.
   c. For **Starting position**, choose one of the following:
      - **Latest** – Start reading the topic with the most recent record in the shard.
      - **Trim horizon** – Start reading the topic with the last untrimmed record in the shard. This is the oldest record in the shard.

> **Note**
>
> **Trim horizon** is the same as **Earliest** for Apache Kafka.

Self managed Apache Kafka

To configure a self managed Apache Kafka source by using the console

1. For **Source**, choose **Self-managed Apache Kafka**.
2. For **Bootstrap servers**, enter the `host:port` pair addresses of your brokers.

3. For **Topic name**, enter the name of topic that the pipe will read from.
4. (Optional) For **VPC**, choose the VPC that you want. Then, for **VPC subnets**, choose the desired subnets. For **VPC security groups**, choose the security groups.
5. (Optional) For **Authentication - optional**, turn on **Use Authentication** and do the following:
   a. For **Authentication method**, choose the authentication type.
   b. For **Secret key**, choose the secret key.
6. (Optional) For **Additional setting - optional**, do the following:
   a. For **Starting position**, choose one of the following:
      - **Latest** – Start reading the stream with the most recent record in the shard.
      - **Trim horizon** – Start reading the stream with the last untrimmed record in the shard. This is the oldest record in the shard.
   b. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 100.
   c. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.

Amazon SQS

To configure an Amazon SQS source by using the console

1. For **Source**, choose **SQS**.
2. For **SQS queue**, choose the queue you want to use.
3. (Optional) For **Additional setting - optional**, do the following:
   a. For **Batch size - optional**, enter a maximum number of records for each batch. The default value is 100.
   b. For **Batch window - optional**, enter a maximum number of seconds to gather records before proceeding.

Configuring event filtering (optional)

You can add filtering to your pipe so you're sending only a subset of events from your source to the target.

To configure filtering by using the console

1. Choose **Filtering**.
2. Under **Sample event - optional**, you'll see a sample event that you can use to build your event pattern, or you can enter your own event by choosing **Enter your own**.
3. Under **Event pattern**, enter the event pattern that you want to use to filter the events. For more information about constructing filters, see ???.

   The following is an example event pattern that only sends events with the value **Seattle** in the **City** field.

```
{
  "data": {
    "City": ["Seattle"]
  }
}
```

Now that events are being filtered, you can add optional enrichment and a target for the pipe.

Defining event enrichment (optional)

You can send the event data for enrichment to a Lambda function, AWS Step Functions state machine, Amazon API Gateway, or API destination.

To select enrichment

1. Choose **Enrichment**.
2. Under **Details**, for **Service**, select the service and related settings you want to use for enrichment.

You can also transform the data before sending it to be enhanced.

(Optional) To define the input transformer

1. Choose **Enrichment Input Transformer - optional**.
2. For **Sample events/Event Payload**, choose the sample event type.

3. For **Transformer**, enter the transformer syntax, such as `"Event happened at <$.detail.field>."` where `<$.detail.field>` is a reference to a field from the sample event. You can also double-click a field from the sample event to add it to the transformer.
4. For **Output**, verify that the output looks like you want it to.

Now that the data has been filtered and enhanced, you must define a target to send the event data to.

Configuring a target

To configure a target

1. Choose **Target**.
2. Under **Details**, for **Target service**, choose the target. The fields that display vary depending on the target that you choose. Enter information specific to this target type, as needed.

You can also transform the data before sending it to the target.

(Optional) To define the input transformer

1. Choose **Target Input Transformer - optional**.
2. For **Sample events/Event Payload**, choose the sample event type.
3. For **Transformer**, enter the transformer syntax, such as `"Event happened at <$.detail.field>."` where `<$.detail.field>` is a reference to a field from the sample event. You can also double-click a field from the sample event to add it to the transformer.
4. For **Output**, verify that the output looks like you want it to.

Now that the pipe is configured, make sure that its settings are configured correctly.

Configuring the pipe settings

A pipe is active by default, but you can deactivate it. You can also specify the permissions of the pipe, set up pipe logging, and add tags.

To configure the pipe settings

1. Choose the **Pipe settings** tab.

2. By default, newly created pipes are active as soon as they're created. If you want to create an inactive pipe, under **Activation**, for **Activate pipe**, turn off **Active**.
3. Under **Permissions**, for **Execution role**, do one of the following:

   a. To have EventBridge create a new execution role for this pipe, choose **Create a new role for this specific resource.** Under **Role name**, you can optionally edit the role name.

   b. To use an existing execution role, choose **Use existing role**. Under **Role name**, choose the role.

   > **Role manager enabled**
   >
   > If role manager is enabled in your account, EventBridge attaches the role for you. EventBridge replaces the **Create a new role for this specific resource** and **Use existing role** options with a **Customize** option. To use a different role, choose **Customize**. For more information, see IAM role creation in the *IAM User Guide*.

4. (Optional) If you have specified a Kinesis or DynamoDB stream as the pipe source, you can configure a retry policy and dead-letter queue (DLQ).

   For **Retry policy and Dead-letter queue - optional**, do the following:

   Under **Retry policy**, do the following:

   a. If you want to turn on retry policies, turn on **Retry**. By default, newly created pipes don't have a retry policy turned on.

   b. For **Maximum age of event**, enter a value between one minute (00:01) and 24 hours (24:00).

   c. For **Retry attempts**, enter a number between 0 and 185.

   d. If you want to use a dead-letter queue (DLQ), turn on **Dead-letter queue**, choose the method of your choice, and choose the queue or topic you'd like to use. By default, newly created pipes don't use a DLQ.

5. Choose the KMS key for EventBridge to use when encrypting pipe data.

   For more information on how EventBridge uses KMS keys, see Encryption at rest.

   - Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

This is the default.

- Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

  Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

  - Specify an existing customer managed key, or choose **Create a new KMS key**.

    EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

6. (Optional) Under **Logs - optional**, you can set up how EventBridge Pipes sends logging information to supported services, including how to configure those logs.

   For more information about logging pipe records, see ???.

   CloudWatch logs is selected as a log destination by default, as is the ERROR log level. So, by default, EventBridge Pipes creates a new CloudWatch log group to which it sends log records containing the ERROR level of detail.

   To have EventBridge Pipes send log records to any of the supported log destinations, do the following:

   a. Under **Logs - optional**, choose the destinations to which you want log records delivered.

   b. For **Log level**, choose the level of information for EventBridge to include in log records. The ERROR log level is selected by default.

      For more information, see ???.

   c. Select **Include execution data** if you want EventBridge to include event payload information and service request and response information in log records.

      For more information, see ???.

   d. Configure each log destination you selected:

For CloudWatch Logs logs, under **CloudWatch logs** do the following:

- For **CloudWatch log group**, choose whether to have EventBridge create a new log group, or you can select an existing log group or specifying the ARN of an existing log group.
- For new log groups, edit the log group name as desired.

CloudWatch logs is selected by default.

For Firehose stream logs, under **Firehose stream log**, select the Firehose stream.

For Amazon S3 logs, under **S3 logs** do the following:

- Enter the name of the bucket to use as the log destination.
- Enter the AWS account ID of the bucket owner.
- Enter any prefix text you want used when EventBridge creates S3 objects.

  For more information, see Organizing objects using prefixes in the *Amazon Simple Storage Service User Guide*.
- Choose how you want EventBridge to format S3 log records:
  - `json`: JSON
  - `plain`: Plain text
  - `w3c`: W3C extended logging file format

7. (Optional) Under **Tags - optional**, choose **Add new tag** and enter one or more tags for the rule. For more information, see ???.
8. Choose **Create pipe**.

Validating configuration parameters

After a pipe is created, EventBridge validates the following configuration parameters:

- **IAM role** – Because the source of a pipe can't be changed after the pipe is created, EventBridge verifies that the provided IAM role can access the source.

> **Note**
>
> EventBridge doesn't perform the same validation for enrichments or targets because they can be updated after the pipe is created.

- **Batching** – EventBridge validates that the batch size of the source doesn't exceed the maximum batch size of the target. If it does, EventBridge requires a lower batch size. Additionally, if a target doesn't support batching, you can't configure batching in EventBridge for the source.
- **Enrichments** – EventBridge validates that the batch size for API Gateway and API destination enrichments is 1 because only batch sizes of 1 are supported.

Starting or stopping an Amazon EventBridge pipe

By default, a pipe is `Running` and processes events when it's created.

If you create a pipe with Amazon SQS, Kinesis, or DynamoDB sources, pipe creation can typically take a minute or two.

If you create a pipe with Amazon MSK, self managed Apache Kafka, or Amazon MQ sources, pipes creation can take up to ten minutes.

To create a pipe without processing events using the console

- Turn off the **Activate pipe** setting.

To create a pipe without processing events programmatically

- In your API call, set the `DesiredState` to `Stopped`.

To start or stop an existing pipe using the console

- On the **Pipes settings** tab, under **Activation**, for **Activate pipe**, turn **Active** on or off.

To start or stop an existing pipe programmatically

- In your API call, set the `DesiredState` parameter to either `RUNNING` or `STOPPED`.

There can be a delay between when a pipe is STOPPED and when it no longer processes events:

- For Amazon SQS and stream sources, this delay is typically less than two minutes.
- For Amazon MQ and Apache Kafka sources, this delay may be up to fifteen minutes.

Amazon EventBridge Pipes sources

EventBridge Pipes receives event data from a variety of sources, applies optional filters and enrichments to that data, and sends it to a destination.

If a source enforces order to the events sent to EventBridge Pipes, that order is maintained throughout the entire process to the destination.

The following AWS services can be specified as sources for EventBridge Pipes:

- Amazon DynamoDB stream
- Amazon Kinesis stream
- Amazon MQ broker
- Amazon MSK stream
- Amazon SQS queue
- Apache Kafka stream

  When you specify an Apache Kafka stream as a pipe source, you can specify an Apache Kafka stream that you manage yourself, or one managed by a third-party provider such as:

  - Confluent Cloud
  - CloudKarafka
  - Redpanda

Amazon DynamoDB stream as a source for EventBridge Pipes

You can use EventBridge Pipes to receive records in a DynamoDB stream. You can then optionally filter or enhance these records before sending them to a target for processing. There are settings specific to Amazon DynamoDB Streams that you can choose when setting up the pipe. EventBridge Pipes maintains the order of records from the data stream when sending that data to the destination.

Important

Disabling a DynamoDB stream that is the source of a pipe results in that pipe becoming unusable, even if you then re-enable the stream. This happens because:

- You cannot stop, start, or update a pipe whose source is disabled.
- You cannot update a pipe with a new source after creation. When you re-enable a DynamoDB stream, that stream is assigned a new Amazon Resource Name (ARN), and is no longer associated with your pipe.

If you do re-enable the DynamoDB stream, you will then need to create a new pipe using the stream's new ARN.

Example event

The following sample event shows the information that's received by the pipe. You can use this event to create and filter your event patterns , or to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.

```
[
  {
    "eventID": "1",
    "eventVersion": "1.0",
    "dynamodb": {
      "Keys": {
        "Id": {
          "N": "101"
        }
      },
      "NewImage": {
        "Message": {
          "S": "New item!"
        },
        "Id": {
          "N": "101"
        }
      },
      "StreamViewType": "NEW_AND_OLD_IMAGES",
      "SequenceNumber": "111",
```

```
      "SizeBytes": 26
    },
    "awsRegion": "us-west-2",
    "eventName": "INSERT",
    "eventSourceARN": "arn:aws:dynamodb:us-east-1:111122223333:table/EventSourceTable",
    "eventSource": "aws:dynamodb"
  },
  {
    "eventID": "2",
    "eventVersion": "1.0",
    "dynamodb": {
      "OldImage": {
        "Message": {
          "S": "New item!"
        },
        "Id": {
          "N": "101"
        }
      },
      "SequenceNumber": "222",
      "Keys": {
        "Id": {
          "N": "101"
        }
      },
      "SizeBytes": 59,
      "NewImage": {
        "Message": {
          "S": "This item has changed"
        },
        "Id": {
          "N": "101"
        }
      },
      "StreamViewType": "NEW_AND_OLD_IMAGES"
    },
    "awsRegion": "us-west-2",
    "eventName": "MODIFY",
    "eventSourceARN": "arn:aws:dynamodb:us-east-1:111122223333:table/EventSourceTable",
    "eventSource": "aws:dynamodb"
  }
]
```

Polling and batching streams

EventBridge polls shards in your DynamoDB stream for records at a base rate of four times per second. When records are available, EventBridge processes the event and waits for the result. If processing succeeds, EventBridge resumes polling until it receives more records.

By default, EventBridge invokes your pipe as soon as records are available. If the batch that EventBridge reads from the source has only one record in it, only one event is processed. To avoid processing a small number of records, you can tell the pipe to buffer records for up to five minutes by configuring a batching window. Before processing the events, EventBridge continues to read records from the source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB.

> **Important**
>
> The pipe will deliver the stream records from DynamoDB to Amazon SQS at least once. To ensure that no records get dropped, we suggest setting a retry policy with a maximum age shorter than the retention period of the DynamoDB stream. Generally, DynamoDB streams retain events for 24 hours.

You can also increase concurrency by processing multiple batches from each shard in parallel. EventBridge can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, EventBridge still ensures in-order processing at the partition key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Pipe execution simultaneously. You can specify the number of concurrent batches that EventBridge polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent EventBridge Pipe executions at maximum to process 100 Kinesis data shards. This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high. Note that parallelization factor will not work if you are using Kinesis aggregation.

Polling and stream starting position

Be aware that stream source polling during pipe creation and updates is eventually consistent.

- During pipe creation, it may take several minutes to start polling events from the stream.

- During pipe updates to the source polling configuration, it may take several minutes to stop and restart polling events from the stream.

This means that if you specify `LATEST` as the starting position for the stream, the pipe could miss events sent during pipe creation or updates. To ensure no events are missed, specify the stream starting position as `TRIM_HORIZON`.

Reporting batch item failures

When EventBridge consumes and processes streaming data from an source, by default it checkpoints to the highest sequence number of a batch, but only when the batch is a complete success. To avoid reprocessing successfully processed messages in a failed batch, you can configure your enrichment or target to return an object indicating which messages succeeded and which failed. This is called a partial batch response.

For more information, see ???.

Success and failure conditions

If you return any of the following, EventBridge treats a batch as a complete success:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

If you return any of the following, EventBridge treats a batch as a complete failure:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

EventBridge retries failures based on your retry strategy.

Amazon Kinesis stream as a source for EventBridge Pipes

You can use EventBridge Pipes to receive records in a Kinesis data stream. You can then optionally filter or enhance these records before sending them to one of the available destinations for

processing. There are settings specific to Kinesis that you can choose when setting up the pipe. EventBridge Pipes maintains the order of records from the data stream when sending that data to the destination.

A Kinesis data stream is a set of shards. Each shard contains a sequence of data records. A **consumer** is an application that processes the data from a Kinesis data stream. You can map an EventBridge Pipe to a shared-throughput consumer (standard iterator), or to a dedicated-throughput consumer with enhanced fan-out.

For standard iterators, EventBridge uses the HTTP protocol to poll each shard in your Kinesis stream for records. The pipe shares the read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, you can create a data stream consumer with enhanced fan-out. Stream consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. The dedicated throughput can help if you have many applications reading the same data, or if you're reprocessing a stream with large records. Kinesis pushes records to EventBridge over HTTP/2. For information about Kinesis data streams, see Reading Data from Amazon Kinesis Data Streams.

Example event

The following sample event shows the information that is received by the pipe. You can use this event to create and filter your event patterns, or to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.


```
[
  {
    "kinesisSchemaVersion": "1.0",
    "partitionKey": "1",
    "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
    "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
    "approximateArrivalTimestamp": 1545084650.987,
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
  },
  {
```

```
    "kinesisSchemaVersion": "1.0",
    "partitionKey": "1",
    "sequenceNumber": "49590338271490256608559692540925702759324208523137515618",
    "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
    "approximateArrivalTimestamp": 1545084711.166,
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
  }
]
```

Polling and batching streams

EventBridge polls shards in your Kinesis stream for records at a base rate of once per second. When records are available, EventBridge processes the event and waits for the result. If processing succeeds, EventBridge resumes polling until it receives more records.

By default, EventBridge invokes your pipe as soon as records are available. If the batch that EventBridge reads from the source has only one record in it, only one event is processed. To avoid processing a small number of records, you can tell the pipe to buffer records for up to five minutes by configuring a batching window. Before processing the events, EventBridge continues to read records from the source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB.

You can also increase concurrency by processing multiple batches from each shard in parallel. EventBridge can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, EventBridge still ensures in-order processing at the partition key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Pipe execution simultaneously. You can specify the number of concurrent batches that EventBridge polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent EventBridge Pipe executions at maximum to process 100 Kinesis data shards. This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high. Note that parallelization factor will not work if you are using Kinesis aggregation.

Polling and stream starting position

Be aware that stream source polling during pipe creation and updates is eventually consistent.

- During pipe creation, it may take several minutes to start polling events from the stream.
- During pipe updates to the source polling configuration, it may take several minutes to stop and restart polling events from the stream.

This means that if you specify `LATEST` as the starting position for the stream, the pipe could miss events sent during pipe creation or updates. To ensure no events are missed, specify the stream starting position as `TRIM_HORIZON` or `AT_TIMESTAMP`.

Reporting batch item failures

When EventBridge consumes and processes streaming data from an source, by default it checkpoints to the highest sequence number of a batch, but only when the batch is a complete success. To avoid reprocessing successfully processed messages in a failed batch, you can configure your enrichment or target to return an object indicating which messages succeeded and which failed. This is called a partial batch response.

For more information, see ???.

Success and failure conditions

If you return any of the following, EventBridge treats a batch as a complete success:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

If you return any of the following, EventBridge treats a batch as a complete failure:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

EventBridge retries failures based on your retry strategy.

Amazon MQ message broker as a source in EventBridge Pipes

You can use EventBridge Pipes to receive records from an Amazon MQ message broker. You can then optionally filter or enhance these records before sending them to one of the available destinations for processing. There are settings specific to Amazon MQ that you can choose when setting up a pipe. EventBridge Pipes maintains the order of the records from the message broker when sending that data to the destination.

Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ. A message broker enables software applications and components to communicate using different programming languages, operating systems, and formal messaging protocols with either topics or queues as event destinations.

Amazon MQ can also manage Amazon Elastic Compute Cloud (Amazon EC2) instances on your behalf by installing ActiveMQ or RabbitMQ brokers. After a broker is installed, it provides different network topologies and other infrastructure needs to your instances.

The Amazon MQ source has the following configuration restrictions:

- **Cross account** – EventBridge doesn't support cross-account processing. You can't use EventBridge to process records from an Amazon MQ message broker that is in a different AWS account.
- **Authentication** – For ActiveMQ, only the ActiveMQ SimpleAuthenticationPlugin is supported. For RabbitMQ, only the PLAIN authentication mechanism is supported. To manage credentials, use AWS Secrets Manager. For more information about ActiveMQ authentication, see Integrating ActiveMQ brokers with LDAP in the Amazon MQ Developer Guide.
- **Connection quota** – Brokers have a maximum number of allowed connections for each wire-level protocol. This quota is based on the broker instance type. For more information, see the Brokers section of ***Quotas in Amazon MQ*** in the Amazon MQ Developer Guide.
- **Connectivity** – You can create brokers in a public or private virtual private cloud (VPC). For private VPCs, your pipe needs access to the VPC to receive messages.
- **Event destinations** – Only queue destinations are supported. However, you can use a virtual topic, which behaves as both a topic internally and as a queue externally when it interacts with your pipes. For more information, see Virtual Destinations on the Apache ActiveMQ website, and Virtual Hosts on the RabbitMQ website.
- **Network topology** – For ActiveMQ, only one single-instance or standby broker is supported for pipe. For RabbitMQ, only one single-instance broker or cluster deployment is supported for

each pipe. Single-instance brokers require a failover endpoint. For more information about these broker deployment modes, see Active MQ Broker Architecture and Rabbit MQ Broker Architecture in the Amazon MQ Developer Guide.

- **Protocols** – Supported protocols depend on the Amazon MQ integration that you use.
  - For ActiveMQ integrations, EventBridge uses the OpenWire/Java Message Service (JMS) protocol to consume messages. Message consumption isn't supported on any other protocol. EventBridge only supports the `TextMessage` and `BytesMessage` operations within the JMS protocol. For more information about the OpenWire protocol, see OpenWire on the Apache ActiveMQ website.
  - For RabbitMQ integrations, EventBridge uses the AMQP 0-9-1 protocol to consume messages. No other protocols are supported for consuming messages. For more information about RabbitMQ's implementation of the AMQP 0-9-1 protocol, see AMQP 0-9-1 Complete Reference Guide on the RabbitMQ website.

EventBridge automatically supports the latest versions of ActiveMQ and RabbitMQ that Amazon MQ supports. For the latest supported versions, see Amazon MQ release notes in the Amazon MQ Developer Guide.

> **Note**
>
> By default, Amazon MQ has a weekly maintenance window for brokers. During that window of time, brokers are unavailable. For brokers without standby, EventBridge won't process messages until the window ends.

Example events

The following sample event shows the information that is received by the pipe. You can use this event to create and filter your event patterns, or to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.

ActiveMQ

```
[
  {
    "eventSource": "aws:amq",
    "eventSourceArn": "arn:aws:mq:us-
west-2:112556298976:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
```

```
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "data": "QUJDOkFBQUE=",
    "connectionId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalname": "testQueue"
    },
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959
  },
  {
    "eventSource": "aws:amq",
    "eventSourceArn": "arn:aws:mq:us-
west-2:112556298976:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/bytes-message",
    "data": "3DTOOW7crj51prgVLQaGQ82S48k=",
    "connectionId": "myJMSCoID1",
    "persistent": false,
    "destination": {
      "physicalname": "testQueue"
    },
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959
  }
]
```

RabbitMQ

```
[
  {
    "eventSource": "aws:rmq",
    "eventSourceArn": "arn:aws:mq:us-
west-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
    "eventSourceKey": "pizzaQueue::/",
    "basicProperties": {
      "contentType": "text/plain",
      "contentEncoding": null,
```

```
      "headers": {
        "header1": {
          "bytes": [
            118,
            97,
            108,
            117,
            101,
            49
          ]
        },
        "header2": {
          "bytes": [
            118,
            97,
            108,
            117,
            101,
            50
          ]
        },
        "numberInHeader": 10
      },
      "deliveryMode": 1,
      "priority": 34,
      "correlationId": null,
      "replyTo": null,
      "expiration": "60000",
      "messageId": null,
      "timestamp": "Jan 1, 1970, 12:33:41 AM",
      "type": null,
      "userId": "AIDACKCEVSQ6C2EXAMPLE",
      "appId": null,
      "clusterId": null,
      "bodySize": 80
    },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
  }
]
```

Consumer group

To interact with Amazon MQ, EventBridge creates a consumer group that can read from your Amazon MQ brokers. The consumer group is created with the same ID as the pipe UUID.

For Amazon MQ sources, EventBridge batches records together and sends them to your function in a single payload. To control behavior, you can configure the batching window and batch size. EventBridge pulls messages until one of the following occurs:

- The processed records reach the payload size maximum of 6 MB.
- The batching window expires.
- The number of records reaches the full batch size.

EventBridge converts your batch into a single payload and then invokes your function. Messages aren't persisted or deserialized. Instead, the consumer group retrieves them as a BLOB of bytes. It then base64-encodes them into a JSON payload. If the pipe returns an error for any of the messages in a batch, EventBridge retries the entire batch of messages until processing succeeds or the messages expire.

Network configuration

By default, Amazon MQ brokers are created with the `PubliclyAccessible` flag set to false. It's only when `PubliclyAccessible` is set to true that the broker receives a public IP address. For full access with your pipe, your broker must either use a public endpoint or provide access to the VPC.

If your Amazon MQ broker isn't publicly accessible, EventBridge must have access to the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your broker.

- To access the VPC of your Amazon MQ brokers, EventBridge can use outbound internet access for the subnets of your source. For public subnets this must be a managed NAT gateway. For private subnets it can be a NAT gateway, or your own NAT. Ensure that the NAT has a public IP address and can connect to the internet.
- EventBridge Pipes also supports event delivery through AWS PrivateLink, allowing you to send events from an event source located in an Amazon Virtual Private Cloud (Amazon VPC) to a Pipes target without traversing the public internet. You can use Pipes to poll from Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, and Amazon MQ

sources residing in a private subnet without the need to deploy an internet gateway, configure firewall rules, or set up proxy servers.

To set up a VPC endpoint, see Create a VPC endpoint in the *AWS PrivateLink User Guide*. For service name, select `com.amazonaws.region.pipes-data`.

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Amazon MQ broker port for the security groups specified for your source.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Amazon MQ broker port for the security groups specified for your source.

  Broker ports include:

  - 9092 for plaintext
  - 9094 for TLS
  - 9096 for SASL
  - 9098 for IAM

> **Note**
>
> Your Amazon VPC configuration is discoverable through the Amazon MQ API. You don't need to configure it during setup.

Amazon Managed Streaming for Apache Kafka topic as a source in EventBridge Pipes

You can use EventBridge Pipes to receive records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) topic, including from the Amazon MSK Serverless cluster type. You can optionally filter or enhance these records before sending them to one of the available destinations for processing. There are settings specific to Amazon MSK that you can choose when setting up a pipe. EventBridge Pipes maintains the order of the records from the message broker when sending that data to the destination.

Amazon MSK is a fully managed service that you can use to build and run applications that use Apache Kafka to process streaming data. Amazon MSK simplifies the setup, scaling, and management of clusters running Apache Kafka. With Amazon MSK, you can configure your application for multiple Availability Zones and for security with AWS Identity and Access Management (IAM). Amazon MSK supports multiple open-source versions of Kafka.

Amazon MSK as an source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. EventBridge internally polls for new messages from the source and then synchronously invokes the target. EventBridge reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable. (The default is 100 messages.)

For Apache Kafka-based sources, EventBridge supports processing control parameters, such as batching windows and batch size.

EventBridge reads the messages sequentially for each partition. After EventBridge processes each batch, it commits the offsets of the messages in that batch. If the pipe's target returns an error for any of the messages in a batch, EventBridge retries the entire batch of messages until processing succeeds or the messages expire.

EventBridge sends the batch of messages in the event when it invokes the target. The event payload contains an array of messages. Each array item contains details of the Amazon MSK topic and partition identifier, together with a timestamp and a base64-encoded message.

Example events

The following sample event shows the information that is received by the pipe. You can use this event to create and filter your event patterns, or for to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.

```
[
  {
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:sa-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
    "eventSourceKey": "mytopic-0",
    "topic": "mytopic",
    "partition": "0",
    "offset": 15,
    "timestamp": 1545084650987,
    "timestampType": "CREATE_TIME",
```

```
    "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
    "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
    "headers": [
      {
        "headerKey": [
          104,
          101,
          97,
          100,
          101,
          114,
          86,
          97,
          108,
          117,
          101
        ]
      }
    ]
  }
]
```

Polling and stream starting position

Be aware that stream source polling during pipe creation and updates is eventually consistent.

- During pipe creation, it may take several minutes to start polling events from the stream.
- During pipe updates to the source polling configuration, it may take several minutes to stop and restart polling events from the stream.

This means that if you specify `LATEST` as the starting position for the stream, the pipe could miss events sent during pipe creation or updates. To ensure no events are missed, specify the stream starting position as `TRIM_HORIZON`.

MSK cluster authentication

EventBridge needs permission to access the Amazon MSK cluster, retrieve records, and perform other tasks. Amazon MSK supports several options for controlling client access to the MSK cluster. For more information about which authentication method is used when, see ???.

Cluster access options

- Unauthenticated access
- SASL/SCRAM authentication
- IAM role-based authentication
- Mutual TLS authentication
- Configuring the mTLS secret
- How EventBridge chooses a bootstrap broker

Unauthenticated access

We recommend only using unauthenticated access for development. Unauthenticated access will only work if IAM role-based authentication is disabled for the cluster.

SASL/SCRAM authentication

Amazon MSK supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption. For EventBridge to connect to the cluster, you store the authentication credentials (sign-in credentials) in an AWS Secrets Manager secret.

For more information about using Secrets Manager, see User name and password authentication with AWS Secrets Manager in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Amazon MSK doesn't support SASL/PLAIN authentication.

IAM role-based authentication

You can use IAM to authenticate the identity of clients that connect to the MSK cluster. If IAM authentication is active on your MSK cluster, and you don't provide a secret for authentication, EventBridge automatically defaults to using IAM authentication. To create and deploy IAM user or role-based policies, use the IAM console or API. For more information, see IAM access control in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

To allow EventBridge to connect to the MSK cluster, read records, and perform other required actions, add the following permissions to your pipes's execution role.

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
            "arn:aws-eusc:kafka:eusc-de-east-1:123456789012:cluster/cluster-
name/cluster-uuid",
    "arn:aws-eusc:kafka:eusc-de-east-1:123456789012:topic/cluster-name/cluster-
uuid/topic-name",
    "arn:aws-eusc:kafka:eusc-de-east-1:123456789012:group/cluster-name/cluster-
uuid/consumer-group-id"
            ]
        }
    ]
}
```

You can scope these permissions to a specific cluster, topic, and group. For more information, see the Amazon MSK Kafka actions in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

For Amazon MSK, EventBridge acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate EventBridge with the brokers in your MSK cluster. The client certificate must be signed by a certificate authority (CA) in the server's trust store. The MSK cluster sends a server certificate to EventBridge to authenticate the brokers with EventBridge. The server certificate must be signed by a CA that's in the AWS trust store.

Amazon MSK doesn't support self-signed server certificates, because all brokers in Amazon MSK use public certificates signed by Amazon Trust Services CAs, which EventBridge trusts by default.

For more information about mTLS for Amazon MSK, see Mutual TLS Authentication in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Configuring the mTLS secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

> **Note**
>
> EventBridge supports the PBES1 (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in PKCS #8 format, with the following structure:

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, you include the private key password in the secret.

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

How EventBridge chooses a bootstrap broker

EventBridge chooses a bootstrap broker based on the authentication methods available on your cluster, and whether you provide a secret for authentication. If you provide a secret for mTLS or SASL/SCRAM, EventBridge automatically chooses that authentication method. If you don't provide a secret, EventBridge chooses the strongest authentication method that's active on your cluster. The following is the order of priority in which EventBridge selects a broker, from strongest to weakest authentication:

- mTLS (secret provided for mTLS)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (no secret provided, and IAM authentication is active)
- Unauthenticated TLS (no secret provided, and IAM authentication is not active)
- Plaintext (no secret provided, and both IAM authentication and unauthenticated TLS are not active)

> **Note**
>
> If EventBridge can't connect to the most secure broker type, it doesn't attempt to connect to a different (weaker) broker type. If you want EventBridge to choose a weaker broker type, deactivate all stronger authentication methods on your cluster.

Network configuration

EventBridge must have access to the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your Amazon MSK cluster.

- To access the VPC of your Amazon MSK cluster, EventBridge can use outbound internet access for the subnets of your source. For private subnets it can be a NAT gateway, or your own NAT. Ensure that the NAT has a public IP address and can connect to the internet. For public subnets you must use VPC Endpoints (explained below).
- EventBridge Pipes also supports event delivery through AWS PrivateLink, allowing you to send events from an event source located in an Amazon Virtual Private Cloud (Amazon VPC) to a Pipes target without traversing the public internet. You can use Pipes to poll from Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, and Amazon MQ sources residing in a private subnet without the need to deploy an internet gateway, configure firewall rules, or set up proxy servers. You can also use VPC Endpoints to support delivery from Kafka clusters in public subnets.

  To set up a VPC endpoint, see Create a VPC endpoint in the *AWS PrivateLink User Guide*. For service name, select `com.amazonaws.region.pipes-data`.

  If your Amazon MSK cluster runs in a fully isolated private subnet, EventBridge reaches the cluster and the services it depends on only through interface VPC endpoints. This applies when the subnet has no internet gateway and no NAT gateway. In this case, create interface VPC endpoints for the following services in addition to `pipes-data`:

  - `com.amazonaws.region.lambda` – always required.
  - `com.amazonaws.region.sts` – always required.
  - `com.amazonaws.region.secretsmanager` – required when your cluster uses secret-based authentication, such as SASL/SCRAM or mTLS, so that EventBridge can retrieve the secret from Secrets Manager.

- `com.amazonaws.region.kms` – required when the pipe uses a customer managed key, so that EventBridge can decrypt that key.

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Amazon MSK broker port for the security groups specified for your source.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Amazon MSK broker port for the security groups specified for your source.

  Broker ports include:
  - 9092 for plaintext
  - 9094 for TLS
  - 9096 for SASL
  - 9098 for IAM

> **Note**
>
> Your Amazon VPC configuration is discoverable through the Amazon MSK API. You don't need to configure it during setup.

Customizable consumer group ID

When setting up Apache Kafka as an source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Apache Kafka consumer group that you want your pipe to join. You can use this feature to migrate any ongoing Apache Kafka record processing setups from other consumers to EventBridge.

If you specify a consumer group ID and there are other active pollers within that consumer group, Apache Kafka distributes messages across all consumers. In other words, EventBridge doesn't receive all messages for the Apache Kafka topic. If you want EventBridge to handle all messages in the topic, turn off any other pollers in that consumer group.

Additionally, if you specify a consumer group ID, and Apache Kafka finds a valid existing consumer group with the same ID, EventBridge ignores the `StartingPosition` parameter for your pipe. Instead, EventBridge begins processing records according to the committed offset of the consumer

group. If you specify a consumer group ID, and Apache Kafka can't find an existing consumer group, then EventBridge configures your source with the specified `StartingPosition`.

The consumer group ID that you specify must be unique among all your Apache Kafka event sources. After creating a pipe with the consumer group ID specified, you can't update this value.

Auto scaling of the Amazon MSK source

When you initially create an Amazon MSK source, EventBridge allocates one consumer to process all partitions in the Apache Kafka topic. Each consumer has multiple processors running in parallel to handle increased workloads. Additionally, EventBridge automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.

In one-minute intervals, EventBridge evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than EventBridge can process them. If necessary, EventBridge adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target is overloaded, EventBridge reduces the number of consumers. This action reduces the workload on the pipe by reducing the number of messages that consumers can retrieve and send to the pipe.

Apache Kafka streams as a source in EventBridge Pipes

Apache Kafka is an open-source event streaming platform that supports workloads such as data pipelines and streaming analytics. You can use Amazon Managed Streaming for Apache Kafka (Amazon MSK), or a self managed Apache Kafka cluster. In AWS terminology, a *self managed* cluster refers to any Apache Kafka cluster not hosted by AWS. This includes both clusters you manage yourself, as well as those hosted by a third-party provider, such as Confluent Cloud, CloudKarafka, or Redpanda.

For more information on other AWS hosting options for your cluster, see Best Practices for Running Apache Kafka on AWS on the *AWS Big Data Blog*.

Apache Kafka as a source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. EventBridge internally polls for new messages from the source and then synchronously invokes the target. EventBridge reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable. (The default is 100 messages.)

For Apache Kafka-based sources, EventBridge supports processing control parameters, such as batching windows and batch size.

EventBridge sends the batch of messages in the event parameter when it invokes your pipe. The event payload contains an array of messages. Each array item contains details of the Apache Kafka topic and Apache Kafka partition identifier, together with a timestamp and a base64-encoded message.

Example events

The following sample event shows the information that is received by the pipe. You can use this event to create and filter your event patterns, or to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.


```
[
  {
    "eventSource": "SelfManagedKafka",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
    "eventSourceKey": "mytopic-0",
    "topic": "mytopic",
    "partition": 0,
    "offset": 15,
    "timestamp": 1545084650987,
    "timestampType": "CREATE_TIME",
    "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
    "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
    "headers": [
      {
        "headerKey": [
          104,
          101,
          97,
          100,
          101,
          114,
          86,
          97,
          108,
          117,
          101
        ]
```

```
      }
    ]
  }
]
```

Apache Kafka cluster authentication

EventBridge Pipes supports several methods to authenticate with your self managed Apache Kafka cluster. Make sure that you configure the Apache Kafka cluster to use one of these supported authentication methods. For more information about Apache Kafka security, see the Security section of the Apache Kafka documentation.

VPC access

If you are using a self managed Apache Kafka environment where only Apache Kafka users within your VPC have access to your Apache Kafka brokers, you must configure the Amazon Virtual Private Cloud (Amazon VPC) in the Apache Kafka source.

SASL/SCRAM authentication

EventBridge Pipes supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption. EventBridge Pipes sends the encrypted credentials to authenticate with the cluster. For more information about SASL/SCRAM authentication, see RFC 5802.

EventBridge Pipes supports SASL/PLAIN authentication with TLS encryption. With SASL/PLAIN authentication, EventBridge Pipes sends credentials as clear text (unencrypted) to the server.

For SASL authentication, you store the sign-in credentials as a secret in AWS Secrets Manager.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

In self managed Apache Kafka, EventBridge Pipes acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate EventBridge Pipes with your Apache Kafka brokers. The client certificate must be signed by a certificate authority (CA) in the server's trust store.

The Apache Kafka cluster sends a server certificate to EventBridge Pipes to authenticate the Apache Kafka brokers with EventBridge Pipes. The server certificate can be a public CA certificate or a private CA/self-signed certificate. The public CA certificate must be signed by a CA that's in the EventBridge Pipes trust store. For a private CA/self-signed certificate, you configure the server root CA certificate (as a secret in Secrets Manager). EventBridge Pipes uses the root certificate to verify the Apache Kafka brokers.

For more information about mTLS, see Introducing mutual TLS authentication for Amazon MSK as an source.

Configuring the client certificate secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

> **Note**
>
> EventBridge Pipes supports the PBES1 (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in PKCS #8 format, with the following structure:

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, include the private key password in the secret.

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Configuring the server root CA certificate secret

You create this secret if your Apache Kafka brokers use TLS encryption with certificates signed by a private CA. You can use TLS encryption for VPC, SASL/SCRAM, SASL/PLAIN, or mTLS authentication.

The server root CA certificate secret requires a field that contains the Apache Kafka broker's root CA certificate in PEM format. The following example shows the structure of the secret.

```
{
     "certificate": "-----BEGIN CERTIFICATE-----
  MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
  EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
```

```
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
```

Network configuration

If you are using a self managed Apache Kafka environment that uses private VPC connectivity, EventBridge must have access to the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your Apache Kafka brokers.

- To access the VPC of your Apache Kafka cluster, EventBridge can use outbound internet access for the subnets of your source. For private subnets it can be a NAT gateway, or your own NAT. Ensure that the NAT has a public IP address and can connect to the internet. For public subnets you must use VPC Endpoints (explained below).
- EventBridge Pipes also supports event delivery through AWS PrivateLink, allowing you to send events from an event source located in an Amazon Virtual Private Cloud (Amazon VPC) to a Pipes target without traversing the public internet. You can use Pipes to poll from Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, and Amazon MQ sources residing in a private subnet without the need to deploy an internet gateway, configure firewall rules, or set up proxy servers. You can also use VPC Endpoints to support delivery from Kafka clusters in public subnets.

  To set up a VPC endpoint, see Create a VPC endpoint in the *AWS PrivateLink User Guide*. For service name, select `com.amazonaws.region.pipes-data`.

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Apache Kafka broker port for the security groups specified for your source.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Apache Kafka broker port for the security groups specified for your source.

  Broker ports include:

  - 9092 for plaintext
  - 9094 for TLS
  - 9096 for SASL
  - 9098 for IAM

Consumer auto scaling with Apache Kafka sources

When you initially create an Apache Kafka source, EventBridge allocates one consumer to process all partitions in the Kafka topic. Each consumer has multiple processors running in parallel to handle increased workloads. Additionally, EventBridge automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.

In one-minute intervals, EventBridge evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than EventBridge can process them. If necessary, EventBridge adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target is overloaded, EventBridge reduces the number of consumers. This action reduces the workload on the function by reducing the number of messages that consumers can retrieve and send to the function.

Amazon Simple Queue Service as a source in EventBridge Pipes

You can use EventBridge Pipes to receive records from an Amazon SQS queue. You can then optionally filter or enhance these records before sending them to an available destination for processing.

You can use a pipe to process messages in an Amazon Simple Queue Service (Amazon SQS) queue. EventBridge Pipes support standard queues and first-in, first-out (FIFO) queues. With Amazon SQS, you can offload tasks from one component of your application by sending them to a queue and processing them asynchronously.

EventBridge polls the queue and invokes your pipe synchronously with an event that contains queue messages. EventBridge reads messages in batches and invokes your pipe once for each batch. When your pipe successfully processes a batch, EventBridge deletes its messages from the queue.

By default, EventBridge polls up to 10 messages in your queue simultaneously and sends that batch to your pipe. To avoid invoking the pipe with a small number of records, you can tell the event source to buffer records for up to five minutes by configuring a batch window. Before invoking the pipe, EventBridge continues to poll messages from the Amazon SQS standard queue until one of these things occurs:

- The batch window expires.

- The invocation payload size quota is reached.
- The configured maximum batch size is reached.

> **Note**
>
> If you're using a batch window and your Amazon SQS queue contains low traffic, EventBridge might wait for up to 20 seconds before invoking your pipe. This is true even if you set a batch window for fewer than 20 seconds. For FIFO queues, records contain additional attributes that are related to deduplication and sequencing.

When EventBridge reads a batch, the messages stay in the queue but are hidden for the length of the queue's visibility timeout. If your pipe successfully processes the batch, EventBridge deletes the messages from the queue. By default, if your pipe encounters an error while processing a batch, all messages in that batch become visible in the queue again. For this reason, your pipe code must be able to process the same message multiple times without unintended side effects. You can modify this reprocessing behavior by including batch item failures in your pipe response. The following example shows an event for a batch of two messages.

Example events

The following sample event shows the information that is received by the pipe. You can use this event to create and filter your event patterns, or to define input transformation. Not all of the fields can be filtered. For more information about which fields you can filter, see ???.

Standard queue

```
[
  {
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "Test message.",
    "attributes": {
      "ApproximateReceiveCount": "1",
      "SentTimestamp": "1545082649183",
      "SenderId": "AIDAIENQZJOLO23YVJ4VO",
      "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
```

```
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
  },
  {
    "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
    "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
    "body": "Test message.",
    "attributes": {
      "ApproximateReceiveCount": "1",
      "SentTimestamp": "1545082650636",
      "SenderId": "AIDAIENQZJOLO23YVJ4VO",
      "ApproximateFirstReceiveTimestamp": "1545082650649"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
  }
]
```

FIFO queue

```
[
  {
    "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
    "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
    "body": "Test message.",
    "attributes": {
      "ApproximateReceiveCount": "1",
      "SentTimestamp": "1573251510774",
      "SequenceNumber": "18849496460467696128",
      "MessageGroupId": "1",
      "SenderId": "AIDAIO23YVJENQZJOL4VO",
      "MessageDeduplicationId": "1",
      "ApproximateFirstReceiveTimestamp": "1573251510774"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
```

```
    "awsRegion": "us-east-2"
  }
]
```

Scaling and processing

For standard queues, EventBridge uses long polling to poll a queue until it becomes active. When messages are available, EventBridge reads up to five batches and sends them to your pipe. If messages are still available, EventBridge increases the number of processes that are reading batches by up to 300 more instances per minute. The maximum number of batches that a pipe can process simultaneously is 1,000.

For FIFO queues, EventBridge sends messages to your pipe in the order that it receives them. When you send a message to a FIFO queue, you specify a message group ID. Amazon SQS facilitates delivering messages in the same group to EventBridge, in order. EventBridge sorts the received messages into groups and sends only one batch at a time for a group. If your pipe returns an error, the pipe attempts all retries on the affected messages before EventBridge receives additional messages from the same group.

Configuring a queue to use with EventBridge Pipes

Create an Amazon SQS queue to serve as an source for your pipe. Then configure the queue to allow time for your pipe to process each batch of events—and for EventBridge to retry in response to throttling errors as it scales up.

To allow your pipe time to process each batch of records, set the source queue's visibility timeout to at least six times the combined runtime of the pipe enrichment and target components. The extra time allows for EventBridge to retry if your pipe is throttled while processing a previous batch.

If your pipe fails to process a message multiple times, Amazon SQS can send it to a dead-letter queue. When your pipe returns an error, EventBridge keeps it in the queue. After the visibility timeout occurs, EventBridge receives the message again. To send messages to a second queue after a number of receives, configure a dead-letter queue on your source queue.

> **Note**
>
> Make sure that you configure the dead-letter queue on the source queue, not on the pipe. The dead-letter queue that you configure on a pipe is used for the pipe's asynchronous invocation queue, not for source queues.

If your pipe returns an error, or can't be invoked because it's at maximum concurrency, processing might succeed with additional attempts. To give messages more chances to be processed before sending them to the dead-letter queue, set the `maxReceiveCount` on the source queue's redrive policy to at least **5**.

Reporting batch item failures

When EventBridge consumes and processes streaming data from an source, by default it checkpoints to the highest sequence number of a batch, but only when the batch is a complete success. To avoid reprocessing successfully processed messages in a failed batch, you can configure your enrichment or target to return an object indicating which messages succeeded and which failed. This is called a partial batch response.

For more information, see ???.

Success and failure conditions

If you return any of the following, EventBridge treats a batch as a complete success:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

If you return any of the following, EventBridge treats a batch as a complete failure:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

EventBridge retries failures based on your retry strategy.

Event filtering in Amazon EventBridge Pipes

With EventBridge Pipes, you can filter a given source's events and process only a subset of them. This filtering works in the same way as filtering on an EventBridge event bus or Lambda event source mapping, by using event patterns. For more information about event patterns, see ???.

A filter criteria `FilterCriteria` object is a structure that consists of a list of filters (`Filters`). Each filter is a structure that defines a filtering pattern (`Pattern`). A `Pattern` is a string representation of a JSON filter rule. A `FilterCriteria` object looks like the following example:

```
{
  "Filters": [
    {"Pattern": "{ \"Metadata1\": [ pattern1 ], \"data\": { \"Data1\": [ pattern2 ] }}"
    }
  ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON:

```
{
  "Metadata1": [ pattern1 ],
  "data": {"Data1": [ pattern2 ]}
}
```

Amazon Kinesis, Amazon MQ, Amazon MSK, and self managed Apache Kafka apply Base64 encoding to the payload, but not the metadata fields. For example, suppose your Kinesis stream contains an event like this:

```
{
  "kinesisSchemaVersion": "1.0",
  "partitionKey": "1",
  "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
  "data": {"City": "Seattle",
    "State": "WA",
    "Temperature": "46",
    "Month": "December"
  },
  "approximateArrivalTimestamp": 1545084650.987
}
```

When the event flows through your pipe, it'll look like the following with the `data` field base64-encoded:

```
{
  "kinesisSchemaVersion": "1.0",
  "partitionKey": "1",
  "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
  "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
  "approximateArrivalTimestamp": 1545084650.987,
  "eventSource": "aws:kinesis",
  "eventVersion": "1.0",
  "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
  "eventName": "aws:kinesis:record",
  "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
  "awsRegion": "us-east-2",
  "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

When you're creating event filters, EventBridge Pipes can access event content. This content is either JSON-escaped, such as the Amazon SQS `body` field, or base64-encoded, such as the Kinesis `data` field. If your data is valid JSON, your input templates or JSON paths for target parameters can reference the content directly, as EventBridge Pipes will automatically decode it. For example, if a Kinesis event source is valid JSON, you can reference a variable using `<$.data.someKey>`.

Continuing our example, to filter on the non-encoded `partitionKey` metadata outside the `data` object and the base64 encoded `City` property inside the `data` object, you would use the following filter:

```
{
  "partitionKey": [ "1" ],
  "data": {
    "City": [ "Seattle" ]
  }
}
```

When creating event patterns, you can filter based on the fields sent by the source API, and not on fields added by the polling operation. The following fields can't be used in event patterns:

- `awsRegion`
- `eventSource`

- `eventSourceARN`
- `eventVersion`
- `eventID`
- `eventName`
- `invokeIdentityArn`
- `eventSourceKey`

The following sections explain filtering behavior for each supported event source type.

Filtering Amazon SQS messages

If an Amazon SQS message doesn't satisfy your filter criteria, EventBridge automatically removes the message from the queue. You don't have to delete these messages manually in Amazon SQS. Connecting multiple pipes to one SQS queue is unlikely to be a useful setup - the pipes would be competing for messages that'll be dropped if unmatched.

An Amazon SQS message body can contain any string, not just JSON. EventBridge Pipes expects your `FilterCriteria` format to match the format of the incoming messages, either valid JSON or a plain string. If there is a mismatch, EventBridge Pipes drops the message. If your `FilterCriteria` don't include body, meaning you filter only by metadata, EventBridge Pipes skips this check. The following table summarizes the evaluation:

Filter pattern format	Incoming format	Result
Plain string	Plain string	EventBridge filters based on your filter criteria.
Plain string	Valid JSON	EventBridge drops the message.
Valid JSON	Plain string	EventBridge drops the message.
Valid JSON	Valid JSON	EventBridge filters based on your filter criteria.

Filter pattern format	Incoming format	Result
No filter pattern for body	Plain string	EventBridge filters based on your filter criteria.
No filter pattern for body	Valid JSON	EventBridge filters based on your filter criteria.

Filtering Kinesis and DynamoDB messages

After your filter criteria processes a Kinesis or DynamoDB record, the streams iterator advances past this record. If the record doesn't satisfy your filter criteria, you don't have to delete the record manually from your event source. After the retention period, Kinesis and DynamoDB automatically delete these old records. If you want records to be deleted sooner, see Changing the Data Retention Period.

To properly filter events from stream event sources, both the data field and your filter criteria for the data field must be in valid JSON format. (For Kinesis, the data field is `data`. For DynamoDB, the data field is `dynamodb`.) If either field isn't in a valid JSON format, EventBridge drops the message or throws an exception. The following table summarizes the specific behavior:

Filter pattern format	Incoming format	Result
Valid JSON	Valid JSON	EventBridge filters based on your filter criteria.
Valid JSON	Non-JSON	EventBridge drops the message.
No filter pattern for `data` (Kinesis) or `dynamodb` (DynamoDB)	Valid JSON	EventBridge filters based on your filter criteria.
No filter pattern for `data` (Kinesis) or `dynamodb` (DynamoDB)	Non-JSON	EventBridge filters based on your filter criteria.

Filter pattern format	Incoming format	Result
Non-JSON	Any	EventBridge throws an exception at the time of Pipe creation or update. The filter pattern must be valid JSON format.

Filtering Amazon Managed Streaming for Apache Kafka, self managed Apache Kafka, and Amazon MQ messages

> **Note**
>
> After you attach filter criteria to a pipe with an Apache Kafka or Amazon MQ event source, it can take up to 15 minutes to apply your filtering rules to events.

For Amazon MQ sources, the message field is `data`. For Apache Kafka sources (Amazon MSK and self managed Apache Kafka), there are two message fields: `key` and `value`.

EventBridge drops messages that don't match all fields included in the filter. For Apache Kafka, EventBridge commits offsets for matched and unmatched messages after successfully invoking the target. For Amazon MQ, EventBridge acknowledges matched messages after successfully invoking the function and acknowledges unmatched messages when filtering them.

Apache Kafka and Amazon MQ messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because EventBridge decodes Apache Kafka and Amazon MQ byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the `FilterCriteria` format, EventBridge processes metadata filters only. The following table summarizes the specific behavior:

Filter pattern format	Incoming format	Result
Plain string	Plain string	EventBridge filters based on your filter criteria.

Filter pattern format	Incoming format	Result
Plain string	Valid JSON	EventBridge filters on metadata only, ignoring the `data` field (Amazon MQ) or `key` and `value` fields (Apache Kafka)
Valid JSON	Plain string	EventBridge filters on metadata only, ignoring the `data` field (Amazon MQ) or `key` and `value` fields (Apache Kafka)
Valid JSON	Valid JSON	EventBridge filters based on your filter criteria.
No filter pattern for `data` (Amazon MQ) or `key` and `value` (Apache Kafka)	Plain string	EventBridge filters on metadata only, ignoring the `data` field (Amazon MQ) or `key` and `value` fields (Apache Kafka)
No filter pattern for `data` (Amazon MQ) or `key` and `value` (Apache Kafka)	Valid JSON	EventBridge filters on metadata only, ignoring the `data` field (Amazon MQ) or `key` and `value` fields (Apache Kafka)
Any	Non-UTF encoded string	EventBridge filters on metadata only, ignoring the `data` field (Amazon MQ) or `key` and `value` fields (Apache Kafka)

Differences between Lambda ESM and EventBridge Pipes

When filtering events, Lambda ESM and EventBridge Pipes operate generally the same way. The main difference is that `eventSourceKey` field isn't present in ESM payloads.

Using comparison operators in pipe filters

Comparison operators enable you to construct event patterns that match against field values in events.

For a complete list of the comparison operators supported for use in pipe filters, see Comparison operators.

Event enrichment in Amazon EventBridge Pipes

With the enrichment step of EventBridge Pipes, you can enhance the data from the source before sending it to the target. For example, you might receive *Ticket created* events that don't include the full ticket data. Using enrichment, you can have a Lambda function call the `get-ticket` API for the full ticket details. Pipes can then send that information to a target.

You can configure the following enrichments when setting up a pipe in EventBridge:

- API destination
- Amazon API Gateway
- Lambda function
- Step Functions state machine

> **Note**
>
> EventBridge Pipes only supports Express workflows as enrichments.

EventBridge invokes enrichments synchronously because it must wait for a response from the enrichment before invoking the target.

Enrichment responses are limited to a maximum size of 6MB.

You can also transform the data you receive from the source before sending it for enhancement. For more information, see ???.

Filtering events using enrichment

EventBridge Pipes passes the enrichment responses directly to the configured target. This includes array responses for targets that support batches. For more information about batch behavior, see ???. You can also use your enrichment as a filter and pass fewer events than were received from the source. If you don't want to invoke the target, return an empty response, such as `""`, `{}`, or `[]`.

> **Note**
>
> If you want to invoke the target with an empty payload, return an array with empty JSON `[{}]`.

Invoking enrichments

EventBridge invokes enrichments synchronously (invocation type set to `REQUEST_RESPONSE`) because it must wait for a response from the enrichment before invoking the target.

> **Note**
>
> For Step Functions state machines, EventBridge only supports Express workflows as enrichments, because they can be invoked synchronously.

Amazon EventBridge Pipes targets

You can send data in your pipe to a specific target. You can configure the following targets when setting up a pipe in EventBridge:

- API destination
- API Gateway
- Batch job queue
- CloudWatch log group
- ECS task
- Event bus in the same account and Region
- Firehose delivery stream
- Kinesis stream

- Lambda function (SYNC or ASYNC)
- Redshift cluster data API queries
- SageMaker AI Pipeline
- Amazon SNS topic (SNS FIFO topics not supported)
- Amazon SQS queue
- Step Functions state machine
  - Express workflows (SYNC or ASYNC)
  - Standard workflows (ASYNC)
- Timestream for LiveAnalytics table

Target parameters

Some target services don't send the event payload to the target, instead, they treat the event as a trigger for invoking a specific API. EventBridge uses the `PipeTargetParameters` to specify what information gets sent to that API. These include the following:

- API destinations (The data sent to an API destination must match the structure of the API. You must use the `InputTemplate` object to make sure the data is structured correctly. If you want to include the original event payload, reference it in the `InputTemplate`.)
- API Gateway (The data sent to API Gateway must match the structure of the API. You must use the `InputTemplate` object to make sure the data is structured correctly. If you want to include the original event payload, reference it in the `InputTemplate`.)
- `PipeTargetRedshiftDataParameters` (Amazon Redshift Data API clusters)
- `PipeTargetSageMakerPipelineParameters` (Amazon SageMaker Runtime Model Building Pipelines)
- `PipeTargetBatchJobParameters` (AWS Batch)

> **Note**
>
> EventBridge does not support all JSON Path syntax and evaluate it at runtime. Supported syntax includes:
>
> - dot notation (for example,`$.detail`)
> - dashes

- underscores
- alphanumeric characters
- array indices
- wildcards (*)
- forward slashes

Dynamic path parameters

EventBridge Pipes target parameters support optional dynamic JSON path syntax. You can use this syntax to specify JSON paths instead of static values (for example `$.detail.state`). The entire value has to be a JSON path, not only part of it. For example, `RedshiftParameters.Sql` can be `$.detail.state` but it can't be `"SELECT * FROM $.detail.state"`. These paths are replaced dynamically at runtime with data from the event payload itself at the specified path. Dynamic path parameters can't reference new or transformed values resulting from input transformation. The supported syntax for dynamic parameter JSON paths is the same as when transforming input. For more information, see ???.

Dynamic syntax can be used on all string, non-enum fields of all EventBridge Pipes enrichment and target parameters except:

- `PipeTargetCloudWatchLogsParameters.LogStreamName`
- `PipeTargetEventBridgeEventBusParameters.EndpointId`
- `PipeEnrichmentHttpParameters.HeaderParameters`
- `PipeTargetHttpParameters.HeaderParameters`

For example, to set the `PartitionKey` of a pipe Kinesis target to a custom key from your source event, set the KinesisTargetParameter.PartitionKey to:

- `"$.data.`*`someKey`*`"` for a Kinesis source
- `"$.body.`*`someKey`*`"` for an Amazon SQS source

Then, if the event payload is a valid JSON string, such as `{"`*`someKey`*`":"`*`someValue`*`"}`, EventBridge extracts the value from the JSON path and uses it as the target parameter. In this example, EventBridge would set the Kinesis `PartitionKey` to "*someValue*".

Permissions

To make API calls on the resources that you own, EventBridge Pipes needs appropriate permission. EventBridge PIpes uses the IAM role that you specify on the pipe for enrichment and target calls using the IAM principal `pipes.amazonaws.com`.

Invoking targets

EventBridge has the following ways to invoke a target:

- **Synchronously** (invocation type set to `REQUEST_RESPONSE`) – EventBridge waits for a response from the target before proceeding.
- **Asynchronously** (invocation type set to `FIRE_AND_FORGET`) – EventBridge doesn't wait for a response before proceeding.

By default, for pipes with ordered sources, EventBridge invokes targets synchronously because a response from the target is needed before proceeding to the next event.

If an source doesn't enforce order, such as a standard Amazon SQS queue, EventBridge can invoke a supported target synchronously or asynchronously.

With Lambda functions and Step Functions state machines, you can configure the invocation type.

> **Note**
>
> For Step Functions state machines, Standard workflows must be invoked asynchronously.

Payload size limits

EventBridge Pipes supports payloads up to 6 MB. However, the effective payload size limit is determined by whichever is smaller: the Pipes limit of 6 MB or the target service's maximum payload size. For example:

- Lambda functions support payloads up to 6 MB, so the effective limit for a pipe targeting Lambda is 6 MB.
- EventBridge event buses support payloads up to 1 MB, so the effective limit for a pipe targeting an event bus is 1 MB.

- Step Functions state machines support payloads up to 256 KB, so the effective limit for a pipe targeting Step Functions is 256 KB.

When configuring your pipe, ensure your payload size, including any transformations applied by enrichment or input transformation, does not exceed the target's maximum payload size.

AWS Batch job queues target specifics

All AWS Batch `submitJob` parameters are configured explicitly with `BatchParameters`, and as with all Pipe parameters, these can be dynamic using a JSON path to your incoming event payload.

CloudWatch Logs group target specifics

Whether you use an input transformer or not, the event payload is used as the log message.
You can set the `Timestamp` (or the explicit `LogStreamName` of your destination) through `CloudWatchLogsParameters` in `PipeTarget`. As with all pipe parameters, these parameters can be dynamic when using a JSON path to your incoming event payload.

Amazon ECS task target specifics

All Amazon ECS `runTask` parameters are configured explicitly through `EcsParameters`. As with all pipe parameters, these parameters can be dynamic when using a JSON path to your incoming event payload.

Lambda functions and Step Functions workflow target specifics

Lambda and Step Functions do not have a batch API. To process batches of events from a pipe source, the batch is converted to a JSON array and passed to as input to the Lambda or Step Functions target. For more information, see ???.

Timestream for LiveAnalytics table target specifics

Considerations when specifying a Timestream for LiveAnalytics table as a pipe target include:

- Apache Kafka streams (including fromAmazon MSK or third-party providers) are not currently supported as a pipe source.
- If you have specified a Kinesis or DynamoDB stream as the pipe source, you must specify the number of retry attempts.

  For more information, see ???.

EventBridge event bus target specifics

When you configure an EventBridge event bus as a pipe target, the payload from your pipe is automatically placed in the `detail` section of the EventBridge event. Use `PipeTargetEventBridgeEventBusParameters` to configure the event's `source` and `detail-type` fields. Both fields support dynamic JSON path syntax to extract values from your event payload. For example, set `Source` to `$.body.source` or `DetailType` to `$.data.eventType`. You can also use input transformers to modify the event structure before it's placed in the `detail` field. For more information, see ???.

Amazon EventBridge Pipes batching and concurrency

Batching behavior

EventBridge Pipes supports batching from the source and to targets that support it. In addition, batching to enrichment is supported for AWS Lambda and AWS Step Functions. Because different services support different levels of batching, you can't configure a pipe with a larger batch size than the target supports. For example, Amazon Kinesis stream sources support a maximum batch size of 10,000 records, but Amazon Simple Queue Service supports a maximum of 10 messages per batch as a target. Therefore, a pipe from a Kinesis stream to an Amazon SQS queue can have a maximum configured batch size on the source of 10.

If you configure a pipe with an enrichment or target that doesn't support batching, you won't be able to activate batching on the source.

When batching is activated on the source, arrays of JSON records are passed through the pipe and then mapped to the batch API of a supported enrichment or target. Input transformers are applied separately on each individual JSON record in the array, not the array as a whole. For examples of these arrays, see ??? and select a specific source. Pipes will use the batch API for the supported enrichment or target even if the batch size is 1. If the enrichment or target doesn't have a batch API but receives full JSON payloads, such as Lambda and Step Functions, the entire JSON array is sent in one request. The request will be sent as a JSON array even if the batch size is 1.

If a pipe is configured for batching at the source, and the target supports batching, you can return an array of JSON items from your enrichment. This array can include a shorter or longer array than the original source. However, if the array is larger than the batch size supported by the target, the pipe won't invoke the target.

Supported batchable targets

Target	Maximum batch size
CloudWatch Logs	10,000
EventBridge event bus	10
Firehose stream	500
Kinesis stream	500
Lambda function	customer defined
Step Functions state machine	customer defined
Amazon SNS topic	10
Amazon SQS queue	10

The following enrichments and targets receive the full batch event payload for processing and are constrained by the total payload size of the event, rather than the size of the batch:

- Step Functions state machine (262144 characters)
- Lambda function (6MB)

Partial batch failure

For Amazon SQS and stream sources, such as Kinesis and DynamoDB, EventBridge Pipes supports partial batch failure handling of target failures. If the target supports batching and only part of the batch succeeds, EventBridge automatically retries batching the remainder of the payload. For the most up-to-date enriched content, this retry occurs through the entire pipe, including re-invoking any configured enrichment.

Partial batch failure handling of the enrichment is not supported.

For Lambda and Step Functions targets, you can also specify a partial failure by returning a payload with defined structure from the target. This indicates events that need to be retried.

Example partial failure payload structure

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": "id2"
    },
    {
      "itemIdentifier": "id4"
    }
]
```

In the example, the `itemIdentifier` match the ID of the events handled by your target from their original source. For Amazon SQS, this is the `messageId`. For Kinesis and DynamoDB, this is the `eventID`. For EventBridge Pipes to adequately handle partial batch failures from the targets, these fields need to be included in any array payload returned by the enrichment.

Throughput and concurrency behavior

Every event or batch of events received by a pipe that travel to an enrichment or target is considered a pipe *execution*. A pipe in `STARTED` state continuously polls for events from the source, scaling up and down depending on the available backlog and configured batching settings.

For quotas on concurrent pipe executions, and number of pipes per account and Region, see ???.

By default, a single pipe will scale to the following maximum concurrent executions, depending on the source:

- **DynamoDB** – The concurrent executions can climb as high as the `ParallelizationFactor` configured on the pipe multiplied by the number of shards in the stream.
- **Apache Kafka** – The concurrent executions can climb as high the number of partitions on the topic, up to 1000.
- **Kinesis** – The concurrent executions can climb as high as the `ParallelizationFactor` configured on the pipe multiplied by the number of shards in the stream.
- **Amazon MQ** – 5
- **Amazon SQS** – 1250

If you have requirements for higher maximum polling throughputs or concurrency limits, contact support.

> **Note**
>
> The execution limits are considered best-effort safety limitations. Although polling isn't throttled below these values, a pipe or account might burst higher than these recommend values.

Pipe executions are limited to a maximum of 5 minutes including the enrichment and target processing. This limit currently can't be increased.

Pipes with strictly ordered sources, such as Amazon SQS FIFO queues, Kinesis and DynamoDB Streams, or Apache Kafka topics) are further limited in concurrency by the configuration of the source, such as the number of message group IDs for FIFO queues or the number of shards for Kinesis queues. Because ordering is strictly guaranteed within these constraints, a pipe with an ordered source can't exceed those concurrency limits.

Amazon EventBridge Pipes input transformation

Amazon EventBridge Pipes support optional input transformers when passing data to the enrichment and the target. You can use Input transformers to reshape the JSON event input payload to serve the needs of the enrichment or target service. For Amazon API Gateway and API destinations, this is how you shape the input event to the RESTful model of your API. Input transformers are modeled as an `InputTemplate` parameter. They can be free text, a JSON path to the event payload, or a JSON object that includes inline JSON paths to the event payload. For enrichment, the event payload is coming from the source. For targets, the event payload is what is returned from the enrichment, if one is configured on the pipe. In addition to the service-specific data in the event payload, you can use reserved variables in your `InputTemplate` to reference data for your pipe.

To access items in an array, use square bracket notation.

> **Note**
>
> EventBridge does not support all JSON Path syntax and evaluate it at runtime. Supported syntax includes:
>
> - dot notation (for example,`$.detail`)
> - dashes

- underscores
- alphanumeric characters
- array indices
- wildcards (*)
- forward slashes

The following are sample `InputTemplate` parameters referencing an Amazon SQS event payload:

Static string

```
InputTemplate: "Hello, sender"
```

JSON Path

```
InputTemplate: <$.attributes.SenderId>
```

Dynamic string

```
InputTemplate: "Hello, <$.attributes.SenderId>"
```

Static JSON

```
InputTemplate: >
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3",
}
```

Dynamic JSON

```
InputTemplate: >
{
  "key1": "value1"
  "key2": <$.body.key>,
  "d": <aws.pipes.event.ingestion-time>
}
```

Using square bracket notation to access an item in an array:

```
InputTemplate: >
{
  "key1": "value1"
  "key2": <$.body.Records[3]>,
  "d": <aws.pipes.event.ingestion-time>
}
```

> **Note**
>
> EventBridge replaces input transformers at runtime to ensure a valid JSON output. Because of this, put quotes around variables that refer to JSON path parameters, but do not put quotes around variables that refer to JSON objects or arrays.

Reserved variables

Input templates can use the following reserved variables:

- `<aws.pipes.pipe-arn>` – The Amazon Resource Name (ARN) of the pipe.
- `<aws.pipes.pipe-name>` – The name of the pipe.
- `<aws.pipes.source-arn>` – The ARN of the event source of the pipe.
- `<aws.pipes.enrichment-arn>` – The ARN of the enrichment of the pipe.
- `<aws.pipes.target-arn>` – The ARN of the target of the pipe.
- `<aws.pipes.event.ingestion-time>` – The time at which the event was received by the input transformer. This is an ISO 8601 timestamp. This time is different for the enrichment input transformer and the target input transformer, depending on when the enrichment completed processing the event.
- `<aws.pipes.event>` – The event as received by the input transformer.

  For an enrichment input transformer, this is the event from the source. This contains the original payload from the source, plus additional service-specific metadata. See the topics in ??? for service-specific examples.

  For a target input transformer, this is the event returned by the enrichment, if one is configured, with no additional metadata. As such, an enrichment-returned payload may be non-JSON. If no enrichment is configured on the pipe, this is the event from the source with metadata.

- <aws.pipes.event.json> – The same as aws.pipes.event, but the variable only has a value if the original payload, either from the source or returned by the enrichment, is JSON. If the pipe has an encoded field, such as the Amazon SQS body field or the Kinesis data, those fields are decoded and turned into valid JSON. Because it isn't escaped, the variable can only be used as a value for a JSON field. For more information, see ???.

Input transform example

The following is an example Amazon EC2 event that we can use as our *sample event*.

```
{
  "version": "0",
  "id": "7bf73129-1428-4cd3-a780-95db273d1602",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "123456789012",
  "time": "2015-11-11T21:29:54Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:ec2:us-east-1:123456789012:instance/i-abcd1111"
  ],
  "detail": {
    "instance-id": "i-0123456789",
    "state": "RUNNING"
  }
}
```

Let's use the following JSON as our *Transformer*.

```
{
  "instance" : <$.detail.instance-id>,
  "state": <$.detail.state>,
  "pipeArn" : <aws.pipes.pipe-arn>,
  "pipeName" : <aws.pipes.pipe-name>,
  "originalEvent" : <aws.pipes.event.json>
}
```

The following will be the resulting *Output*:

```
{
  "instance" : "i-0123456789",
  "state": "RUNNING",
  "pipeArn" : "arn:aws:pipe:us-east-1:123456789012:pipe/example",
  "pipeName" : "example",
  "originalEvent" : {
    ... // commented for brevity
  }
}
```

Implicit body data parsing

The following fields in the incoming payload may be JSON-escaped, such as the Amazon SQS body object, or base64-encoded, such as the Kinesis `data` object. For both filtering and input transformation, EventBridge transforms these fields into valid JSON so sub-values can be referenced directly. For example, `<$.data.someKey>` for Kinesis.

To have the target receive the original payload without any additional metadata, use an input transformer with this body data, specific to the source. For example, `<$.body>` for Amazon SQS, or `<$.data>` for Kinesis. If the original payload is a valid JSON string (for example `{"key": "value"}`), then use of the input transformer with source specific body data will result in the quotes within the original source payload being removed. For example, `{"key": "value"}` will become `"{key: value}"` when delivered to the target. If your target requires valid JSON payloads (for example, EventBridge Lambda or Step Functions), this will cause delivery failure. To have the target receive the original source data without generating invalid JSON, wrap the source body data input transformer in JSON. For example, `{"data": <$.data>}`.

Implicit body parsing can also be used to dynamically populate values for most pipe target or enrichment parameters. For more information, see ???

> **Note**
>
> If the original payload is valid JSON, this field will contain the unescaped, non-base64-encoded JSON. However, if the payload is not valid JSON, EventBridge base64-encodes for the fields listed below, with the exception of Amazon SQS.

- **Active MQ** – `data`
- **Kinesis** – `data`

- **Amazon MSK** – `key` and `value`
- **Rabbit MQ** – `data`
- **Self managed Apache Kafka;** – `key` and `value`
- **Amazon SQS** – `body`

Common issues with transforming input

These are some common issues when transforming input in EventBridge pipes:

- For Strings, quotes are required.
- There is no validation when creating JSON path for your template.
- If you specify a variable to match a JSON path that doesn't exist in the event, EventBridge replaces that variable with an empty string. EventBridge includes the key in the output with an empty-string value instead of omitting it.
- JSON properties like `aws.pipes.event.json` can only be used as the value of a JSON field, not inline in other strings.
- EventBridge doesn't escape values extracted by *Input Path*, when populating the *Input Template* for a target.
- If a JSON path references a JSON object or array, but the variable is referenced in a string, EventBridge removes any internal quotes to ensure a valid string. For example, "Body is < $.body>" would result in EventBridge removing quotes from the object.

  Therefore, if you want to output a JSON object based on a single JSON path variable, you must place it as a key. In this example, `{"body": <$.body>}`.
- Quotes are not required for variables that represent strings. They are permitted, but EventBridge Pipes automatically adds quotes to string variable values during transformation, to ensure the transformation output is valid JSON. EventBridge Pipes does not add quotes to variables that represent JSON objects or arrays. Do not add quotes for variables that represent JSON objects or arrays.

  For example, the following input template includes variables that represent both strings and JSON objects:

```
{
  "pipeArn" : <aws.pipes.pipe-arn>,
  "pipeName" : <aws.pipes.pipe-name>,
```

```
  "originalEvent" : <aws.pipes.event.json>
}
```

Resulting in valid JSON with proper quotation:

```
{
  "pipeArn" : "arn:aws:events:us-east-2:123456789012:pipe/example",
  "pipeName" : "example",
  "originalEvent" : {
    ... // commented for brevity
  }
}
```

- For Lambda or Step Functions enrichments or targets, batches are delivered to the target as JSON arrays, even if the batch size is 1. However, input transformers will still be applied to individual records in the JSON Array, not the array as a whole. For more information, see ???.
- Input transformers and filtering can extract JSON values that have been string-encoded once, but not values that have been string-encoded twice. This commonly occurs when an Amazon SNS message is sent to Amazon SQS. When Amazon SQS receives the Amazon SNS message, it stringifies the entire message. When Pipes then receives this Amazon SQS message, the Amazon SNS message content appears in the `body` field and is accessible. However, if the Amazon SNS `Message` field itself contains stringified JSON, that nested content is double-encoded and cannot be accessed by input transformers or filters. For example, `<$.body.TopicArn>` is accessible, but `<$.body.Message.operation>` is not if the `Message` field contains stringified JSON such as `"{\\\"operation\\\":\\\"UPDATE\\\",\\\"email\\\":\\\"user@example.com\\\"}"`.

  To work around this limitation, use an enrichment step with a Lambda function to parse the double-encoded content and extract the nested values. For more information about enrichment, see Enrichment.

Logging Amazon EventBridge Pipes performance

EventBridge Pipes logging enables you to have EventBridge Pipes send records detailing pipe performance to supported AWS services. Use logs to gain insight into your pipe's execution performance, and to help with troubleshooting and debugging.

You can select the following AWS services as *log destinations* to which EventBridge Pipes delivers records:

- CloudWatch Logs

  EventBridge delivers log records to the specified CloudWatch Logs log group.

  Use CloudWatch Logs to centralize the logs from all of your systems, applications, and AWS services that you use, in a single, highly scalable service. For more information, see Working with log groups and log streams in the *Amazon CloudWatch Logs User Guide*.
- Firehose stream logs

  EventBridge delivers log records to a Firehose delivery stream.

  Amazon Data Firehose is a fully-managed service for delivering real-time streaming data to destinations such as certain AWS services, as well as any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers. For more information, see Creating an Amazon Data Firehose delivery stream in the *Amazon Data Firehose User Guide*.
- Amazon S3 logs

  EventBridge delivers log records as Amazon S3 objects to the specified bucket.

  Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. For more information, see Uploading, downloading, and working with objects in Amazon S3 in the *Amazon Simple Storage Service User Guide*.

How Amazon EventBridge Pipes logging works

A pipe *execution* is an event or batch of events received by a pipe that travel to an enrichment and/or target. If enabled, EventBridge generates a log record for each execution step it performs as the event batch is processed. The information contained in the record applies to the event batch, be it a single event or up to 10,000 events.

You can configure the size of the event batch on the pipe source and target. For more information, see ???.

The record data sent to each log destination is the same.

If a Amazon CloudWatch Logs destination is configured, the log records delivered to all destinations have a limit of 256kb. Fields will be truncated as necessary.

You can customize the records EventBridge sends to the selected log destinations in the following way:

- You can specify the *log level*, which determines the execution steps for which EventBridge sends records to the selected log destinations. For more information, see ???.
- You can specify whether EventBridge Pipes includes execution data in records for execution steps where it is relevant. This data includes:
  - The payload of the event batch
  - The request sent to the AWS enrichment or target service
  - The response returned by the AWS enrichment or target service

  For more information, see ???.

Specifying EventBridge Pipes log level

You can specify the types of execution steps for which EventBridge sends records to the selected log destinations.

Choose from the following levels of detail to include in log records. The log level applies to all log destinations specified for the pipe. Each log level includes the execution steps of the previous log levels.

- **OFF** – EventBridge does not send any records to any specified log destinations. This is the default setting.
- **ERROR** – EventBridge sends any records related to errors generated during pipe execution to the specified log destinations.
- **INFO** – EventBridge sends any records related to errors, as well as select other steps performed during pipe execution to the specified log destinations.
- **TRACE** – EventBridge sends any records generated during any steps in the pipe execution to the specified log destinations.

In the EventBridge console, CloudWatch logs is selected as a log destination by default, as is the ERROR log level. So, by default, EventBridge Pipes creates a new CloudWatch log group to which it sends log records containing the ERROR level of detail. No default is selected when you configure logs programmatically.

The following table lists the execution steps included in each log level.

Step	TRACE	INFO	ERROR	OFF
Execution Failed	x	x	x	
Execution Partially Failed	x	x	x	
Execution Started	x	x		
Execution Succeeded	x	x		
Execution Throttled	x	x	x	
Execution Timeout	x	x	x	
Enrichment Invocation Failed	x	x	x	
Enrichment Invocation Skipped	x	x		
Enrichment Invocation Started	x			
Enrichment Invocation Succeeded	x			
Enrichment Stage Entered	x	x		
Enrichment Stage Failed	x	x	x	
Enrichment Stage Succeeded	x	x		
Enrichment Transformation Failed	x	x	x	
Enrichment Transformation Started	x			
Enrichment Transformation Succeeded	x			

Step	TRACE	INFO	ERROR	OFF
Target Invocation Failed	x	x	x	
Target Invocation Partially Failed	x	x	x	
Target Invocation Skipped	x			
Target Invocation Started	x			
Target Invocation Succeeded	x			
Target Stage Entered	x	x		
Target Stage Failed	x	x	x	
Target Stage Partially Failed	x	x	x	
Target Stage Skipped	x			
Target Stage Succeeded	x	x		
Target Transformation Failed	x	x	x	
Target Transformation Started	x			
Target Transformation Succeeded	x			

Including execution data in EventBridge Pipes logs

You can specify for EventBridge to include *execution data* in the records it generates. Execution data includes fields representing the event batch payload, as well as the request sent to and the response from the enrichment and target.

Execution data is useful for troubleshooting and debugging. The `payload` field contains the actual contents of each event included in the batch, enabling you to correlate individual events to a specific pipe execution.

If you choose to include execution data, it is included for all log destinations specified for the pipe.

> **Important**
>
> These fields may contain sensitive information. EventBridge makes no attempt to redact the contents of these fields during logging.

When including execution data, EventBridge adds the following fields to the relevant records:

- **`payload`**

  Represents the contents of the event batch being processed by the pipe.

  EventBridge includes the `payload` field in records generated at steps where the event batch contents may have been updated. This includes the following steps:

  - `EXECUTION_STARTED`
  - `ENRICHMENT_TRANSFORMATION_SUCCEEDED`
  - `ENRICHMENT_STAGE_SUCCEEDED`
  - `TARGET_TRANSFORMATION_SUCCEEDED`
  - `TARGET_STAGE_SUCCEEDED`

- **`awsRequest`**

  Represents the request sent to the enrichment or target as a JSON string. For requests sent to an API destination, this represents the HTTP request sent to that endpoint.

  EventBridge includes the `awsRequest` field in records generated at the final steps of enrichment and targeting; that is, after EventBridge has executed or attempted to execute the request against the specified enrichment or target service. This includes the following steps:

  - `ENRICHMENT_INVOCATION_FAILED`
  - `ENRICHMENT_INVOCATION_SUCCEEDED`
  - `TARGET_INVOCATION_FAILED`
  - `TARGET_INVOCATION_PARTIALLY_FAILED`
  - `TARGET_INVOCATION_SUCCEEDED`

- **`awsResponse`**

Represents the response returned by the enrichment or target, in JSON format. For requests sent to an API destination, this represents the HTTP response returned from that endpoint.

As with `awsRequest`, EventBridge includes the `awsResponse` field in records generated at the final steps of enrichment and targeting; that is, after EventBridge has executed or attempted to execute a request against the specified enrichment or target service and received a response. This includes the following steps:

- `ENRICHMENT_INVOCATION_FAILED`
- `ENRICHMENT_INVOCATION_SUCCEEDED`
- `TARGET_INVOCATION_FAILED`
- `TARGET_INVOCATION_PARTIALLY_FAILED`
- `TARGET_INVOCATION_SUCCEEDED`

For a discussion of pipe execution steps, see ???.

Truncating execution data in EventBridge Pipes log records

If you choose to have EventBridge include execution data in a pipe's log records, there is a possibility that a record may exceed the 256 KB size limit. To prevent this, EventBridge automatically truncates the execution data fields, in the following order. EventBridge truncates each field entirely before progressing to truncate the next field. EventBridge truncates field data simply by removing characters from the end of the data string; no attempt is made to truncate based on data importance, and truncation will invalidate JSON formatting.

- `payload`
- `awsRequest`
- `awsResponse`

If EventBridge does truncate fields in the event, the `truncatedFields` field includes a list of the truncated data fields.

Error reporting in EventBridge Pipes log records

EventBridge also includes error data, where available, in pipe execution steps that represent failure states. These steps include:

- `ExecutionThrottled`
- `ExecutionTimeout`
- `ExecutionFailed`
- `ExecutionPartiallyFailed`
- `EnrichmentTransformationFailed`
- `EnrichmentInvocationFailed`
- `EnrichmentStageFailed`
- `TargetTransformationFailed`
- `TargetInvocationFailed`
- `TargetInvocationPartiallyFailed`
- `TargetStageFailed`
- `TargetStagePartiallyFailed`

EventBridge Pipes execution steps

Understanding the flow of pipe execution steps can aid you in troubleshooting or debugging your pipe's performance using logs.

A pipe *execution* is an event or batch of events received by a pipe that travel to an enrichment or target. If enabled, EventBridge generates a log record for each execution step it performs as the event batch is processed.

At a high level, the execution contains two *stages*, or collection of steps: enrichment, and target. Each of these stages consists of transformation and invocation steps.

The main steps of a successful pipe execution follows this flow:

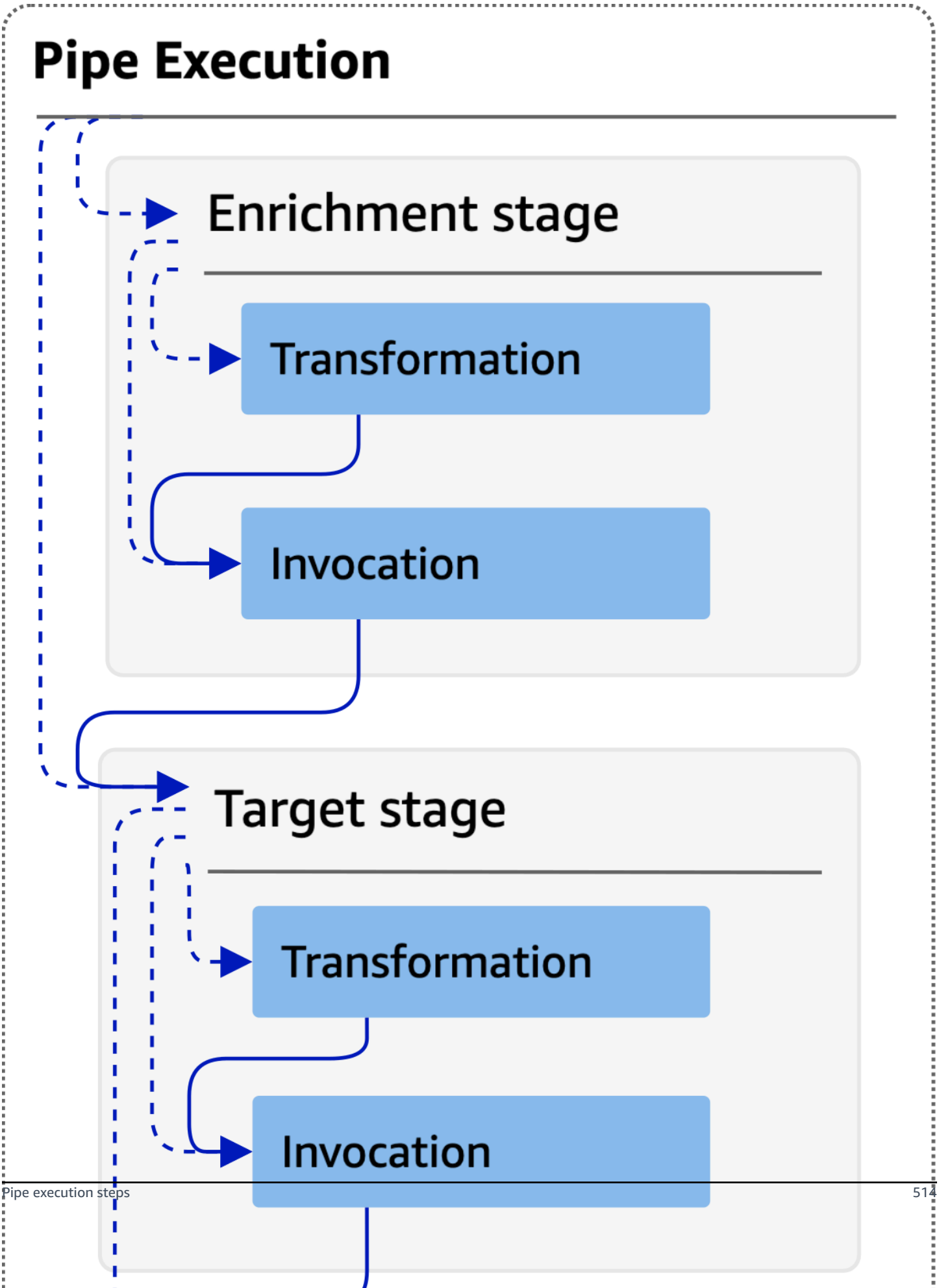
- The pipe execution starts.
- The execution enters the enrichment stage if you have specified an enrichment for the events. If you haven't specified an enrichment, the execution proceeds to the target stage.

  In the enrichment stage, the pipe performs any transformation you have specified, then invokes the enrichment.
- In the target stage, the pipe performs any transformation you have specified, then invokes the target.

If you haven't specified transformation or target, the execution skips the target stage.

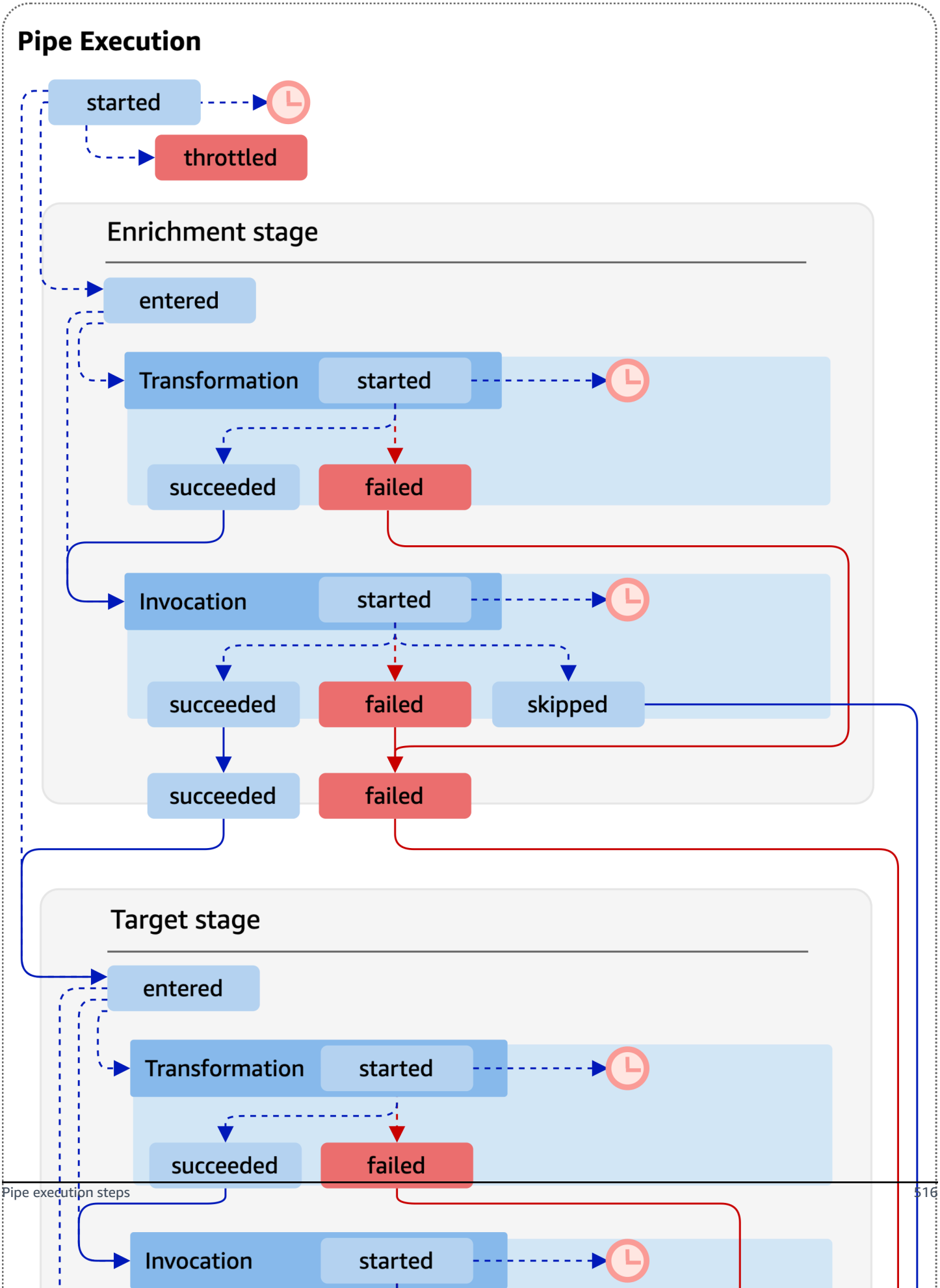
- The pipe execution completes successfully.

The diagram below demonstrates this flow. Diverging paths are formatted as dotted lines.


Pipe Execution
Enrichment stage
Transformation
Invocation
Target stage
Transformation
Invocation

The diagram below presents a detailed view of the pipe execution flow, with all possible execution steps represented. Again, diverging paths are formatted as dotted lines

For a complete list of pipe execution steps, see ???.


Pipe Execution
started
throttled
Enrichment stage
entered
Transformation
started
succeeded
failed
Invocation
started
succeeded
failed
skipped
succeeded
failed
Target stage
entered
Transformation
started
succeeded
failed
Invocation
started

Note that target invocation may result in a partial failure of the batch. For more information, see ???.

EventBridge Pipes log schema reference

The following reference details the schema for EventBridge Pipes log records.

Each log record represents a pipe execution step, and may contain up to 10,000 events if the pipe source and target have been configured for batching.

For more information, see ???.

```
{
    "executionId": "guid",
    "timestamp": "date_time",
    "messageType": "execution_step",
    "resourceArn": "arn:aws:pipes:region:account:pipe/pipe-name",
    "logLevel": "TRACE | INFO | ERROR",
    "payload": "{}",
    "awsRequest": "{}"
    "awsResponse":"{}"
    "truncatedFields": ["awsRequest","awsResponse","payload"],
    "error": {
        "httpStatusCode": code,
        "message": "error_message",
        "details": "",
        "awsService": "service_name",
        "requestId": "service_request_id"
    }
}
```

executionId

The ID of the pipe execution.

A pipe execution is an event or batch of events received by a pipe that travel to an enrichment or target. For more information, see ???.

timestamp

The date and time the log event was emitted.

Unit: millisecond

messageType

The pipe execution step for which the record was generated.

For more information on pipe execution steps, see ???.

resourceArn

The Amazon Resource Name (ARN) for the pipe.

logLevel

The level of detail specified for the pipe log.

Valid values: `ERROR` | `INFO` | `TRACE`

For more information, see ???.

payload

The contents of the event batch being processed by the pipe.

EventBridge includes this field only if you have specified to include execution data in the logs for this pipe. For more information, see ???

> **Important**
>
> These fields may contain sensitive information. EventBridge makes no attempt to redact the contents of these fields during logging.

For more information, see ???.

awsRequest

The request sent to the enrichment or target, in JSON format. For requests sent to an API destination, this represents the HTTP request sent to that endpoint.

EventBridge includes this field only if you have specified to include execution data in the logs for this pipe. For more information, see ???

> **Important**
>
> These fields may contain sensitive information. EventBridge makes no attempt to redact the contents of these fields during logging.

For more information, see ???.

awsResponse

The response returned by the enrichment or target, in JSON format. For requests sent to an API destination, this represents the HTTP response returned from that endpoint, and not the response returned by the API Destination service itself.

EventBridge includes this field only if you have specified to include execution data in the logs for this pipe. For more information, see ???

> **Important**
>
> These fields may contain sensitive information. EventBridge makes no attempt to redact the contents of these fields during logging.

For more information, see ???.

truncatedFields

A list of any execution data fields EventBridge has truncated to keep the record below the 256 KB size limitation.

If EventBridge did not have to truncate any of the execution data fields, this field is present but `null`.

For more information, see ???.

error

Contains information for any error generated during this pipe execution step.

If no error was generated during this pipe execution step, this field is present but `null`.

httpStatusCode

The HTTP status code returned by the called service.

message

The error message returned by the called service.

details

Any detailed error information returned by the called service.

awsService

The name of the service called.

requestId

The request ID for this request from the called service.

Logging and monitoring Amazon EventBridge Pipes using Amazon CloudWatch Logs

You can log EventBridge Pipes invocations using CloudTrail and monitor the health of your pipes using CloudWatch metrics.

CloudWatch metrics

EventBridge Pipes sends metrics to Amazon CloudWatch every minute for everything from a pipe executions being throttled to a target successfully being invoked.

Best-effort CloudWatch metrics delivery

CloudWatch metrics are delivered on a best-effort basis. Most EventBridge Pipes operations result in a data point being sent to CloudWatch. The completeness and timeliness of metrics are not guaranteed. A data point for a given minute might be delayed before being available through CloudWatch, or it might not be delivered at all. CloudWatch metrics give you an idea of the nature of activity in near-real time. They are not meant to be a complete accounting of all operations.

Metric	Description	Dimensions	Units
`Concurrency`	The number of concurrent executions of a pipe.	AwsAccountId	None
`Duration`	Length of time the pipe execution took.	PipeName	Milliseconds
`EventCount`	The number of events a pipe has processed.	PipeName	None

Metric	Description	Dimensions	Units
`EventSize`	The size of the payload of the event that invoked the pipe.	PipeName	Bytes
`ExecutionThrottled`	How many executions of a pipe were throttled. **Note** This value will be `0` if no executions were throttled.	AwsAccountId, PipeName	None
`ExecutionTimeout`	How many executions of a pipe timed out before completing execution. **Note** This value will be `0` if no executions timed out.	PipeName	None
`ExecutionFailed`	How many executions of a pipe failed. **Note** This value will be `0` if no executions failed.	PipeName	None
`ExecutionPartiallyFailed`	How many executions of a pipe partially failed. **Note** This value will be `0` if no executions partially failed.	PipeName	None

Metric	Description	Dimensions	Units
`EnrichmentStageDuration`	How long the enrichment stage took to complete.	PipeName	Milliseconds
`EnrichmentStageFailed`	How many executions of a pipe's enrichment stage failed. **Note** This value will be `0` if no executions failed.	PipeName	None
`Invocations`	Total number of invocations.	AwsAccountId, PipeName	None
`TargetStageDuration`	How long the target stage took to complete.	PipeName	Milliseconds
`TargetStageFailed`	How many executions of a pipe's target stage failed. **Note** This value will be `0` if no executions failed.	PipeName	None

Metric	Description	Dimensions	Units
`TargetStagePartiallyFailed`	How many executions of a pipe's target stage partially failed. **Note** This value will be `0` if no target stage executions partially failed.	PipeName	None
`TargetStageSkipped`	How many executions of a pipe's target stage were skipped (for example, due to the enrichment returning an empty payload).	PipeName	Count

Dimensions for CloudWatch metrics

CloudWatch metrics have *dimensions*, or sortable attributes, which are listed below.

Dimension	Description
`AwsAccountId`	Filters the available metrics by account ID.
`PipeName`	Filters the available metrics by pipe name.

Amazon EventBridge Pipes error handling and troubleshooting

Understanding the types of errors EventBridge Pipes may encounter, and how EventBridge handles those errors, can help you troubleshoot issues with your pipes.

Retry behavior and error handling

EventBridge Pipes automatically retries enrichment and target invocation on any retryable AWS failures with the source service, the enrichment or target services, or EventBridge. However, if there are failures returned by enrichment or target customer implementations, the pipe polling throughput will gradually back off. For nearly continuous 4xx errors (such as authorization

problems with IAM or missing resources), the pipe can be automatically disabled with an explanatory message in the `StateReason`.

Pipe invocation errors and retry behavior

When you invoke a pipe, two main types of errors can occur: *pipe internal errors* and *customer invocation errors*.

Pipe internal errors

Pipe internal errors are errors resulting by aspects of the invocation managed by the EventBridge Pipes service.

These types of errors can include issues such as:

- A HTTP connection failure when attempting to invoke the customer target service
- A transient drop in availability on the pipe service itself.

In general, EventBridge Pipes retries internal errors an indefinite number of times, and stops only when the record expires in the source.

For pipes with a stream source, EventBridge Pipes does not count retries for internal errors against the maximum number of retries specified on the retry policy for the stream source. For pipes with an Amazon SQS source, EventBridge Pipes does not count retries for internal errors against the maximum receive count for the Amazon SQS source.

Customer invocation errors

Customer invocation errors are errors resulting from configuration or code managed by the user.

These types of errors can include issues such as:

- Insufficient permissions on the pipe to invoke the target.
- A logic error in a synchronously-invoked customer Lambda, Step Functions, API destination, or API Gateway endpoint.

For customer invocation errors, EventBridge Pipes does the following:

- For pipes with a stream source, EventBridge Pipes retries up to the maximum retry times configured on the pipe retry policy or until the maximum record age expires, whichever comes first.
- For pipes with an Amazon SQS source, EventBridge Pipes retries a customer error up to the maximum receive count on the source queue.
- For pipes with a Apache Kafka or Amazon MQ source, EventBridge retries customer errors the same as it retries internal errors.

For pipes with compute targets, you must invoke the pipe synchronously in order for EventBridge Pipes to be aware of any runtime errors that are thrown from the customer compute logic and retry on such errors. Pipes cannot retry on errors thrown from the logic of a Step Functions standard workflow, as this target must be invoked asynchronously.

For Amazon SQS and stream sources, such as Kinesis and DynamoDB, EventBridge Pipes supports partial batch failure handling of target failures. For more information, see Partial batch failure.

Amazon SQS source retry timing with maxBatchWindowInSeconds

When using an Amazon SQS queue as a pipe source, the retry timing behavior differs depending on whether you configure the `maxBatchWindowInSeconds` parameter:

- **Without maxBatchWindowInSeconds** – The SQS poller uses the queue's visibility timeout setting for retries. When processing fails, messages remain hidden for the visibility timeout duration before becoming available for retry. To reduce retry delays, configure the queue's visibility timeout to an appropriate value for your use case.
- **With maxBatchWindowInSeconds** – The SQS poller dynamically sets the visibility timeout for polled messages using the formula: `functionTimeout + maxBatchWindowInSeconds + 30 seconds`. For EventBridge Pipes, the function timeout is 7 minutes, resulting in a visibility timeout of approximately 7.5 minutes plus your configured `maxBatchWindowInSeconds` value. When processing fails, messages remain hidden for this extended duration before becoming available for retry.

This behavior is particularly relevant when using partial batch responses. If you require faster retry timing, avoid setting `maxBatchWindowInSeconds` and instead rely on the queue's configured visibility timeout.

Pipe DLQ behavior

A pipe inherits dead-letter queue (DLQ) behavior from the source:

- If the source Amazon SQS queue has a configured DLQ, messages are automatically delivered there after the specified number of attempts.
- For stream sources, such as DynamoDB and Kinesis streams, you can configure a DLQ for the pipe and route events. DynamoDB and Kinesis stream sources support Amazon SQS queues and Amazon SNS topics as DLQ targets.

If you specify a `DeadLetterConfig` for a pipe with a Kinesis or DynamoDB source, make sure that the `MaximumRecordAgeInSeconds` property on the pipe is less than the `MaximumRecordAge` of the source event. `MaximumRecordAgeInSeconds` controls when the pipe poller will give up on the event and deliver it to the DLQ and the `MaximumRecordAge` controls how long the message will be visible in the source stream before it gets deleted. Therefore, set `MaximumRecordAgeInSeconds` to a value that is less than the source `MaximumRecordAge` so that there's adequate time between when the event gets sent to the DLQ, and when it gets automatically deleted by the source for you to determine why the event went to the DLQ.

The `MaximumRecordAgeInSeconds` parameter applies independently of retry behavior. When polling a stream source, if a record's age exceeds the `MaximumRecordAgeInSeconds` value, EventBridge Pipes will not process that record, regardless of whether a retry situation exists. These records are sent directly to the DLQ (if configured) without any processing attempt.

For Amazon MQ sources, the DLQ can be configured directly on the message broker.

EventBridge Pipes does not support first-in first-out (FIFO) DLQs for stream sources.

EventBridge Pipes does not support DLQ for Amazon MSK stream and Self managed Apache Kafka stream sources.

Pipe failure states

Creating, deleting, and updating pipes are asynchronous operations that might result in a failure state. Likewise, a pipe might be automatically disabled due to failures. In all cases, the pipe `StateReason` provides information to help troubleshoot the failure.

The following is a sample of the possible `StateReason` values:

- Stream not found. To resume processing please delete the pipe and create a new one.

- Pipes does not have required permissions to perform Queue operations (sqs:ReceiveMessage, sqs:DeleteMessage and sqs:GetQueueAttributes)
- Connection error. Your VPC must be able to connect to pipes. You can provide access by configuring a NAT Gateway or a VPC Endpoint to pipes-data. For how to setup NAT gateway or VPC Endpoint to pipes-data, please check AWS documentation.
- MSK cluster does not have security groups associated with it

A pipe may be automatically stopped with an updated `StateReason`. Possible reasons include:

- A Step Functions standard workflow configured as an enrichment.
- A Step Functions standard workflow configured as as a target to be invoked synchronously.

Custom encryption failures

If you configure a source to use an AWS KMS custom encryption key (CMK), rather than an AWS-managed AWS KMS key, you must explicitly give your pipe's Execution Role decryption permission. To do so, include the following additional permission in the custom CMK policy:

```
  {
      "Sid": "Allow Pipes access",
      "Effect": "Allow",
      "Principal": {
          "AWS": "arn:aws:iam::01234567890:role/service-role/
Amazon_EventBridge_Pipe_DDBStreamSourcePipe_12345678"
      },
      "Action": "kms:Decrypt",
      "Resource": "*"
  }
```

Replace the above role with your pipe's Execution Role.

Next, ensure that the same permissions for KMS are added to your Pipe execution role.

This is true for all pipe sources with AWS KMS CMK, including:

- Amazon DynamoDB Streams
- Amazon Kinesis Data Streams
- Amazon MQ

- Amazon MSK
- Amazon SQS

Tutorial: Create an EventBridge pipe that filters source events

In this tutorial, you'll create a pipe that connects a DynamoDB stream source to an Amazon SQS queue target. This includes specifying an event pattern for the pipe to use when filtering events to deliver to the queue. You'll then test the pipe to ensure that only the desired events are being delivered.

Prerequisites: Create the source and target

Before you create the pipe, you'll need to create the source and target that the pipe is to connect. In this case, an Amazon DynamoDB data stream to act as the pipe source, and an Amazon SQS queue as the pipe target.

To simplify this step, you can use AWS CloudFormation to deploy the source and target resources. To do this, you'll create a CloudFormation template defining the following resources:

- The pipe source

  An Amazon DynamoDB table, named `pipe-tutorial-source`, with a stream enabled to provide an ordered flow of information about changes to items in the DynamoDB table.
- The pipe target

  An Amazon SQS queue, named `pipe-tutorial-target`, to receive the DynamoDB stream of events from your pipe.

To create the CloudFormation template for provisioning pipe resources

1. Copy the JSON template text in the ??? section, below.
2. Save the template as a JSON file (for example, `~/pipe-tutorial-resources.json`).

Next, use the template file you just created to provision a CloudFormation stack.

> **Note**
>
> Once you create your CloudFormation stack, you will be charged for the AWS resources it provisions.

Provision the tutorial prerequisites using the AWS CLI

- Run the following CLI command, where `--template-body` specifies the location of your template file:

```
aws cloudformation create-stack --stack-name pipe-tuturial-resources --template-body file://~/pipe-tutorial-resources.json
```

Provision tutorial prerequisites using the CloudFormation console

1. Open the CloudFormation console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudformation.
2. Select **Stacks**, then select **Create stack**, and choose **with new resources (standard)**.

   CloudFormation displays the **Create stack** wizard.
3. For **Prerequisite - Prepare template**, leave the default, **Template is ready**, selected.
4. Under **Specify template**, select **Upload a template file**, and then choose the file and select **Next**.
5. Configure the stack and the resources it will provision:
   - For **Stack name**, enter `pipe-tuturial-resources`.
   - For **Parameters**, leave the default names for the DynamoDB table and Amazon SQS queue.
   - Choose **Next**.
6. Choose **Next**, then choose **Submit**.

   CloudFormation creates the stack and provisions the resources defined in the template.

For more information about CloudFormation, see What is CloudFormation? in the *CloudFormation User Guide*.

Step 1: Create the pipe

With the pipe source and target provisioned, you can now create the pipe to connect the two services.

Create the pipe using the EventBridge console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. On the navigation pane, choose **Pipes**.
3. Choose **Create pipe**.
4. For **Name**, name your pipe `pipe-tutorial`.
5. Specify the DynamoDB data stream source:
   a. Under **Details**, for **Source**, select **DynamoDB data stream** .

      EventBridge displays DynamoDB-specific source configuration settings.
   b. For **DynamoDB stream**, select `pipe-tutorial-source`.

      Leave **Starting position** set to the default, `Latest`.
   c. Choose **Next**.
6. Specify and test an event pattern to filter events:

   Filtering enables you to control which events the pipes sends to enrichment or the target. The pipe only sends events that match the event pattern on to enrichment or the target.

   For more information, see ???.

   > **Note**
   >
   > You are only billed for those events sent to enrichment or the target.

   a. Under **Sample event - *optional***, leave **AWS events** selected, and make sure that **DynamoDB Stream Sample event 1** is selected.

      This is the sample event which you'll use to test our event pattern.
   b. Under **Event pattern**, enter the following event pattern:

```
{
  "eventName": ["INSERT", "MODIFY"]
}
```

c. Choose **Test pattern**.

EventBridge displays a message that the sample event matches the event pattern. This is because the sample event has an `eventName` value of `INSERT`.

d. Choose **Next**.

7. Choose **Next** to skip specifying an enrichment.

In this example, you won't select an enrichment. Enrichments enable you to select a service to enhance the data from the source before sending it to the target. For more details, see ???.

8. Specify your Amazon SQS queue as the pipe target:

a. Under **Details**, for **Target service**, select **Amazon SQS queue**.

b. For **Queue**, select `pipe-tutorial-target`.

c. Leave the **Target Input transformer** section empty.

For more information, see ???.

9. Choose **Create Pipe**

EventBridge creates the pipe and displays the pipe detail page. The pipe is ready once its status updates to `Running`.

Step 2: Confirm the pipe filters events

Pipe is set up, but has yet to receive events from table.

To test the pipe, you'll update entries in the DynamoDB table. Each update will generate events that the DynamoDB stream sends to our pipe. Some will match the event pattern you specified, some will not. You can then examine the Amazon SQS queue to ensure that the pipe only delivered those event that matched our event pattern.

Update table items to generate events

1. Open the DynamoDB console at https://eusc-de-east-1.console.amazonaws-eusc.eu/dynamodb/.

2. From the left navigation, select **Tables**. Select the `pipe-tutorial-source` table.

   DynamoDB displays the table details page for `pipe-tutorial-source`.
3. Select **Explore table items**, and then choose **Create item**.

   DynamoDB displays the **Create item** page.
4. Under **Attributes**, create a new table item:

   a. For **Album** enter `Album A`.

   b. For **Artist** enter `Artist A`.

   c. Choose **Create item**.
5. Update the table item:

   a. Under **Items returned**, choose **Album A**.

   b. Select **Add new attribute**, then select **String**.

   c. Enter a new value of Song, with a value of Song `A`.

   d. Choose **Save changes**.
6. Delete the table item:

   a. Under **Items returned**, check **Album A**.

   b. From the **Actions** menu, select **Delete items**.

You have made three updates to the table item; this generates three events for the DynamoDB data stream:

- An `INSERT` event when you created the item.
- A `MODIFY` event when you added an attribute to the item.
- A `REMOVE` event when you deleted the item.

However, the event pattern you specified for the pipe should filter out any events that are not `INSERT` or `MODIFY` events. Next, confirm that the pipe delivered the expected events to the queue.

Confirm the expected events were delivered to the queue

1. Open the Amazon SQS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sqs/.
2. Choose the `pipe-tutorial-target` queue.

Amazon SQS displays the queue details page.

3. Select **Send and receive messages**, then under **Receive messages** choose **Poll for messages**.

   The queue polls the pipe and then lists the events it receives.

4. Choose the event name to see the event JSON that was delivered.

There should be two events in the queue: one with an `eventName` of `INSERT`, and one with an `eventName` of `MODIFY`. However, the pipe did not deliver the event for deleting the table item, since that event had an `eventName` of `REMOVE`, which did not match the event pattern you specified in the pipe.

Step 3: Clean up your resources

First, delete the pipe itself.

Delete the pipe using the EventBridge console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. On the navigation pane, choose **Pipes**.
3. Select the `pipe-tutorial` pipe, and choose **Delete**.

Then, delete the CloudFormation stack, to prevent being billed for the continued usage of the resources provisioned within it.

Delete the tutorial prerequisites using the AWS CLI

- Run the following CLI command, where `--stack-name` specifies the name of your stack:

```
aws cloudformation delete-stack --stack-name pipe-tuturial-resources
```

Delete the tutorial prerequisites using the CloudFormation console

1. Open the CloudFormation console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudformation.
2. On the **Stacks** page, select the stack and then select **Delete**.

3. Select **Delete** to confirm your action.

CloudFormation template for generating prerequisites

Use the JSON below to create a CloudFormation template for provisioning the source and target resources necessary for this tutorial.

```
{
  "AWSTemplateFormatVersion": "2010-09-09",

  "Description" : "Provisions resources to use with the EventBridge Pipes tutorial. You
 will be billed for the AWS resources used if you create a stack from this template.",

  "Parameters" : {
    "SourceTableName" : {
      "Type" : "String",
      "Default" : "pipe-tutorial-source",
      "Description" : "Specify the name of the table to provision as the pipe source,
 or accept the default."
    },
    "TargetQueueName" : {
      "Type" : "String",
      "Default" : "pipe-tutorial-target",
      "Description" : "Specify the name of the queue to provision as the pipe target,
 or accept the default."
    }
  },
  "Resources": {
    "PipeTutorialSourceDynamoDBTable": {
      "Type": "AWS::DynamoDB::Table",
      "Properties": {
        "AttributeDefinitions": [{
            "AttributeName": "Album",
            "AttributeType": "S"
          },
          {
            "AttributeName": "Artist",
            "AttributeType": "S"
          }

        ],
        "KeySchema": [{
```

```
            "AttributeName": "Album",
            "KeyType": "HASH"

          },
          {
            "AttributeName": "Artist",
            "KeyType": "RANGE"
          }
        ],
        "ProvisionedThroughput": {
          "ReadCapacityUnits": 10,
          "WriteCapacityUnits": 10
        },
        "StreamSpecification": {
          "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "TableName": { "Ref" : "SourceTableName" }
      }
    },
    "PipeTutorialTargetQueue": {
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": { "Ref" : "TargetQueueName" }
      }
    }
  }
}
```

Generating an CloudFormation template from EventBridge Pipes

AWS CloudFormation enables you to configure and manage your AWS resources across accounts and regions in a centralized and repeatable manner by treating infrastructure as code. CloudFormation does this by letting you create *templates*, which define the resources you want to provision and manage.

EventBridge enables you to generate templates from the existing pipes in your account, as an aid to help you jumpstart developing CloudFormation templates. You can select a single pipe, or multiple pipes to include in the template. You can then use these templates as the basis for creating stacks of resources under CloudFormation management.

For more information on CloudFormation, see *The CloudFormation User Guide.*

For event buses, you can generate CloudFormation templates from event buses and event bus rules.

Resources included in EventBridge Pipe templates

When EventBridge generates the CloudFormation template, it creates an AWS::Pipes::Pipe resource for each selected pipe. In addition, EventBridge includes the following resources under the described conditions:

- AWS::Events::ApiDestination

  If your pipes include API destinations, either as enrichments or targets, EventBridge includes them in the CloudFormation template as AWS::Events::ApiDestination resources.

- AWS::Events::EventBus

  If your pipes includes an event bus as a target, EventBridge includes it in the CloudFormation template as an AWS::Events::EventBus resource.

- AWS::IAM::Role

  If you had EventBridge create a new execution role when you configured the pipe, you can choose to have EventBridge include that role in the template as an AWS::IAM::Role resource. EventBridge does not include roles you create. (In either case, the `RoleArn` property of the AWS::Pipes::Pipe resource contains the ARN of the role.)

Considerations when using CloudFormation templates generated from EventBridge Pipes

Consider the following factors when using a CloudFormation template you generated from EventBridge:

- EventBridge does not include any passwords in the generate template.

  You can edit the template to include template parameters that enable users to specify passwords or other sensitive information when using the template to create or update a CloudFormation stack.

In addition, users can use Secrets Manager to create a secret in the desired region and then edit the generated template to employ dynamic parameters.

- Targets in the generated template remain exactly as they were specified in the original pipe. This can lead to cross-region issues if you do not appropriately edit the template before using it to create stacks in other regions.

  Additionally, the generated template does not create the downstream targets automatically.

Generating a CloudFormation template from EventBridge Pipes

To generate a CloudFormation template from one or more pipes using the EventBridge console, do the following:

To generate an CloudFormation template from one or more pipes

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Pipes**.
3. Under **Pipes**, choose one or more pipes you want to include in the generated CloudFormation template.

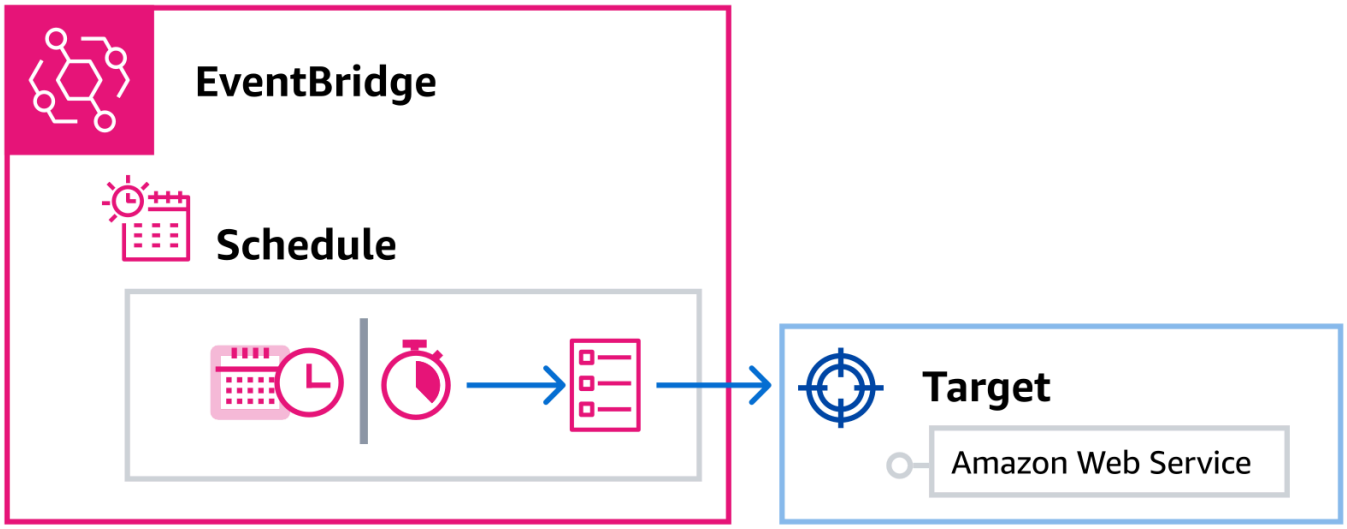
   For a single pipe, you can also choose the pipe name to display the pipe's details page.
4. Choose **CloudFormation Template**, and then choose which format you want EventBridge to generate the template in: **JSON** or **YAML**.

   EventBridge displays the template, generated in the selected format.
5. If you had EventBridge create a new execution role for any of the selected pipes, and you want EventBridge to include those roles in the template, choose **Include IAM roles created by console on your behalf**.
6. EventBridge gives you the option of downloading the template file, or copying the template to the clipboard.

   - To download the template file, choose **Download**.
   - To copy the template to the clipboard, choose **Copy**.
7. To exit the template, choose **Cancel**.

Amazon EventBridge Scheduler

Amazon EventBridge Scheduler is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.



EventBridge Scheduler is highly customizable, and offers improved scalability over EventBridge scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use EventBridge Scheduler to invoke targets on a schedule.

Set up the execution role

When you create a new schedule, EventBridge Scheduler must have permission to invoke its target API operation on your behalf. You grant these permissions to EventBridge Scheduler using an *execution role*. The permission policy you attach to your schedule's execution role defines the required permissions. These permissions depend on the target API you want EventBridge Scheduler to invoke.

When you use the EventBridge Scheduler console to create a schedule, as in the following procedure, EventBridge Scheduler automatically sets up an execution role based on your selected target. If you want to create a schedule using one of the EventBridge Scheduler SDKs, the AWS

CLI, or CloudFormation, you must have an existing execution role that grants the permissions EventBridge Scheduler requires to invoke a target. For more information about manually setting up an execution role for your schedule, see Setting up an execution role in the *EventBridge Scheduler User Guide*.

Related resources

For more information about EventBridge Scheduler, see the following:

- EventBridge Scheduler User Guide
- EventBridge Scheduler API Reference
- EventBridge Scheduler Pricing

Create a schedule

To create a schedule by using the console

1. Open the Amazon EventBridge Scheduler console at https://eusc-de-east-1.console.amazonaws-eusc.eu/scheduler/home.
2. On the **Schedules** page, choose **Create schedule**.
3. On the **Specify schedule detail** page, in the **Schedule name and description** section, do the following:

   a. For **Schedule name**, enter a name for your schedule. For example, **`MyTestSchedule`**.

   b. (Optional) For **Description**, enter a description for your schedule. For example, **`My first schedule`**.

   c. For **Schedule group**, choose a schedule group from the dropdown list. If you don't have a group, choose **default**. To create a schedule group, choose **create your own schedule**.

   You use schedule groups to add tags to groups of schedules.

4. • Choose your schedule options.

Occurrence	Do this...	
One-time schedule	For **Date and time**, do the following:	

Occurrence	Do this...	
A one-time schedule invokes a target only once at the date and time that you specify.	• Enter a valid date in `YYYY/MM/DD` format. • Enter a timestamp in 24-hour `hh:mm` format. • For **Timezone**, choose the timezone.	

Occurrence	Do this...	
Recurring schedule A recurring schedule invokes a target at a rate that you specify using a **cron** expression or rate expression.	a. For **Schedule type**, do one of the following: • To use a cron expression to define the schedule, choose **Cron-based schedule** and enter the cron expression. • To use a rate expression to define the schedule, choose **Rate-based schedule** and enter the rate expression. For more informati on about cron and rate expressions, see Schedule types on EventBridge Scheduler in the *Amazon EventBridge Scheduler User Guide*. b. For **Flexible time window**, choose **Off** to turn off the option, or choose one of the pre-defined time windows. For example, if you choose **15 minutes** and you set a recurring schedule to invoke its target once every hour, the schedule runs within	

Occurrence	Do this...	
	15 minutes after the start of every hour.	

5. (Optional) If you chose **Recurring schedule** in the previous step, in the **Timeframe** section, do the following:

   a. For **Timezone**, choose a timezone.

   b. For **Start date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.

   c. For **End date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.

6. Choose **Next**.

7. On the **Select target** page, choose the AWS API operation that EventBridge Scheduler invokes:

   a. For **Target API**, choose **Templated targets**.

   b. Choose **Amazon EventBridge PutEvents**.

   c. Under **PutEvents**, specify the following:

   - For **EventBridge event bus**, choose the event bus from the drop-down menu. For example, **`default`**.

     You can also create a new event bus in the EventBridge console by choosing **Create new event bus**.

   - For **Detail-type**, enter the detail type of the events you want to match. For example, **`Object Created`**.

   - For **Source**, enter the name of the service that is the source of the events.

     For AWS service events, specify the service prefix as the source. Do not include the `aws.` prefix. For example, for Amazon S3 events, enter **`s3`**.

     To determine a service's prefix, see The condition keys table in the *Service Authorization Reference*. For more information about source and detail-type event values, see AWS service event metadata in the *Events Reference*.>.

   - (Optional): For **Detail**, enter an event pattern to further filter the events EventBridge Scheduler sends to EventBridge.

For more information, see ???.

8. Choose **Next**.
9. On the **Settings** page, do the following:

   a. To turn on the schedule, under **Schedule state**, toggle **Enable schedule**.

   b. To configure a retry policy for your schedule, under **Retry policy and dead-letter queue (DLQ)**, do the following:

   - Toggle **Retry**.
   - For **Maximum age of event**, enter the maximum **hour(s)** and **min(s)** that EventBridge Scheduler must keep an unprocessed event.
   - The maximum time is 24 hours.
   - For **Maximum retries**, enter the maximum number of times EventBridge Scheduler retries the schedule if the target returns an error.

     The maximum value is 185 retries.

   With retry policies, if a schedule fails to invoke its target, EventBridge Scheduler re-runs the schedule. If configured, you must set the maximum retention time and retries for the schedule.

   c. Choose where EventBridge Scheduler stores undelivered events.

Dead-letter queue (DLQ) option	Do this...
Don't store	Choose **None**.
Store the event in the same AWS account where you're creating the schedule	a. Choose **Select an Amazon SQS queue in my AWS account as a DLQ**. b. Choose the Amazon Resource Name (ARN) of the Amazon SQS queue.

Dead-letter queue (DLQ) option	Do this...
Store the event in a different AWS account from where you're creating the schedule	a. Choose **Specify an Amazon SQS queue in other AWS accounts as a DLQ**. b. Enter the Amazon Resource Name (ARN) of the Amazon SQS queue.

d. To use a customer managed key to encrypt your target input, under **Encryption**, choose **Customize encryption settings (advanced)**.

   If you choose this option, enter an existing KMS key ARN or choose **Create an AWS KMS key** to navigate to the AWS KMS console. For more information about how EventBridge Scheduler encrypts your data at rest, see Encryption at rest in the *Amazon EventBridge Scheduler User Guide*.

e. To have EventBridge Scheduler create a new execution role for you, choose **Create new role for this schedule**. Then, enter a name for **Role name**. If you choose this option, EventBridge Scheduler attaches the required permissions necessary for your templated target to the role.

10. Choose **Next**.
11. In the **Review and create schedule** page, review the details of your schedule. In each section, choose **Edit** to go back to that step and edit its details.
12. Choose **Create schedule**.

    You can view a list of your new and existing schedules on the **Schedules** page. Under the **Status** column, verify that your new schedule is **Enabled**.

Creating a scheduled rule (legacy) in Amazon EventBridge

Note

Scheduled rules are a legacy feature of EventBridge.

EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.
Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
For more information, see ??? or the *EventBridge Scheduler User Guide*.

In EventBridge, you can create two types of scheduled rules:

- Rules that run at a regular rate

  EventBridge runs these rules at regular intervals; for example, every 20 minutes.

  To specify the rate for a scheduled rule, you define a *rate expression*.
- Rules that run at specific times

  EventBridge runs these rules at specific times and dates; for example, 8:00 a.m. PST on the first Monday of every month.

  To specify the time and dates a scheduled rule runs, you define a *cron expression*.

Rate expressions are simpler to define, while cron expressions offer detailed schedule control. For example, with a cron expression, you can define a rule that runs at a specified time on a certain day of each week or month. In contrast, rate expressions run a rule at a regular rate, such as once every hour or once every day.

All scheduled events use UTC+0 time zone, and the minimum precision for a schedule is one minute.

Note

EventBridge doesn't provide second-level precision in schedule expressions. The finest resolution using a cron expression is one minute. Due to the distributed nature of EventBridge and the target services, there can be a delay of several seconds between

the time the scheduled rule is triggered and the time the target service runs the target resource.

Create a scheduled rule (legacy)

The following steps walk you through how to create an EventBridge rule that runs on a regular schedule.

> **Note**
>
> You can only create scheduled rules using the default event bus.

Steps

- Define the rule
- Define the schedule
- Select targets
- Configure tags and review rule

Define the rule

First, enter a name and description for your rule to identify it.

To define the rule detail

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, under **Scheduler**, choose **Scheduled rule (legacy)**.
3. Choose **Create scheduled rule**.
4. Enter a **Name** and, optionally, a **Description** for the rule.

   A rule can't have the same name as another rule in the same AWS Region and on the same event bus.
5. To have the rule take effect as soon as you create it, make sure the **Enable the scheduled rule** option is enabled.

Define the schedule

Next, define the schedule pattern.

To define the schedule pattern

- For **Schedule pattern**, choose whether you want the schedule to run at a specific time, or at a regular rate:

  Specific time

  1. Choose **A fine-grained schedule that runs at a specific time, such as 8:00 a.m. PST on the first Monday of every month.**
  2. For **Cron expression**, specify fields to define the cron expresssion that EventBridge should use to determine when to execute this scheduled rule.

     Once you have specified all fields, EventBridge displays the next ten dates when EventBridge will execute this scheduled rule. You can choose whether to display those dates in **UTC** or **Local time zone**.

     For more information on constructing a cron expression, see ???.

  Regular rate

  1. Choose **A schedule that runs at a regular rate, such as every 10 minutes.**
  2. For **Rate expression**, specify the **Value** and **Unit** fields to define the rate at which EventBridge should execute this scheduled rule.

     For more information on constructing a rate expression, see ???.

Select targets

Choose one or more targets to receive events that match the specified pattern. Targets can include an EventBridge event bus, EventBridge API destinations, including SaaS partners such as Salesforce, or another AWS service.

To select targets

1. For **Target type**, choose one of the following target types:

Event bus

To select an EventBridge event bus, select **EventBridge event bus**, then do the following:

- To use an event bus in the same AWS Region as this rule:
  1. Select **Event bus in the same account and Region**.
  2. For **Event bus for target**, choose the dropdown box and enter the name of the event bus. You can also select the event bus from the dropdown list.

     For more information, see ???.
- To use an event bus in a different AWS Region or account as this rule:
  1. Select **Event bus in a different account or Region**.
  2. For **Event bus as target**, enter the ARN of the event bus you want to use.

     For more information, see:

     - ???
     - ???

API destination

To use an EventBridge API destination, select **EventBridge API destination**, then do one of the following:

- To use an existing API destination, select **Use an existing API destination**. Then select an API destination from the dropdown list.
- To create a new API destination, select **Create a new API destination**. Then, provide the following details for the destination:
  - **Name** – Enter a name for the destination.

    Names must be unique within your AWS account. Names can have up to 64 characters. Valid characters are **A-Z**, **a-z**, **0-9**, and **. _ -** (hyphen).
  - (Optional) **Description** – Enter a description for the destination.

    Descriptions can have up to 512 characters.
  - **API destination endpoint** – The URL endpoint for the target.

The endpoint URL must start with **`https`**. You can include the **`*`** as a path parameter wildcard. You can set path parameters from the target's `HttpParameters` attribute.

- **HTTP method** – Select the HTTP method used when you invoke the endpoint.
- (Optional) **Invocation rate limit per second** – Enter the maximum number of invocations accepted for each second for this destination.

  This value must be greater than zero. By default, this value is set to 300.
- **Connection** – Choose to use a new or existing connection:
  - To use an existing connection, select **Use an existing connection** and select the connection from the dropdown list.
  - To create a new connection for this destination select **Create a new connection**, then define the connection's **Name**, **Destination type**, and **Authorization type**. You can also add an optional **Description** for this connection.

For more information, see ???.

AWS service

To use an AWS service, select **AWS service**, then do the following:

1. For **Select a target**, select an AWS service to use as the target. Provide the information requested for the service you select.

   > **Note**
   >
   > The fields displayed vary depending on the service selected. For more information about available targets, see Event bus targets available in the EventBridge console.

2. For many target types, EventBridge needs permissions to send events to the target. In these cases, EventBridge can create the IAM role needed for your rule to run.

   For **Execution role**, do one of the following:

   - To create a new execution role for this rule:

     a. Select **Create a new role for this specific resource**.

b. Either enter a name for this execution role, or use the name generated by EventBridge.

- To use an existing execution role for this rule:

a. Select **Use existing role**.

b. Enter or select the name of the execution role to use from the dropdown list.

3. (Optional) For **Additional settings**, specify any of the optional settings available for your target type:

Event bus

(Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:

- Choose **None** to not use a dead-letter queue.
- Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.
- Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

  For more information, see Granting permissions to the dead-letter queue.

API destination

1. (Optional) For Configure target input, choose how you want to customize the text sent to the target for matching events. Choose one of the following:
   - **Matched events** – EventBridge sends the entire original source event to the target. This is the default.
   - **Part of the matched events** – EventBridge only sends the specified portion of the original source event to the target.

     Under **Specify the part of the matched event**, specify a JSON path that defines the part of the event you want EventBridge to send to the target.
   - **Constant (JSON text)** – EventBridge sends only the specified JSON text to the target. No part of the original source event is sent.

Under **Specify the constant in JSON**, specify the JSON text that you want EventBridge to send to the target instead of the event.

- **Input transformer** – Configure an input transformer to customize the text you want EventBridge send to the target. For more information, see ???.

  a. Select **Configure input transformer**.

  b. Configure the input transformer following the steps in ???.

2. (Optional) Under **Retry policy**, specify how EventBridge should retry sending an event to a target after an error occurs.
   - **Maximum age of event** – Enter the maximum amount of time (in hours, minutes, and seconds) for EventBridge to retain unprocessed events. The default is 24 hours.
   - **Retry attempts** – Enter the maximum number of times EventBridge should retry sending an event to the target after an error occurs. The default is 185 times.
3. (Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:
   - Choose **None** to not use a dead-letter queue.
   - Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.
   - Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

     For more information, see Granting permissions to the dead-letter queue.

AWS service

Note that EventBridge may not display all of the following fields for a given AWS service.

1. (Optional) For Configure target input, choose how you want to customize the text sent to the target for matching events. Choose one of the following:
   - **Matched events** – EventBridge sends the entire original source event to the target. This is the default.
   - **Part of the matched events** – EventBridge only sends the specified portion of the original source event to the target.

Under **Specify the part of the matched event**, specify a JSON path that defines the part of the event you want EventBridge to send to the target.

- **Constant (JSON text)** – EventBridge sends only the specified JSON text to the target. No part of the original source event is sent.

  Under **Specify the constant in JSON**, specify the JSON text that you want EventBridge to send to the target instead of the event.

- **Input transformer** – Configure an input transformer to customize the text you want EventBridge send to the target. For more information, see ???.

  a. Select **Configure input transformer**.

  b. Configure the input transformer following the steps in ???.

2. (Optional) Under **Retry policy**, specify how EventBridge should retry sending an event to a target after an error occurs.
   - **Maximum age of event** – Enter the maximum amount of time (in hours, minutes, and seconds) for EventBridge to retain unprocessed events. The default is 24 hours.
   - **Retry attempts** – Enter the maximum number of times EventBridge should retry sending an event to the target after an error occurs. The default is 185 times.
3. (Optional) For **Dead-letter queue**, choose whether to use a standard Amazon SQS queue as a dead-letter queue. EventBridge sends events that match this rule to the dead-letter queue if they are not successfully delivered to the target. Do one of the following:
   - Choose **None** to not use a dead-letter queue.
   - Choose **Select an Amazon SQS queue in the current AWS account to use as the dead-letter queue** and then select the queue to use from the drop-down list.
   - Choose **Select an Amazon SQS queue in an other AWS account as a dead-letter queue** and then enter the ARN of the queue to use. You must attach a resource-based policy to the queue that grants EventBridge permission to send messages to it.

     For more information, see Granting permissions to the dead-letter queue.
4. (Optional) Choose **Add another target** to add another target for this rule.
5. Choose **Next**.

Configure tags and review rule

Finally, enter any desired tags for the rule, then review and create the rule.

To configure tags, and review and create the rule

1. (Optional) Enter one or more tags for the rule. For more information, see Tagging resources in Amazon EventBridge.
2. Choose **Next**.
3. Review the details for the new rule. To make changes to any section, choose the **Edit** button next to that section.

   When satisfied with the rule details, choose **Create rule**.

Setting a schedule pattern for scheduled rules (legacy) in Amazon EventBridge

> **Note**
>
> Scheduled rules are a legacy feature of EventBridge.
> EventBridge offers a more flexible and powerful way to create, run, and manage scheduled tasks centrally, at scale: EventBridge Scheduler. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for failed API invocations.
> Scheduler is highly customizable, and offers improved scalability over scheduled rules, with a wider set of target API operations and AWS services. We recommend that you use Scheduler to invoke targets on a schedule.
> For more information, see ??? or the *EventBridge Scheduler User Guide*.

When you create a scheduled rule in EventBridge you can specify a schedule pattern that determines when EventBridge runs the rule:

- Use a *cron* expression to run the rule at specific times and dates.
- Use a *rate* expression to run the rule at regular intervals.

Cron expressions

Cron expressions have six required fields, which are separated by white space.

Syntax

```
cron(fields)
```

Field	Values	Wildcards
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W
Month	1-12 or JAN-DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? L #
Year	1970-2199	, - * /

Wildcards

- The **,** (comma) wildcard includes additional values. In the Month field, JAN,FEB,MAR includes January, February, and March.
- The **-** (dash) wildcard specifies ranges. In the Day field, 1-15 includes days 1 through 15 of the specified month.
- The ***** (asterisk) wildcard includes all values in the field. In the Hours field, ***** includes every hour. You can't use ***** in both the Day-of-month and Day-of-week fields. If you use it in one, you must use **?** in the other.
- The **/** (slash) wildcard specifies increments. In the Minutes field, you could enter 1/10 to specify every tenth minute, starting from the first minute of the hour (for example, the 11th, 21st, and 31st minute, and so on).
- The **?** (question mark) wildcard specifies any. In the Day-of-month field you could enter **7** and if any day of the week was acceptable, you could enter **?** in the Day-of-week field.
- The **L** wildcard in the Day-of-month or Day-of-week fields specifies the last day of the month or week.
- The **W** wildcard in the Day-of-month field specifies a weekday. In the Day-of-month field, **3W** specifies the weekday closest to the third day of the month.

- The **#** wildcard in the Day-of-week field specifies a certain instance of the specified day of the week within a month. For example, 3#2 would be the second Tuesday of the month: the 3 refers to Tuesday because it is the third day of each week, and the 2 refers to the second day of that type within the month.

> **Note**
>
> If you use a '#' character, you can define only one expression in the day-of-week field. For example, `"3#1,6#3"` is not valid because it is interpreted as two expressions.

Limitations

- You can't specify the Day-of-month and Day-of-week fields in the same cron expression. If you specify a value or a * (asterisk) in one of the fields, you must use a **?** (question mark) in the other.
- Cron expressions that lead to rates faster than 1 minute are not supported.

Examples

You can use the following sample cron strings when creating a rule with schedule.

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0	10	*	*	?	*	Run at 10:00 am (UTC+0) every day
15	12	*	*	?	*	Run at 12:15 pm (UTC+0) every day
0	18	?	*	MON-FRI	*	Run at 6:00 pm (UTC+0) every

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
						Monday through Friday
0	8	1	*	?	*	Run at 8:00 am (UTC+0) every 1st day of the month
0/15	*	*	*	?	*	Run every 15 minutes
0/10	*	?	*	MON-FRI	*	Run every 10 minutes Monday through Friday
0/5	8-17	?	*	MON-FRI	*	Run every 5 minutes Monday through Friday between 8:00 am and 5:55 pm (UTC +0)

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0/30	20-2	?	*	MON-FRI	*	Run every 30 minutes Monday through Friday between 8:00 pm on the starting day to 2:00 am on the following day (UTC) Run from 12:00 am to 2:00 am on Monday morning (UTC).

The following example creates a rule that runs every day at 12:00pm UTC+0.

```
aws events put-rule --schedule-expression "cron(0 12 * * ? *)" --name MyRule1
```

The following example creates a rule that runs every day, at 2:05pm and 2:35pm UTC+0.

```
aws events put-rule --schedule-expression "cron(5,35 14 * * ? *)" --name MyRule2
```

The following example creates a rule that runs at 10:15am UTC+0 on the last Friday of each month during the years 2019 to 2022.

```
aws events put-rule --schedule-expression "cron(15 10 ? * 6L 2019-2022)" --name MyRule3
```

Rate expressions

A *rate expression* starts when you create the scheduled event rule, and then it runs on a defined schedule.

Rate expressions have two required fields separated by white space.

Syntax

```
rate(value unit)
```

value

A positive number.

unit

The unit of time. Different units are required for values of 1, such as `minute`, and values over 1, such as `minutes`.

Valid values: minute | minutes | hour | hours | day | days

Limitations

If the value is equal to 1, then the unit must be singular. If the value is greater than 1, the unit must be plural. For example, rate(1 hours) and rate(5 hour) aren't valid, but rate(1 hour) and rate(5 hours) are valid.

Examples

The following examples show how to use rate expressions with the AWS CLI `put-rule` command. The first example triggers the rule every minute, the next triggers it every five minutes, the third example triggers it once an hour, and the final example triggers it once per day.

```
aws events put-rule --schedule-expression "rate(1 minute)" --name MyRule2
```

```
aws events put-rule --schedule-expression "rate(5 minutes)" --name MyRule3
```

```
aws events put-rule --schedule-expression "rate(1 hour)" --name MyRule4
```

```
aws events put-rule --schedule-expression "rate(1 day)" --name MyRule5
```

Amazon EventBridge schemas

A schema defines the structure of events that are sent to EventBridge. EventBridge provides schemas for all events that are generated by AWS services. You can also create or upload custom schemas or infer schemas directly from events on an event bus. Once you have a schema for an event, you can download code bindings for popular programming languages and speed up development. You can work with code bindings for schemas and manage schemas from the EventBridge console, by using the API, or directly in your IDE by using the AWS toolkits. To build serverless apps that use events, use AWS Serverless Application Model.

> **Note**
>
> When using the input transformer feature, the original event is inferred by schema discovery, not the transformed event that's sent to the target.

EventBridge supports both OpenAPI 3 and JSONSchema Draft4 formats.

For AWS Toolkit for JetBrains and AWS Toolkit for VS Code, you can browse or search for schemas and download code bindings for schemas directly in your IDE.

Amazon EventBridge: Using the Schema Registry

Schema registry API property value masking

Some property values of events that are used to create a schema registry may contain sensitive customer information. To protect the customer's information, the values will be masked with asterisks (*). Because we're masking these values, EventBridge recommends not building applications that explicitly depend on the following properties or their values:

- **CreateSchema** – The `Content` property of the `requestParameters` body
- **GetDiscoveredSchema** – The `Events` property of the `requestParameters` body and the `Content` property of the `responseElements` body
- **SearchSchemas** – The `keywords` property of the `requestParameters`
- **UpdateSchema** – The `Content` property of the `requestParameters`

Finding an AWS service event schema in Amazon EventBridge

EventBridge includes schemas for all AWS services that generate events. You can find these schemas in the EventBridge console, or you can find them by using the API action `SearchSchemas`.

To find schemas for AWS services in the EventBridge console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas**.
3. On the **Schemas** page, select **AWS event schema registry**.
   <result>

   The first page of available schemas is displayed.
   </result>
4. To find a schema, in **Search AWS event schemas**, enter a search term**.**

   A search returns matches for both the name and contents of the available schemas, and then displays which versions of the schema contain matches.
5. Open an event schema by selecting the name of the schema.

Schema registries in Amazon EventBridge

Schema registries are containers for schemas. Schema registries collect and organize schemas so that your schemas are in logical groups. The default schema registries are:

- **All schemas** – All the schemas from the AWS event, discovered, and custom schema registries.
- **AWS event schema registry** – The built-in schemas.
- **Discovered schema registry** – The schemas discovered by Schema discovery.

You can create custom registries to organize the schemas you create or upload.

> **Note**
>
> Exporting custom schemas from the EventBridge Schema Registry is not supported.

To create a custom registry

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas** and then choose **Create registry**.
3. On the **Registry details** page, enter a **Name**.
4. (Optional) Enter a description for your new registry.
5. Choose **Create**.

To create a custom schema in your new registry, select **Create custom schema**. To add a schema to your registry, select that registry when you're creating a new schema.

To create a registry by using the API, use `CreateRegistry`. For more information, see Amazon EventBridge Schema Registry API Reference.

For information about using the EventBridge schema registry through AWS CloudFormation, see EventSchemas Resource Type Reference in CloudFormation.

Creating an event schema in Amazon EventBridge

You create schemas by using JSON files with either the OpenAPI Specification or the JSONSchema Draft4 specification. You can create or upload your own schemas in EventBridge by using a template or generating a schema based on the JSON of an event. You can also infer the schema from events on an event bus. To create a schema by using the EventBridge Schema Registry API, use the `CreateSchema` API action.

When you choose between OpenAPI 3 and JSONSchema Draft4 formats, consider the following differences:

- JSONSchema format supports additional keywords that aren't supported in OpenAPI, such as `$schema, additionalItems`.
- There are minor differences in how keywords are handled, such as `type` and `format`.
- OpenAPI doesn't support JSONSchema Hyper-Schema hyperlinks in JSON documents.
- Tools for OpenAPI tend to focus on build-time, whereas tools for JSONSchema tend to focus on run-time operations, such as client tools for schema validation.

We recommend using JSONSchema format to implement client-side validation so that events sent to EventBridge conform to the schema. You can use JSONSchema to define a contract for valid JSON documents, and then use a JSON schema validator before sending the associated events.

After you have a new schema, you can download code bindings to help create applications for events with that schema.

Creating a schema by using a template in Amazon EventBridge

You can create a schema from a template file you download, or by editing a template directly in the EventBridge console.

Create a schema from a template file

To get the template, you download it from the console. You can edit the template so that the schema matches your events. Then upload your new template through the console.

To download the schema template

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.

2. In the navigation pane, choose **Schema registry**.
3. In the **Getting started** section under **Schema template**, choose **Download**.

Alternatively, you can copy the JSON template from the following code example.

```
{
    "openapi": "3.0.0",
    "info": {
      "version": "1.0.0",
      "title": "Event"
    },
    "paths": {},
    "components": {
      "schemas": {
        "Event": {
          "type": "object",
          "properties": {
            "ordinal": {
              "type": "number",
              "format": "int64"
            },
            "name": {
              "type": "string"
            },
            "price": {
              "type": "number",
              "format": "double"
            },
            "address": {
              "type": "string"
            },
            "comments": {
              "type": "array",
              "items": {
                "type": "string"
              }
            },
            "created_at": {
              "type": "string",
              "format": "date-time"
            }
          }
```

```
        }
      }
    }
  }
```

To upload a schema template

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas** and then choose **Create schema**.
3. (Optional) Select or create a schema registry.
4. Under **Schema details**, enter a name for your schema.
5. (Optional) Enter a description for your schema.
6. For **Schema type**, choose either **OpenAPI 3.0** or **JSON Schema Draft 4**.
7. On the **Create** tab, in the text box, either drag your schema file to the text box, or paste the schema source.
8. Select **Create**.

Edit a schema template directly in the console

You can create a schema directly in the EventBridge console.

To edit a schema in the console

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas** and then choose **Create schema**.
3. (Optional) Select or create a schema registry.
4. Under **Schema details**, enter a name for your schema.
5. For **Schema type**, choose either **OpenAPI 3.0** or **JSON Schema Draft 4**.
6. (Optional) Enter a description for the schema to create.
7. On the **Create** tab, choose **Load template**.
8. In the text box, edit the template so that the schema matches your events.
9. Select **Create**.

Creating a schema from event JSON in Amazon EventBridge

If you have the JSON of an event, you can automatically create a schema for that type of event.

To create a schema based on the JSON of an event

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas** and then choose **Create schema**.
3. (Optional) Select or create a schema registry.
4. Under **Schema details** enter a name for your schema.
5. (Optional) Enter a description for the schema you created.
6. For **Schema type**, choose **OpenAPI 3.0**.

   You can't use JSONSchema when you create a schema from the JSON of an event.
7. Select **Discover from JSON**
8. In the text box under **JSON**, paste or drag the JSON source of an event.

   For example, you could paste in the source from this AWS Step Functions event for a failed execution.

```
{
    "version": "0",
    "id": "315c1398-40ff-a850-213b-158f73e60175",
    "detail-type": "Step Functions Execution Status Change",
    "source": "aws.states",
    "account": "012345678912",
    "time": "2019-02-26T19:42:21Z",
    "region": "us-east-1",
    "resources": [
      "arn:aws:states:us-east-1:012345678912:execution:state-machine-
name:execution-name"
    ],
    "detail": {
        "executionArn": "arn:aws:states:us-east-1:012345678912:execution:state-
machine-name:execution-name",
        "stateMachineArn": "arn:aws:states:us-
east-1:012345678912:stateMachine:state-machine",
        "name": "execution-name",
        "status": "FAILED",
```

```
        "startDate": 1551225146847,
        "stopDate": 1551225151881,
        "input": "{}",
        "output": null
    }
}
```

9. Choose **Discover schema**.
10. EventBridge generates an OpenAPI schema for the event. For example, the following schema is generated for the preceding Step Functions event.

```
{
  "openapi": "3.0.0",
  "info": {
    "version": "1.0.0",
    "title": "StepFunctionsExecutionStatusChange"
  },
  "paths": {},
  "components": {
    "schemas": {
      "AWSEvent": {
        "type": "object",
        "required": ["detail-type", "resources", "detail", "id", "source", "time",
 "region", "version", "account"],
        "x-amazon-events-detail-type": "Step Functions Execution Status Change",
        "x-amazon-events-source": "aws.states",
        "properties": {
          "detail": {
            "$ref": "#/components/schemas/StepFunctionsExecutionStatusChange"
          },
          "account": {
            "type": "string"
          },
          "detail-type": {
            "type": "string"
          },
          "id": {
            "type": "string"
          },
          "region": {
            "type": "string"
          },
          "resources": {
```

```
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "source": {
            "type": "string"
          },
          "time": {
            "type": "string",
            "format": "date-time"
          },
          "version": {
            "type": "string"
          }
        }
      },
      "StepFunctionsExecutionStatusChange": {
        "type": "object",
        "required": ["output", "input", "executionArn", "name", "stateMachineArn",
"startDate", "stopDate", "status"],
        "properties": {
          "executionArn": {
            "type": "string"
          },
          "input": {
            "type": "string"
          },
          "name": {
            "type": "string"
          },
          "output": {},
          "startDate": {
            "type": "integer",
            "format": "int64"
          },
          "stateMachineArn": {
            "type": "string"
          },
          "status": {
            "type": "string"
          },
          "stopDate": {
            "type": "integer",
```

```
            "format": "int64"
          }
        }
      }
    }
  }
}
```

11. After the schema has been generated, choose **Create**.

Inferring schemas from event bus events in EventBridge

Amazon EventBridge can infer schemas by discovering events. To infer schemas, you turn on event discovery on an event bus and every unique schema is added to the schema registry, including those for cross-account events. Schemas discovered by EventBridge appear in **Discovered schemas registry** on the **Schemas** page.

If the contents of events on the event bus change, EventBridge creates new versions of the related EventBridge schema.

Considerations when starting schema discovery on an event bus

Take into account the following considerations before enabling schema discover on an event bus:

- Enabling event discovery on an event bus can incur a cost. The first five million processed events in each month are free.
- EventBridge infers schemas from cross-account events by default but you can disable it by updating the `cross-account` property. For more information, see Discoverers in the EventBridge Schema Registry API Reference.

> **Note**
>
> Schema discovery is not supported for event buses encrypted using a customer managed key. To enable schema discovery on an event bus, choose to use an AWS owned key. For more information, see KMS key options.

To start or stop schema discovery on an event bus (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Select the event bus on which you want to start or stop schema discovery.
4. Do one of the following:
   - To start schema discovery, choose **Start discovery**.
   - To stop schema discovery, choose **Delete discovery**.

To start or stop schema discovery on an event bus (AWS CLI)

- To start schema discovery, use create-discoverer.

  To stop schema discovery, use delete-discoverer.

> **Note**
>
> Schema discovery does not support events larger than 1000 KiB. These events will not be discovered and no error notification will be generated. To track schemas for larger events, you must manually create schemas using the schema registry.

Generating code bindings for event schemas in Amazon EventBridge

You can generate code bindings for event schemas to speed up development in Golang, Java, Python, and TypeScript. Code bindings are available for AWS service events, schemas you create, and for schemas you generate based on events on an event bus. You can generate code bindings for a schema by using the EventBridge console, the EventBridge Schema Registry API, or in your IDE with an AWS toolkit.

To generate code bindings from an EventBridge schema

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Schemas**.
3. Find a schema that you want code bindings for, either by browsing the schema registries, or by searching for a schema.
4. Select the schema name..
5. On the **Schema details** page, in the **Version** section, choose **Download code bindings**.
6. On the **Download code bindings** page, select the language of the code bindings you want to download.
7. Select **Download**.

   It may take a few seconds for your download to begin. The downloaded file is a zip file of code bindings for the language you selected.

Third-party integrations with Amazon EventBridge

In addition to AWS services, Amazon EventBridge supports integration with custom and third-party systems and applications, both as event sources and targets.

Third-party integration that EventBridge supports includes:

- Creating API destinations to send events to public HTTP endpoints from an event bus or pipe.
- Enabling your event bus to receive events from a Software as a Service (SaaS) partner source.

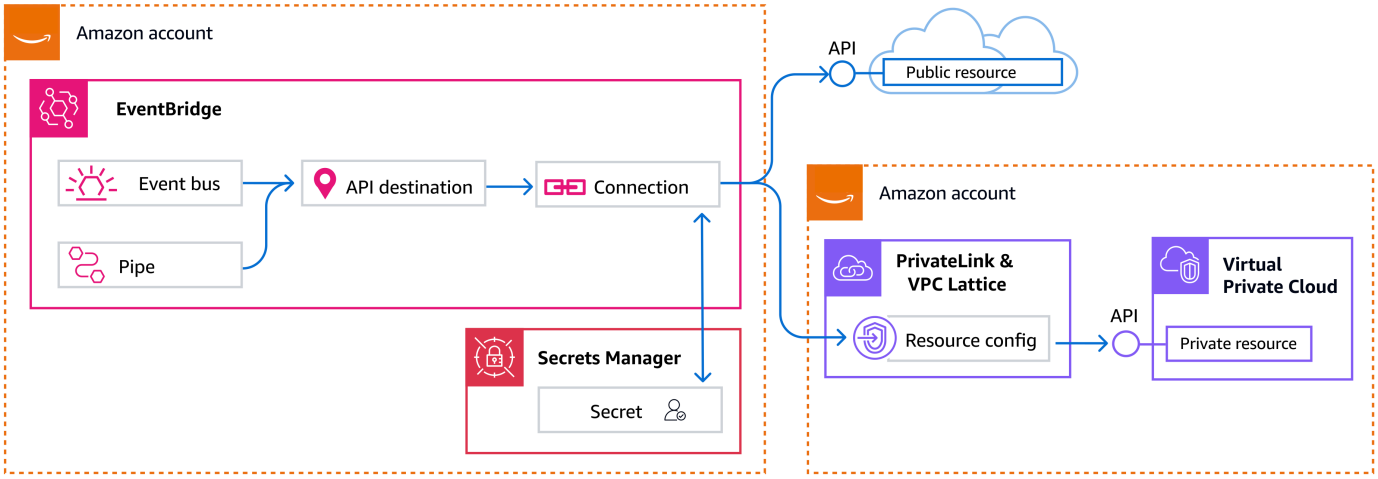
API destinations as targets in Amazon EventBridge

EventBridge *API destinations* are HTTPS endpoints that you can invoke as the target of an event bus rule, or pipe, similar to how you invoke an AWS service or resource as a target. Using API destinations, you can route events between AWS services, integrated software as a service (SaaS) applications, and public or private applications by using API calls.

> **Note**
>
> The examples on this page use rules. A Custom Event Bus subscriber uses the same API destinations: set the API destination ARN as the subscriber's `TargetArn` with `HttpParameters`. See the section called “HTTP and API destinations”.

When you specify an API destination as a rule or pipe target, EventBridge invokes the HTTPS endpoint for any event that matches the event pattern specified in the rule or pipe and then delivers the event information with the request. With EventBridge, you can use any HTTP method except CONNECT and TRACE for the request. The most common HTTP methods to use are PUT and POST.

You can also use input transformers to customize the event to the parameters of a specific HTTP endpoint parameters. For more information, see Amazon EventBridge input transformation.



EventBridge API destinations use *connections* to define the authorization method and credentials and network connectivity for EventBridge to use when connecting to a given HTTPS endpoint. Connections support both public and private APIs. For more information, see Connections.

Note

EventBridge API destinations currently only support public domain names with publicly trusted certificates for HTTPS endpoints when using private APIs. API destinations do not support mutual TLS (mTLS).

Important

EventBridge requests to an API destination endpoint must have a maximum client execution timeout of 5 seconds. If the target endpoint takes longer than 5 seconds to respond, EventBridge times out the request. EventBridge retries timed out requests up to the maximums that are configured on your retry policy.

- For event buses, by default the maximums are 24 hours and 185 times.
- For pipes, retries are determined the pipe source type and its configuration. EventBridge will retry until the event expires from the source, or the maximum event age or retry attempts configured has reached.

After the maximum number of retries, events are sent to your dead-letter queue if you have one.Otherwise, the event is dropped.

Amazon EventBridge: Using API destinations

Service-linked role for API destinations

When you create a connection for an API destination, the service-linked role AmazonEventBridgeApiDestinationsServiceRolePolicy is added to your account. EventBridge uses this service-linked role to create and store a secret in Secrets Manager. To grant the necessary permissions to the service-linked role, EventBridge attaches the **AmazonEventBridgeApiDestinationsServiceRolePolicy** policy to the role. The policy limits the permissions granted to only those necessary for the role to interact with the secret for the connection. No other permissions are included, and the role can interact only with the connections in your account to manage the secret.

For more information about service-linked roles, see Using service-linked roles in the *IAM User Guide*.

Headers in requests to API destinations

The following section details how EventBridge handles HTTP headers in requests to API destinations.

Headers included in requests to API destinations

In addition to the authorization headers defined for the connection used for an API destination, EventBridge includes the following headers in each request.

Header key	Header value
User-Agent	Amazon/EventBridge/ApiDestinations
Content-Type	If no custom Content-Type value is specified , EventBridge includes the following default value as Content-Type:

Header key	Header value
	application/json; charset=utf-8
Range	bytes=0-1048575
Accept-Encoding	gzip,deflate
Connection	close
Content-Length	An entity header that indicates the size of the entity-body, in bytes, sent to the recipient.
Host	A request header that specifies the host and port number of the server where the request is being sent.

Headers that cannot be overridden in requests to API destinations

EventBridge does not allow you to override the following headers:

- User-Agent
- Range

Headers EventBridge removes from requests to API destinations

EventBridge removes the following headers for all API destination requests:

- A-IM
- Accept-Charset
- Accept-Datetime
- Accept-Encoding
- Cache-Control
- Connection
- Content-Encoding
- Content-Length

- Content-MD5
- Date
- Expect
- Forwarded
- From
- Host
- HTTP2-Settings
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Origin
- Pragma
- Proxy-Authorization
- Range
- Referer
- TE
- Trailer
- Transfer-Encoding
- User-Agent
- Upgrade
- Via
- Warning

API destination error codes

When EventBridge tries to deliver an event to an API destination and an error occurs, EventBridge does the following:

- Retries events associated with error codes `401`, `407`, `409`, `429`, and `5xx`.
- Does not retry events associated with error codes `1xx`, `2xx`, `3xx`, and `4xx` (other than those noted above).

EventBridge API destinations read the standard HTTP response header `Retry-After` to find out how long to wait before making a follow-up request. For event buses, EventBridge chooses the more conservative value between the defined retry policy and the `Retry-After` header. If `Retry-After` value is negative, EventBridge stops retrying delivery for that event.

How invocation rate affects event delivery

If you set the invocation rate per second to a value much lower than the number of invocations generated, events may not be delivered within the 24 hour retry time for events. For example, if you set the invocation rate to 10 invocations per second, but thousands of events per second are generated, you will quickly have a backlog of events to deliver that exceeds 24 hours. To ensure that no events are lost, set up a dead-letter queue to send events with failed invocations to so you can process the events at a later time. For more information, see Using dead-letter queues to process undelivered events in EventBridge.

Region availability

API destinations to public HTTPS endpoints are supported in the following AWS Regions:

- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- Asia Pacific (Tokyo)
- Asia Pacific (Seoul)
- Asia Pacific (Osaka)
- Asia Pacific (Mumbai)
- Asia Pacific (Hyderabad)

- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Jakarta)
- Canada (Central)
- China (Beijing)
- China (Ningxia)
- Europe (Frankfurt)
- Europe (Zurich)
- Europe (Stockholm)
- Europe (Milan)
- Europe (Spain)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Middle East (UAE)
- Middle East (Bahrain)
- South America (São Paulo)

Create an API destination in Amazon EventBridge

Creating an API destination enables you to specify an HTTP endpoint as the target of a rule.

Each API destination requires a connection. A *connection* specifies the authorization type and credentials to use to authorize with the API destination endpoint. You can choose an existing connection, or create a connection at the same time that you create the API destination. For more information, see ???

To create an API destination using the EventBridge console

1. Log in to AWS using an account that has permissions to manage EventBridge and open the EventBridge console.
2. In the left navigation pane, choose **API destinations**.

3. Scroll down to the **API destinations** table, and then choose **Create API destination**.
4. On the **Create API destination** page, enter a **Name** and Description for the API destination.

   The name must be unique to your account in the current Region. You can use up to 64 uppercase or lowercase letters, numbers, dot (.), dash (-), or underscore (_) characters.

   > **Important**
   >
   > When creating a new API destination for your connection, ensure that the connection reaches the AUTHORIZED state before creating the API destination.

5. Enter an **API destination endpoint** for the API destination.

   The **API destination endpoint** is an HTTP invocation endpoint target for events. The authorization information you include in the connection used for this API destination is used to authorize against this endpoint. The URL must use HTTPS.
6. Enter the **HTTP method** to use to connect to the **API destination endpoint**.
7. (Optional) For **Invocation rate limit per second** field, enter the maximum number of invocations per second to send to the API destination endpoint.

   The rate limit you set may affect how EventBridge delivers events. For more information, see How invocation rate affects event delivery.
8. For **Connection**, do one of the following:
   - Choose **Use an existing connection**, and then select the connection to use for this API destination.
   - Choose **Create a new connection**, and then enter the details for the connection to create.

     For detailed instructions, see Creating connections.
9. Choose **Create**.

Creating rules that send events to an API destination in EventBridge

After you create an API destination, you can select it as the target of a rule. To use an API destination as a target, you must provide an IAM role with the correct permissions. For more information, see ???

Selecting an API destination as a target is part of creating the rule.

To create a rule that sends events to an API destination using the console

1. Follow the steps in the ??? procedure.
2. In the ??? step, when prompted to choose an API destination as the target type:
   a. Select **EventBridge API destination**.
   b. Do one of the following:
      - Choose **Use an existing API destination** and select an existing API destination
      - Choose **Create a new API destination** and specify the necessary setting to define your new API destination.

        For more information on specifying the required settings, see ???.
   c. (Optional): To specify header parameters for the event, under **Header Parameters** choose **Add header parameter**.

      Next, specify the key and value for the header parameter.
   d. (Optional): To specify query string parameters for the event, under **Query string parameters** choose **Add query string parameter**.

      Next, specify the key and value for the query string parameter.
3. Complete creating the rule following the procedure steps.

Sending CloudEvents events to API destinations

CloudEvents is a vendor-neutral specification for event formatting, with the goal of providing interoperability across services, platforms and systems. You can use EventBridge to transform AWS service events to CloudEvents before they are sent to a target, such as an API destination.

> **Note**
>
> The following procedure explains how to transform source events into *structured-mode* CloudEvents. In the CloudEvents specification, a structured-mode message is one where the entire event (attributes and data) is encoded into the payload of the event.

For more information on the CloudEvents specification, see cloudevents.io.

To transform AWS events to the CloudEvents format using the console

To transform events to the CloudEvents format prior to delivery to a target, you start by creating an event bus rule. As part of defining the rule, you use an input transformer to have EventBridge transform events prior to sending to the target you specify.

1. Follow the steps in the ??? procedure.
2. In the ??? step, when prompted to choose an API destination as the target type:

   a. Select **EventBridge API destination**.

   b. Do one of the following:

   - Choose **Use an existing API destination** and select an existing API destination
   - Choose **Create a new API destination** and specify the necessary setting to define your new API destination.

     For more information on specifying the required settings, see ???.

   c. Specify the necessary Content-Type header parameters for the CloudEvents events:

   - Under **Header Parameters** choose **Add header parameter**.
   - For **key**, specify `Content-Type`.

     For **value**, specify `application/cloudevents+json; charset=UTF-8`.
3. Specify an execution role for your target.
4. Define an input transformer to transform the source event data into the CloudEvents format:

   a. Under **Additional settings**, for **Configure target input**, choose **Input transformer**.

   Then choose **Configure input transformer**.

   b. Under **Target input transformer**, specify the **Input path**.

   In the input path below, the region attribute is a custom *extension attribute* of the CloudEvents format. As such it is not required for adherence to the CloudEvents specification.

   CloudEvents allows you to use and create extension attributes not defined in the core specification. For more information, including a list of known extension attributes, see CloudEvents Extension Attributes in the CloudEvents specification documentation on GitHub.

```
{
  "detail": "$.detail",
  "detail-type": "$.detail-type",
  "id": "$.id",
  "region": "$.region",
  "source": "$.source",
  "time": "$.time"
}
```

c. For **Template**, enter the template to transform the source event data to the CloudEvents format.

   In the template below, `region` is not strictly required, since the `region` attribute in the input path is an extension attribute to the CloudEvents specification.

```
{
  "specversion":"1.0",
  "id":<id>,
  "source":<source>,
  "type":<detail-type>,
  "time":<time>,
  "region":<region>,
  "data":<detail>
}
```

5. Complete creating the rule following the procedure steps.

API destination partners in Amazon EventBridge

Use the information provided by the following AWS Partners to configure an API destination and connection for their service or application.

Cisco Cloud Observability

API destination invocation endpoint URL:

`https://tenantName.observe.appdynamics.com/rest/awsevents/aws-eventbridge-integration/endpoint`

Supported authorization types:

OAuth client credentials

OAuth tokens are refreshed when a 401 or 407 response is returned

Additional authorization parameters required:

Cisco AppDynamics Client ID and Client Secret

OAuth endpoint:

`https://tenantName.observe.appdynamics.com/auth/tenantId/default/oauth2/token`

The following OAuth key/value pair parameters:

Type	Key	Value
Body Field	`grant_type`	`client_credentials`
Header	`Content-Type`	`application/x-www-form-urlencoded; charset=utf-8`

Cisco AppDynamics documentation:

AWS events ingestion

Commonly used API operations:

Not applicable

Additional information:

Choosing **Cisco AppDynamics** from the **Partner destinations** drop-down menu prefills the necessary OAuth information, including the header and body key/value pairs required for API calls.

For additional information, see AWS events ingestion in the *Cisco AppDynamics* documentation.

Confluent

API destination invocation endpoint URL:

Typically the following format:

`https://random-id.region.aws.confluent.cloud:443/kafka/v3/clusters/cluster-id/topics/topic-name/records`

For more information, see Find the REST endpoint address and cluster ID in the Confluent documentation.

Supported authorization types:

Basic

Additional authorization parameters required:

Not applicable

Confluent documentation:

Produce Records

Confluent REST Proxy for Apache Kafka

Commonly used API operations:

POST

Additional information:

To transform the event data into a message that the endpoint can process, create a target input transformer.

- To generate a record without specifying a Kafka partitioning key, use the following template for your input transformer. No input path is required.

```
{
  "value":{
      "type":"JSON",
      "data":aws.events.event.json
  },
}
```

- To generate a record using an event data field as the Kafka partitioning key, follow the input path and template example below. This example defines the input path for the `orderId` field, and then specifies that field as the partition key.

  First, define the input path for the event data field:

  ```
  {
    "orderId":"$.detail.orderId"
  }
  ```

  Then, use the input transformer template to specify the data field as the partition key:

  ```
  {
    "value":{
      "type":"JSON",
      "data":aws.events.event.json
    },
    "key":{
      "data":"<orderId>",
      "type":"STRING"
    }
  }
  ```

Coralogix

API destination invocation endpoint URL

For a full list of endpoints, see Coralogix API Reference.

Supported authorization types

API Key

Additional authorization parameters required

Header `"x-amz-event-bridge-access-key"`, the value is the Coralogix API Key

Coralogix documentation

Amazon EventBridge authentication

Commonly used API operations

US: https://ingress.coralogix.us/aws/event-bridge

Singapore: https://ingress.coralogixsg.com/aws/event-bridge

Ireland: https://ingress.coralogix.com/aws/event-bridge

Stockholm: https://ingress.eu2.coralogix.com/aws/event-bridge

India: https://ingress.coralogix.in/aws/event-bridge

Additional information

The events are stored as log entries with `applicationName=[AWS Account]` and `subsystemName=[event.source]`.

Datadog

API destination invocation endpoint URL

For a full list of endpoints, see Datadog API Reference.

Supported authorization types

API Key

Additional authorization parameters required

None

Datadog documentation

Authentication

Commonly used API operations

POST https://api.datadoghq.com/api/v1/events

POST https://http-intake.logs.datadoghq.com/v1/input

Additional information

Endpoint URLs differ depending on the location of your Datadog organization. For the correct URL for your organization, see documentation.

Dynatrace

API destination invocation endpoint URL:

- Logs: https://*environment-id*.live.dynatrace.com/api/v2/logs/ingest

- Davis events: https://*environment-id*.live.dynatrace.com/platform/ingest/v1/events
- SDLC events: https://*environment-id*.live.dynatrace.com/platform/ingest/v1/events.sdlc
- Security events: https://*environment-id*.live.dynatrace.com/platform/ingest/v1/events.security

Supported authorization types:

API Key

Additional authorization parameters required:

Type	Key	Value
Header	`Authorization`	Api-Token *DYNATRACE_API_KEY*

Dynatrace documentation:

- Log ingestion
- How to ingest data (events)
- Ingest Amazon ECR vulnerability findings and scan events

Commonly used API operations:

POST

Additional information:

Each endpoint type has its token scope, which is required for successful authentication. For more information, see Ingestion APIs in the Dynatrace documentation.

Freshworks

API destination invocation endpoint URL

For a list of endpoints, see https://developers.freshworks.com/documentation/

Supported authorization types

Basic, API Key

Additional authorization parameters required

Not applicable

Freshworks documentation

Authentication

Commonly used API operations

https://developers.freshdesk.com/api/#create_ticket

https://developers.freshdesk.com/api/#update_ticket

https://developer.freshsales.io/api/#create_lead

https://developer.freshsales.io/api/#update_lead

Additional information

None

MongoDB

API destination invocation endpoint URL

https://data.mongodb-api.com/app/*App ID*/endpoint/

Supported authorization types

API Key

Email/Password

Custom JWT Authentication

Additional authorization parameters required

None

MongoDB documentation

Atlas Data API

Endpoints

Custom HTTPS Endpoints

Authentication

Commonly used API operations

None

Additional information

None

Momento

API destination invocation endpoint URL:

https://api.cache.*region*.prod.a.momentohq.com/cache/*cacheName*

https://api.cache.*region*.prod.a.momentohq.com/topics/*cacheName*/*topicName*

Supported authorization types:

API Key

Additional authorization parameters required:

Type	Key	Value
Header	`Authorization`	*MOMENTO_API_KEY*

Momento documentation:

Momento + Amazon EventBridge

Using the Momento Topics API

API reference for Momento Cache

Commonly used API operations:

For caches: PUT, DELETE

For topics: POST

Additional information:

When updating or deleting a cache, include the following query string parameters in the configuration of your rule target:

- The key you want to update in your Momento cache
- The Time-To-Live (TTL) for the cache item

For example, if your input event included a `details` field with those values:

```
key: $.details.key ttl_seconds: $.details.ttl_seconds
```

New Relic

API destination invocation endpoint URL

For more information, see Our EU and US region data centers.

Events

US– https://insights-collector.newrelic.com/v1/accounts/*YOUR_NEW_RELIC_ACCOUNT_ID*/events

EU– https://insights-collector.eu01.nr-data.net/v1/accounts/*YOUR_NEW_RELIC_ACCOUNT_ID*/events

Metrics

US– https://metric-api.newrelic.com/metric/v1

EU– https://metric-api.eu.newrelic.com/metric/v1

Logs

US– https://log-api.newrelic.com/log/v1

EU– https://log-api.eu.newrelic.com/log/v1

Traces

US– https://trace-api.newrelic.com/trace/v1

EU– https://trace-api.eu.newrelic.com/trace/v1

Supported authorization types

API Key

New Relic documentation

Metric API

Event API

Log API

Trace API

Commonly used API operations

Metric API

Event API

Log API

Trace API

Additional information

Metric API limits

Event API limits

Log API limits

Trace API limits

Operata

API destination invocation endpoint URL:

https://api.operata.io/v2/aws/events/contact-record

Supported authorization types:

Basic

Additional authorization parameters required:

None

Operata documentation:

How do I create, view, change and revoke API Tokens?

Operata AWS Integration using Amazon EventBridge Scheduler Pipes

Commonly used API operations:

`POST https://api.operata.io/v2/aws/events/contact-record`

Additional information:

The `username` is the Operata Group ID and the password is your API token.

Salesforce

API destination invocation endpoint URL

Sobject– https:// *myDomainName*.my.salesforce.com/services/data/*versionNumber*/sobjects /*SobjectEndpoint*/*

Custom platform events– https://*myDomainName*.my.salesforce.com/services/data /*versionNumber*/sobjects/*customPlatformEndpoint*/*

For a full list of endpoints, see Salesforce API Reference

Supported authorization types

OAuth client credentials

OAUTH tokens are refreshed when a 401 or 407 response is returned.

Additional authorization parameters required

Salesforce Connected App Client Id and Client Secret.

One of the following authorization endpoints:

- **Production–** https://*MyDomainName*.my.salesforce.com./services/oauth2/token
- **Sandbox without enhanced domains–** https://*MyDomainName*-- *SandboxName*.my.salesforce.com/services /oauth2/token
- **Sandbox with enhanced domains–** https://*MyDomainName*--*SandboxName*.sandbox.my.salesforce.com/services/oauth2/token

The following key/value pair:

Key	Value
grant_type	client_credentials

Salesforce documentation

REST API Developer Guide

Commonly used API operations

Working with Object Metadata

Working with Records

Additional information

For a tutorial explaining how to use the EventBridge console to create a connection to Salesforce, an API Destination, and a rule to route information to Salesforce, see ???.

Slack

API destination invocation endpoint URL

For a list of endpoints and other resources, see Using the Slack Web API

Supported authorization types

OAuth 2.0

OAUTH tokens are refreshed when a 401 or 407 response is returned.

When you create a Slack application and install it to your workspace, an OAuth bearer token will be created on your behalf to be used for authenticating calls by your API destination connection.

Additional authorization parameters required

Not applicable

Slack documentation

Basic app setup

Installing with OAuth

Retrieving messages

Sending messages

Sending messages using Incoming Webhooks

Commonly used API operations

https://slack.com/api/chat.postMessage

Additional information

When configuring your EventBridge rule there are two configurations to highlight:

- Include a header parameter that defines the content type as “application/json;charset=utf-8”.
- Use an input transformer to map the input event to the expected output for the Slack API, namely ensure that the payload sent to the Slack API has “channel” and “text” key/value pairs.

Shopify

API destination invocation endpoint URL

For a list of endpoints and other resouces and methods, see Endpoints and requests

Supported authorization types

OAuth, API Key

> **Note**
>
> OAUTH tokens are refreshed when a 401 or 407 response is returned.

Additional authorization parameters required

Not applicable

Shopify documentation

Authentication and authorization overview

Commonly used API operations

POST - /admin/api/2022-01/products.json

GET - admin/api/2022-01/products/{product_id}.json

PUT - admin/api/2022-01/products/{product_id}.json

DELETE - admin/api/2022-01/products/{product_id}.json

Additional information

Create an app

Amazon EventBridge webhook delivery

Access tokens for custom apps in the Shopify admin

Product

Shopify Admin API

Splunk

API destination invocation endpoint URL

https://*SPLUNK_HEC_ENDPOINT*:*optional_port*/services/collector/raw

Supported authorization types

Basic, API Key

Additional authorization parameters required

None

Splunk documentation

For both authorization types, you need an HEC token ID. For more information, see Set up and use HTTP Event Collector in Splunk Web.

Commonly used API operations

POST https://*SPLUNK_HEC_ENDPOINT*:*optional_port*/services/collector/raw

Additional information

API Key – When configuring the endpoint for EventBridge, the API key name is “Authorization” and value is the Splunk HEC token ID.

Basic (Username/Password) – When configuring the endpoint for EventBridge, the username is “Splunk” and the password is the Splunk HEC token ID.

Sumo Logic

API destination invocation endpoint URL

HTTP Log and Metric Source endpoint URLs will be different for every user. For more information, see HTTP Logs and Metrics Source.

Supported authorization types

Sumo Logic doesn’t require authentication on their HTTP Sources because there’s a unique key baked into the URL. For this reason, you should make sure to treat that URL as a secret.

When you configure the EventBridge API destination, an authorization type is required. To meet this requirement, select API Key and give it a key name of “dummy-key” and a key value of “dummy-value”.

Additional authorization parameters required

Not applicable

Sumo Logic documentation

Sumo Logic has already built hosted sources to collect logs and metrics from many AWS services and you can use the information on their website to work with those sources. For more information, see Amazon Web Services.

If you’re generating custom events from an application and want to send them to Sumo Logic as either logs or metrics, then use EventBridge API Destinations and Sumo Logic HTTP Log and Metric Source endpoints.

- To sign up and create a free Sumo Logic instance, see Start your free trial today.
- For more information about using Sumo Logic, see HTTP Logs and Metrics Source.

Commonly used API operations

POST https://endpoint4.collection.us2.sumologic.com/receiver/v1/http/*UNIQUE_ID_PER_COLLECTOR*

Additional information

None

Zendesk

API destination invocation endpoint URL

https://developer.zendesk.com/rest_api/docs/support/tickets

Supported authorization types

Basic, API Key

Additional authorization parameters required

None

Zendesk documentation

Security and Authentication

Commonly used API operations

POST https://*your_Zendesk_subdomain*/api/v2/tickets

Additional information

API requests EventBridge makes count against your Zendesk API limits. For information about Zendesk limits for your plan, see Usage limits.

To better safeguard your account and data, we recommend using an API key rather than basic sign-in credentials authentication.

Connections for API targets in Amazon EventBridge

To enable event buses and pipes to target custom resources, such as HTTPS APIs, you create connections. A *connection* defines the authorization method and credentials for EventBridge to use in connecting to a given resource. If you are connecting to a private API, such as a private API in an Amazon Virtual Private Cloud (Amazon VPC), you can also use the connection to define secure point-to-point network connectivity.

You can create connections to target:

- Public APIs, such as third-party SaaS applications.
- Private APIs, such as custom resources that reside in an Amazon VPC or on-premise.

EventBridge creates connections to private HTTPS endpoints by utilizing *resource configurations* created in Amazon VPC Lattice. A resource configuration is a logical object that identifies a resource, and specifies who can access it and how.

Use connections with:

- API destinations in EventBridge

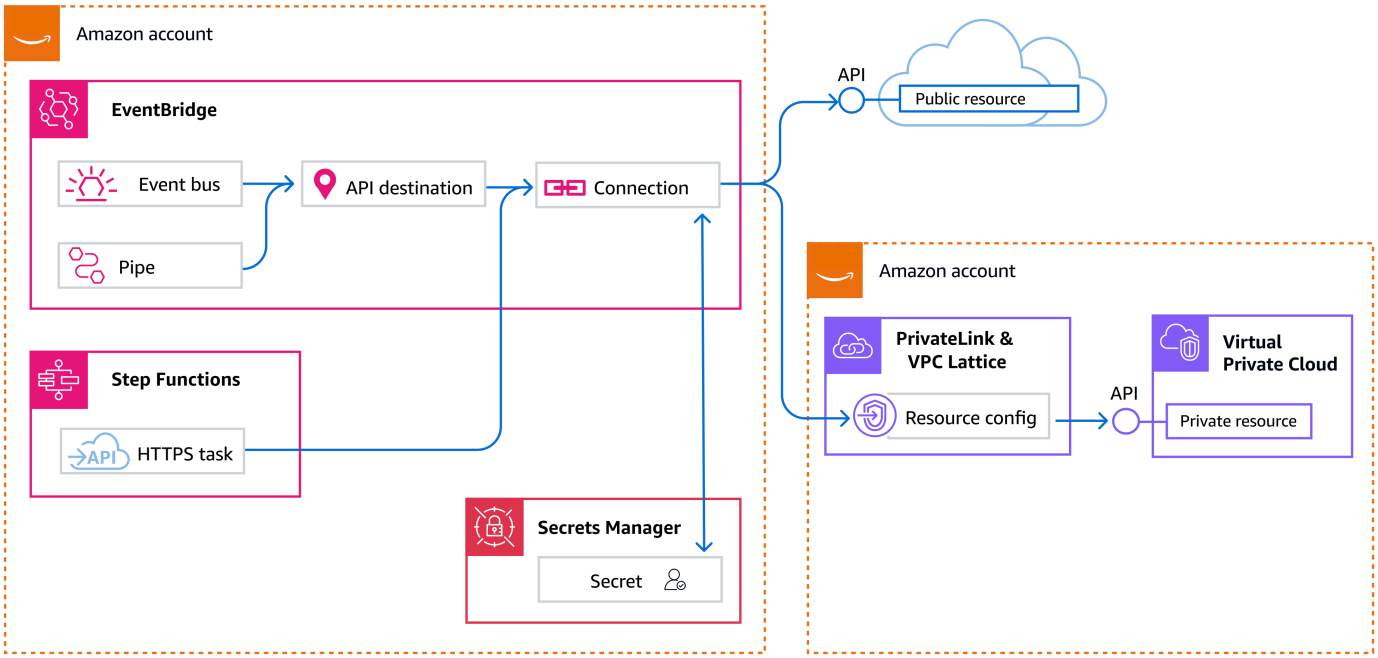
  When you create an API destination, you specify a connection to use for it. You can choose an existing connection from your account, or create a connection when you create an API destination.

  For more information, see API destinations.

- HTTP Endpoint tasks in AWS Step Functions

  An HTTP Endpoint task is a type of Task workflow state that lets you call HTTPS APIs in your workflows. These APIs can be public, such as Salesforce and Stripe, or private APIs that reside in an Amazon VPC or on-premise. The task uses a connection to specify the authorization type and credentials to use for authorizing the API. For private APIs, the connection also defines the network path to the API.

  For more information, see Call HTTPS APIs in Step Functions workflows in the *Step Functions User Guide*.



Connections are reusable. You can use the same connection to the same API for multiple EventBridge API destinations or Step Functions tasks, as long as the authentication method is the same. If API destinations or tasks require different authentication, then you must create separate connections.

Storing connection authorization parameters in AWS Secrets Manager

When you configure the authorization settings and create a connection, it creates a secret in AWS Secrets Manager to securely store the authorization information. You can also add additional parameters to include in the connection as appropriate for your HTTPS endpoint target.

EventBridge connections support the following authentication methods: basic, OAuth, and API Key. For more information, see Connection authorization methods.

While by default EventBridge uses an AWS owned key to encrypt and decrypt the connection secret, you can specify a customer managed key for EventBridge to use instead. For more information, see Encrypting connections.

Connecting to private APIs in EventBridge

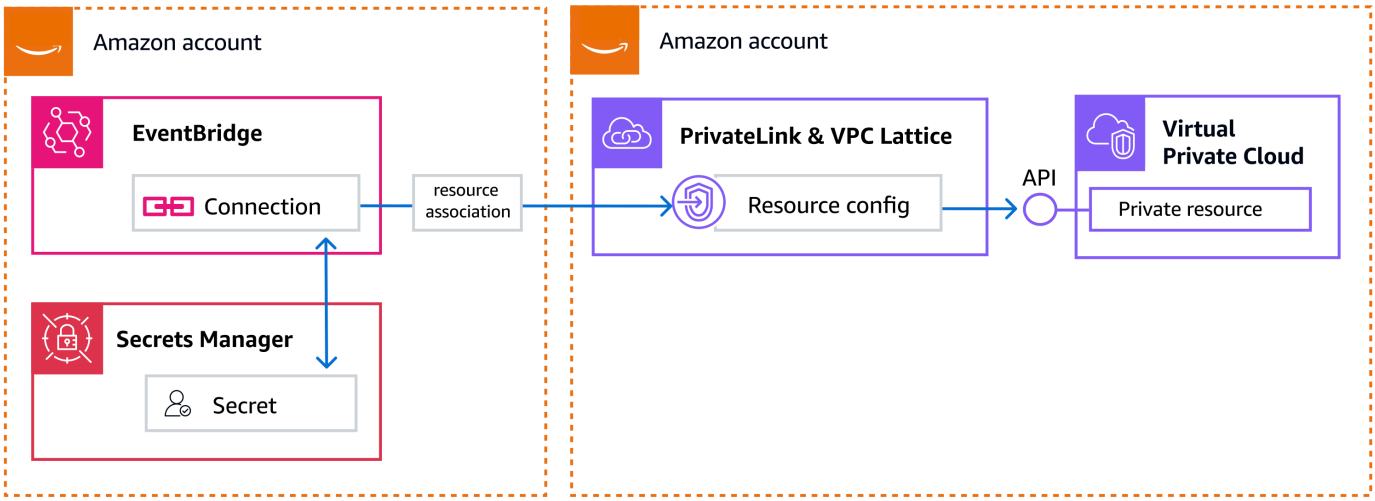
You can create connections to private HTTPS endpoints, to provide secure point-to-point network access to resources in VPCs or on-premises without having to traverse the public internet. For

example, you can create a connection to access an HTTPS-based application behind an Amazon Elastic Load Balancer.

EventBridge creates connections to private HTTPS endpoints by utilizing *resource configurations* created in VPC Lattice. A resource configuration is a logical object that identifies the resource and specifies how and who can access it. To create a connection to a private API in EventBridge, you specify the resource configuration for the private API. For more information, see Resource configuration in VPC Lattice in the *Amazon VPC Lattice User Guide*.

EventBridge then creates a *resource association* that enables EventBridge to access the private API. For more information, see Manage resource associations in the *Amazon VPC Lattice User Guide*.

While EventBridge manages the resource association, it creates the association using your credentials, so you retain visibility into the resource association operation.



You can create connections that access private APIs in other AWS accounts. For more information, see Cross-account private APIs.

Connecting to private APIs in other AWS accounts

EventBridge supports connections to private APIs across accounts in the same Region.

For you to create a connection to a private API in another AWS account, the owner of that account must first share a VPC Lattice resource configuration for that private API with you. To do this, they share the resource with you in AWS Resource Access Manager. AWS RAM enables secure sharing of resources across AWS accounts, within organizational units (OUs), and integrates with AWS Identity

and Access Management roles and users. Once you've accepted the resource share in AWS RAM you can specify the shared VPC Lattice resource configuration when creating a connection.

For more information on AWS RAM, see the following topics in the *AWS Resource Access Manager User Guide*:

- Benefits of AWS RAM
- How resource sharing works
- Access AWS resources shared with you

EventBridge does not support connections to private APIs across Regions. However, to target a private API in a different Region from your event bus:, you can:

1. Define an event bus rule that targets a second event bus that does reside in the same Region as the desired private API.
2. Create a connection for the second event bus to target the private API.

For more information, see Sending and receiving events between AWS Regions in Amazon EventBridge.

Permissions for connecting to private APIs

The following policy example includes the minimal necessary permissions for creating a connection to a private API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "vpc-lattice:CreateServiceNetworkResourceAssociation",
                "vpc-lattice:GetResourceConfiguration",
                "vpc-lattice:AssociateViaAWSService-EventsAndStates",
                "events:CreateConnection"
            ],
            "Resource": [
                "*"
```

```
            ],
            "Effect": "Allow"
        }
    ]
}
```

The following policy example includes the minimal necessary permissions for updating a connection to a private API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "vpc-lattice:CreateServiceNetworkResourceAssociation",
                "vpc-lattice:GetResourceConfiguration",
                "vpc-lattice:AssociateViaAWSService-EventsAndStates",
                "events:UpdateConnection"
            ],
            "Resource": [
                "*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Monitoring creation of connections to private APIs

When you create a connection to a private API, the following logs are generated:

In the account in which the connection was created, AWS CloudTrail logs a `CreateServiceNetworkResourceAssociation` event.

In this log, `sourceIPAddress`, `userAgent`, and `serviceNetworkIdentifier` are set to the EventBridge service principal, `events.amazonaws.com`.

```
{
```

```
  "eventTime": "2024-11-21T00:00:00Z",
  "eventSource": "vpc-lattice.amazonaws.com",
  "eventName": "CreateServiceNetworkResourceAssociation",
  "awsRegion": "region",
  "sourceIPAddress": "events.amazonaws.com",
  "userAgent": "events.amazonaws.com",
  "requestParameters": {
    "x-amzn-vpc-lattice-association-source-arn": "***",
    "x-amzn-vpc-lattice-service-network-identifier": "***",
    "clientToken": "token",
    "serviceNetworkIdentifier": "events.amazonaws.com",
    "resourceConfigurationIdentifier": "arn:partition:vpc-lattice:region:account-
id:resourceconfiguration/resource-configuration-id",
    "tags": {
        "ManagedByServiceAWSEventBridge": "account-id:connection-name"
    }
}
```

In the account which contains the private API , AWS CloudTrail logs a `CreateServiceNetworkResourceAssociationBySharee` event.

This log includes:

- `callerAccountId`: The AWS account in which the connection was created
- `accountId`: The AWS account that contains the private API.
- `resource-configuration-arn`: The VPC Lattice resource configuration for the private API.

```
{
  "eventTime": "2024-11-21T06:31:42Z",
  "eventSource": "vpc-lattice.amazonaws.com",
  "eventName": "CreateServiceNetworkResourceAssociationBySharee",
  "awsRegion": "region",
  "sourceIPAddress": "vpc-lattice.amazonaws.com",
  "userAgent": "user-agent",
  "additionalEventData": {
      "callerAccountId": "consumer-account-id"
  },
  "resources": [
      {
          "accountId": "provider-account-id",
          "type": "AWS::VpcLattice::ServiceNetworkResourceAssociation",
```

```
            "ARN": "resource-configuration-arn"
        }
    ]
}
```

In the case of cross-account connections to private APIs, the account containing the connection will not receive AWS CloudTrail or VPC Lattice logs for the invocation of the private API.

Managing service network resource associations for connections

When you specify the VPC Lattice resource configuration for the private API to which you want to connect, EventBridge enables the connection by creating a resource association between the VPC Lattice resource configuration and a VPC Lattice service network owned by the EventBridge service. While EventBridge manages the resource association, it creates the association using your credentials, so you retain visibility into the resource association. This means you can list and describe the resource associations.

Use describe-connection to return a connection description that includes the Amazon Resource Names (ARNs) of the resource configuration and resource association.

You cannot delete resource associations created by EventBridge. If you delete a connection, EventBridge deletes any corresponding resource associations.

For more information, see Manage resource associations in the *Amazon VPC Lattice User Guide*.

Connecting to on-premise private APIs

Using access to VPC resources through AWS PrivateLink and VPC Lattice, you can connect to on-premise private APIs. To do so, you must configure a network route between your VPC and your on-premise environment. For example, you can use AWS Direct Connect or AWS Site-to-Site VPN to establish such a route.

Provider considerations for cross-account connections in EventBridge

To create a connection to a private API in another AWS account, the owner of that account must share a VPC Lattice resource configuration for the private API with you. A resource configuration is a logical object that identifies the API and specifies how and who can access it. The *provider* account--that is, the account sharing the VPC Lattice resource configuration for the private API with another account--shares the VPC Lattice resource configuration using AWS RAM.

If your account is the provider of a VPC Lattice resource configuration, keep the following considerations in mind:

Resource policy for resource configurations for cross-account private APIs

By default, creating a AWS RAM resource share includes the necessary share policy, `AWSRAMPermissionVpcLatticeResourceConfiguration`. If you create a customer managed permission policy, you must include the necessary permissions.

The following policy example provides the minimum necessary permissions for EventBridge to create the resource association necessary for a connection to a private API.

- `vpc-lattice:GetResourceConfiguration` allows EventBridge to retrieve the Amazon VPC Lattice resource configuration you specify.
- `vpc-lattice:CreateServiceNetworkResourceAssociation` allows EventBridge to create the resource association from the VPC Lattice resource configuration you specify.
- `vpc-lattice:AssociateViaAWSService-EventsAndStates` allows EventBridge to create a resource association to a VPC Lattice service network owned by the service.

```
{
    "Effect": "Allow",
    "Action": [
      "vpc-lattice:CreateServiceNetworkResourceAssociation",
      "vpc-lattice:GetResourceConfiguration",
      "vpc-lattice:AssociateViaAWSService-EventsAndStates"
      ]
}
```

For more information, see Managing permissions in AWS RAM in the *AWS Resource Access Manager User Guide*.

Provider monitoring of connection creation

When another account creates an EventBridge connection using a VPC Lattice resource configuration you have shared, AWS CloudTrail logs a `CreateServiceNetworkResourceAssociationBySharee` event. For more information, see Monitoring connection creation.

Configuring security groups for access to private APIs

With VPC Lattice, you can create and assign security groups to enforce additional network-level security protections for your target API and resource gateway. In order for EventBridge and Step Functions to access your private API successfully, the security groups on the target API and resource gateway must to be configured correctly. If not configured correctly, the services will return "Connection Timed Out" errors when attempting to call your API.

For your target API, your security group must be configured to allow all inbound TCP traffic on port 443 from the security group for your resource gateway.

For your resource gateway, your security group must be configured to allow the following:

- All inbound IPv6 TCP traffic across all ports from the ::/0 IPv6 CIDR range.
- All inbound IPv4 TCP traffic across all ports from the 0.0.0.0/0 IPv6 CIDR range.
- All outbound TCP traffic on port 443 to the security group used by your target resource, for the IP protocol your target API accepts (IPv4 or IPv6).

For more information, see the following topics in the *Amazon VPC Lattice User Guide*:

- Control traffic in VPC Lattice using security groups
- Resource gateway in VPC Lattice

Creating connections to private APIs

The following steps walk you through how to create a connection to a private API. For detailed instructions that include all configuration options for connections, including creating connections to public APIs, see Creating connections.

Define the connection

The following steps walk you through how to create a connection to a private API endpoint. For instructions on creating connections to public APIs, see Creating connections.

1. Open the EventBridge console.
2. In the left navigation pane, under **Integration**, choose **Connections**.
3. Choose **Create connection**.
4. On the **Create connection** page, enter a **Connection name** and **Description**.

Configure the invocation endpoint

Next, use the **Configure invocation** section to specify the HTTPS endpoint you want the connection to invoke.

1. For **API type**, choose **Private**.
2. Specify the Amazon VPC Lattice resource configuration to use to connect to it.

   Under **Private API**:

   - To use an existing VPC Lattice resource configuration, choose a resource configuration from the drop-down menu.
   - To create a new VPC Lattice resource configuration, choose **New resource configuration**.

     You are taken to the Amazon VPC Lattice; service console, where you can create a new configuration. for more information, see Create a resource configuration in the *Amazon VPC Lattice User Guide*.

Configure the endpoint authorization

Lastly, specify the authorization settings to use to access the endpoint.

EventBridge supports basic, OAuth client credentials, and API key authentication methods.

1. Under **Configure authorization**, choose **Custom configuration**.
2. For **Authorization type**, select the authorization method for the connection to use.
3. Specify the authorization configuration details for the authorization method you chose:
   - **Basic**

     Enter the **Username** and **Password** to use to authorize with the HTTPS endpoint.
   - **OAuth Client Credentials**

     a. For **OAuth authorization endpoint**, choose whether the endpoint to use for connection authorization is a public or private (VPC) endpoint.

     If you choose **Private**, specify the **Private OAuth endpoint resource configuration**:

     - To use an existing VPC Lattice resource configuration, choose a resource configuration from the drop-down menu.

- To create a new VPC Lattice resource configuration, choose **New resource configuration**.

  You are taken to the Amazon VPC Lattice service console, where you can create a new configuration. for more information, see Create a resource configuration in the *Amazon VPC Lattice User Guide*.

b. Specify the following authorization information:

- Authorization endpoint
- HTTPS method
- Client ID
- Client secret

c. Under **OAuth HTTP parameters**, add any additional parameters to include for authorization with the authorization endpoint.

To do so:

- Select a **Parameter** from the drop-down list.
- Enter a **Key** and **Value**.

To include an additional parameter, choose **Add parameter**.

- **API Key**

  Enter the **API key name** and associated **Value** to use for API Key authorization.

4. Under **Invocation Http Parameters**, add any additional parameters to include in the authorization request.

   To add a parameter:

   1. Select a **Parameter** from the drop-down list
   2. Enter a **Key** and **Value**

   To include an additional parameter, choose **Add parameter**.

5. Choose **Create Connection**.

> **Note**
>
> For connections for private endpoints, EventBridge creates the necessary resource association when it create the connection. This can take up to 90 seconds.

Authorization methods for connections in EventBridge

EventBridge connections support the following authorization methods:

- Basic
- API Key

  For Basic and API Key authorization, EventBridge populates the required authorization headers for you.
- OAuth

  For OAuth authorization, EventBridge also exchanges your client ID and secret for an access token and then manages it securely.

  When you create a connection that uses OAuth authorization, you have the option of specifying a public or private authorization endpoint.

Considerations when using OAuth

Keep in mind the following when using OAuth as an authorization method for connections:

- EventBridge refreshes OAuth tokens:
  - When a `401` or `407` response is returned.
  - Proactively during an HTTPS invocation, if the token expires within 60 seconds. This refresh only occurs when EventBridge processes an event for the API destination — it is not a background process. If no events arrive during this window, the next event triggers a synchronous token refresh, which may increase invocation latency or result in a failed invocation while the connection is reauthorizing.
- You can use ??? to be notified when a connection changes state, such as becoming deauthorized.

- We recommend you set the retry policy to greater than 0 for rules using connections that require OAuth. That way, if an OAuth token has expired, EventBridge will refresh the token when retrying the invocation.

  For more information on setting a retry policy for a rule, see ???.

Considerations for connection authorization

When you create a connection, you can also include the header, body, and query parameters that are required for authorization with an endpoint. You can use the same connection for more than one HTTPS endpoint if the authorization for the endpoint is the same.

When you create a connection and add authorization parameters, EventBridge creates a secret in AWS Secrets Manager. The cost of both storing and accessing the Secrets Manager secret is included with the charge for using an API destination.

For information on how to have EventBridge re-authorize the connection once you have updated it to address authorization or connectivity issues, see Updating connections.

> **Note**
>
> To successfully create or update a connection, you must use an account that has permission to use Secrets Manager. The required permission is included in the AWS managed policy: AmazonEventBridgeFullAccess. The same permission is granted to the service-linked role that's created in your account for the connection.

For examples of how to create a CloudFormation template that provisions an EventBridge connection with authentication, see AWS::Events::Connection in the *CloudFormation User Guide*.

Connection states in EventBridge

Below are the connection statuses returned by EventBridge when creating or re-authorizing a connection.

For successful connections to public APIs, EventBridge returns a status of `AUTHORIZED`. For successful connections to private APIs, EventBridge returns a status of `ACTIVE`.

Connectivity failures refer to errors involving the network connectivity of the connection. Connectivity errors result in a connection status of `FAILED_CONNECTIVITY`. *Authorization failures*

refer to errors involving the permissions specified for the connection. Authorization failures result in a status of DEAUTHORIZED.

For information on how to have EventBridge re-authorize the connection once you have updated it to address authorization or connectivity issues, see Updating connections.

EventBridge emits events when the state of a connection changes. For more information, see Connection events.

Public APIs

Authorization method	Success	Connectivity failure	Authorization failure
Basic or API Key	AUTHORIZED	n/a	DEAUTHORIZED
OAuth (public endpoint)	AUTHORIZED	n/a	DEAUTHORIZED
OAuth (private endpoint)	AUTHORIZED	FAILED_CONNECTIVITY	DEAUTHORIZED

Private APIs

Authorization method	Success	Connectivity failure	Authorization failure
Basic or API Key	ACTIVE	FAILED_CONNECTIVITY	DEAUTHORIZED
OAuth (public endpoint)	ACTIVE	FAILED_CONNECTIVITY	DEAUTHORIZED
OAuth (private endpoint)	ACTIVE	FAILED_CONNECTIVITY	DEAUTHORIZED

When you create a connection to a private API, there can be a delay of up to several minutes from when the connection is successfully created to when you can successfully make HTTPS calls to the private API. During this period:

- Private OAuth connection will be in `AUTHORIZING` state, and will transition to `AUTHORIZED` when OAuth token exchange is successfully completed.
- The connection will update to `ACTIVE` status, and EventBridge will retry any invocations from event buses or pipes.

If you are using the connection in an HTTP task within a Step Functions workflow, you can configure retries on `httpTimeoutExceptions` to retry any calls to the private API that happen during this period.

Creating connections for API targets in EventBridge

The following steps walk you through how to create a connection to an HTTPS endpoint.

Steps

- Define the connection
- Configure the invocation endpoint
- Configure the endpoint authorization
- Configure encryption

Define the connection

1. Open the EventBridge console.
2. In the left navigation pane, under **Integration**, choose **Connections**.
3. Choose **Create connection**.
4. On the **Create connection** page, enter a **Connection name** and **Description**.

Configure the invocation endpoint

Next, use the **Configure invocation** section to specify the API type you want the connection to invoke. EventBridge connections support public and private APIs.

- For **API type**, choose whether the endpoint is a public or private API.

If you choose a private API, specify the VPC Lattice resource configuration to use to connect to it.

Under **Private API**:

- To use an existing VPC Lattice resource configuration, choose a resource configuration from the drop-down menu.
- To create a new VPC Lattice resource configuration, choose **New Resource configuration**.

  You are taken to the Amazon VPC Lattice service console, where you can create a new configuration. for more information, see Create a resource configuration in the *Amazon VPC Lattice User Guide*.

Configure the endpoint authorization

Next, specify the authorization settings to use to access the endpoint.

EventBridge supports basic, OAuth client credentials, and API key authentication methods.

1. For **Configure authorization**, choose the type of endpoint to which you want to connect.

   Custom endpoint

   If the connection is to an endpoint other than a partner endpoint, choose **Custom configuration**.

   1. For **Authorization type**, select the authorization method for the connection to use.
   2. Specify the authorization configuration details for the authorization method you chose:
      - **Basic**

        Enter the **Username** and **Password** to use to authorize with the HTTPS endpoint.
      - **OAuth Client Credentials**

        a. For **OAuth authorization endpoint**, choose whether the endpoint to use for connection authorization is a public or private endpoint.

        If you choose **Private**, specify the **Private OAuth endpoint resource configuration**:

- To use an existing resource configuration, choose a resource configuration from the drop-down menu.
- To create a new resource configuration, choose **New resource configuration**.

  You are taken to the Amazon VPC Lattice; service console, where you can create a new configuration. for more information, see Create a resource configuration in the *Amazon VPC Lattice User Guide*.

b. Specify the following authorization information:

- Authorization endpoint
- HTTP method
- Client ID
- Client secret

c. Under **OAuth HTTP parameters**, add any additional parameters to include for authorization with the authorization endpoint.

To do so:

- Select a **Parameter** from the drop-down list.
- Enter a **Key** and **Value**.

To include an additional parameter, choose **Add parameter**.

- **API Key**

  Enter the **API key name** and associated **Value** to use for API Key authorization.

Partner endpoint

If the connection is to a partner endpoint, choose **Use partner template**. For a list of available partner endpoints, see API destination partners.

1. From **Partner destination**, select the partner to which to connect.

   Under **Authorization type**, EventBridge enables the authorization methods available for the partner.

2. For **Authorization type**, select the authorization method for the connection to use.
3. Specify the authorization configuration details for the authorization method you chose:
   - **Basic**

     Enter the **Username** and **Password** to use to authorize with the HTTP endpoint.
   - **OAuth Client Credentials**

     a. For **OAuth authorization endpoint**, choose whether the endpoint to use for connection authorization is a public or private (VPC) endpoint.

     If you choose **Private**, specify the **Private OAuth endpoint resource configuration**:

     - To use an existing VPC Lattice resource configuration, choose a resource configuration from the drop-down menu.
     - To create a new VPC Lattice resource configuration, choose **New resource configuration**.

       You are taken to the Amazon VPC Lattice service console, where you can create a new configuration. for more information, see Create a resource configuration in the *Amazon VPC Lattice User Guide*.

     b. Specify the following authorization information:

     - Authorization endpoint
     - HTTPS method
     - Client ID
     - Client secret

     c. Under **OAuth HTTP parameters**, add any additional parameters to include for authorization with the authorization endpoint.

     To do so:

     - Select a **Parameter** from the drop-down list.
     - Enter a **Key** and **Value**.

     To include an additional parameter, choose **Add parameter**.

- **API Key**

  Enter the **API key name** and associated **Value** to use for API Key authorization.

2. Under **Invocation Http Parameters**, add any additional parameters to include in the authorization request.

   To add a parameter:

   1. Select a **Parameter** from the drop-down list
   2. Enter a **Key** and **Value**

   To include an additional parameter, choose **Add parameter**.

Configure encryption

Lastly, specify the type of KMS key you want EventBridge to use when encrypting and decrypting the authorization parameters that it stores as a secret in AWS Secrets Manager.. By default, EventBridge uses an AWS owned key.

For more information, see Encrypting connections.

1. Choose the KMS key for EventBridge to use when encrypting the connection secret.
   - Choose **Use AWS owned key** for EventBridge to encrypt the secret using an AWS owned key.

     This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

     This is the default.
   - Choose **Use customer managed key** for EventBridge to encrypt the secret using the customer managed key that you specify or create.

     Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

     - Specify an existing customer managed key, or choose **Create a new KMS key**.

EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

2. Choose **Create Connection**.

> **Note**
>
> For connections for private endpoints, EventBridge creates the necessary resource association when it create the connection. This can take up to 90 seconds.

Updating connections in EventBridge

You can update existing connections.

Updating authentication of connections

When you update certain authentication parameters, EventBridge re-authorizes and verifies network connectivity of the connection accordingly.

Updating the connection with authorization parameters of a different authorization type, without updating the authorization type itself, has no effect.

When EventBridge re-authenticates connections to public APIs

For connections to public APIs, EventBridge re-authorizes the connection if you:

- Update the connection to invoke a private API instead.
- Update the connection authorization method *and* authorization parameters.

When EventBridge re-authenticates connections to private APIs

For connections to private APIs, if you do any of the following:

- Update the connection to invoke a different private API, or a public API.
- Update the connection authorization method.
- Update the connection authorization parameters for the specified authorization method.

EventBridge does the following:

- Re-authorizes the connection.
- Verifies the network connectivity of the connection.

  If you have updated the connection to use a different private API, or a public API, EventBridge also removes or updates existing network connectivity artifacts as appropriate.

To update a connection using the EventBridge console

1. Log in to AWS using an account that has permissions to manage EventBridge and open the EventBridge console.
2. In the left navigation pane, under **Integration**, choose **Connections**.
3. In the **Connections** table, choose the connection to edit.
4. On the **Connection details** page, choose **Edit**.
5. Update the values for the connection, and then choose **Update**.

   For information on updating the KMS key for EventBridge to use when encrypting a connection, see Updating AWS KMS keys.

To update a connection using the AWS CLI

- Use the `update-connection` command.

  To update a connection's invocation or authorization endpoint from a private API to a public API, you must specify an empty string ("") for the `ResourceConfigurationArn` parameter. Specifying `null` for this parameter has no effect.

  The following example updates the connection to use a public API, by setting the `ResourceConfigurationArn` that represents a private API resource configuration to an empty string.

```
aws events update-connection \
--name myConnection \
--authorization-type BASIC \
--auth-parameters '{"BasicAuthParameters": {"Username": "username", "Password":
 "***"}}' \
--region us-east-1 \
--invocation-connectivity-parameters ResourceParameters={ResourceConfigurationArn=
\"\"}
```

De-authorizing connections in EventBridge

You can de-authorize a connection. De-authorizing a connection removes all authorization parameters. Removing authorization parameters removes the secret from the connection, so you can reuse it without having to create a new connection.

> **Note**
>
> You must update any API destinations or Step Functions workflow tasks that use the de-authorized connection to use a different connection to successfully send requests to the HTTPS endpoint.

In addition to manual de-authorization, EventBridge de-authorizes a connection if the following occurs:

- EventBridge cannot retrieve the secret because it has been deleted.
- For connections using OAuth authentication, EventBridge cannot refresh the necessary OAuth token.
- For connections using Basic or API Key authentication, EventBridge receives a `401 (UnAuthorize)` or `407 (Proxy_Authentication_required)` error code.

  In this case, update the connection with appropriate credentials to resolve the error.
- If you have specified a customer managed key for EventBridge to use for secret encryption, and EventBridge receives a KMS key error when encrypting or decrypting the connection's secret. For more information, see Customer managed key errors.

> **Note**
>
> EventBridge emits events when the state of a connection changes. You can use these events to track changes in a connection.
> For more information, see Connection events.

To de-authorize a connection using the EventBridge console

1. Log in to AWS using an account that has permissions to manage EventBridge and open the EventBridge console.
2. In the left navigation pane, under **Integration**, choose **Connections**.
3. In the **Connections** table, choose the connection.
4. On the **Connection details** page, choose **De-authorize**.
5. In the **Deauthorize connection?** dialog box, enter the name of the connection, and then choose **De-authorize**.

   The status of the connection changes to **De-authorizing** until the process is complete. Then the status changes to **De-authorized**. Now you can edit the connection to add new authorization parameters.

To de-authorize a connection using the AWS CLI

- Use the `deauthorize-connection` command.

Deleting connections in EventBridge

You can delete a connection if you are no longer using it.

To delete a connection using the EventBridge console

1. Log in to AWS using an account that has permissions to manage EventBridge and open the EventBridge console.
2. In the left navigation pane, under **Integration**, choose **Connections**.
3. In the **Connections** table, choose the connection.
4. On the **Connection details** page, choose **Delete**.

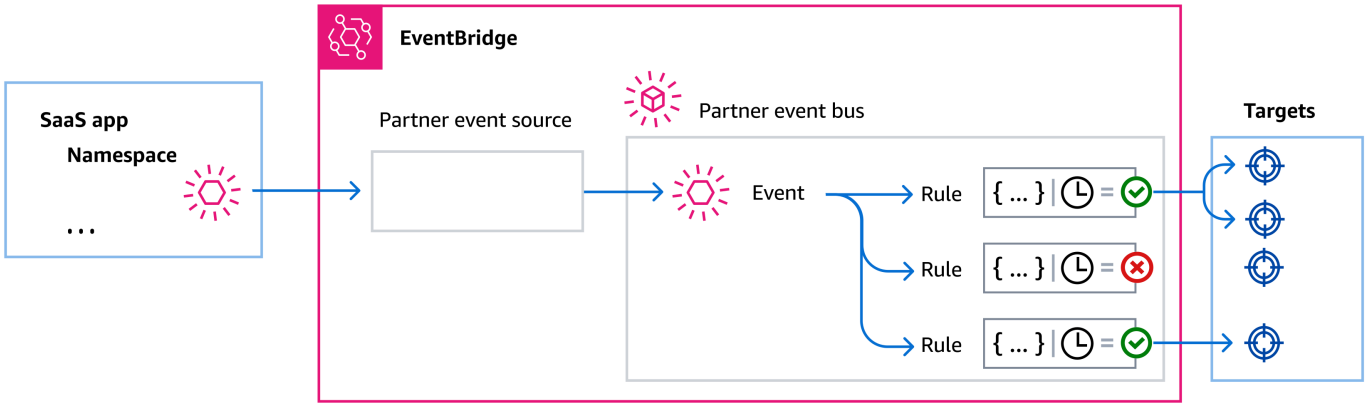
To delete a connection using the AWS CLI

- Use the `delete-connection` command.

Receiving events from a SaaS partner with Amazon EventBridge

To receive events from SaaS partner applications and services, you need a *partner event source* from the partner. A partner event source is a resource created by a partner that you can then accept as an event source. To accept the partner event source, you create a Custom Event Bus - Classic and match it to the partner event source.

> **Note**
>
> To receive partner events on a Custom Event Bus, create an event source that names the partner event source as its origin; the partner integration on this page stays the same. See the section called “Event sources”.



Amazon EventBridge: Software as a service (SaaS) partners

Topics

- Supported SaaS partner integrations
- Configuring Amazon EventBridge to receive events from a SaaS integration
- Receiving SaaS events from AWS Lambda function URLs in Amazon EventBridge
- Receiving events from Salesforce in Amazon EventBridge

Supported SaaS partner integrations

EventBridge supports the following SaaS partner integrations:

- Adobe
- Salesforce (via Amazon AppFlow)
- Apptrail
- Atlan
- Auth0
- Authress
- Benchling
- BigCommerce
- Blitline
- Buildkite
- Chargebee
- Checkout.com
- CleverTap
- CloudAMQP
- commercetools
- Datadog
- EnergySys
- Epsagon
- Freshworks
- Genesys
- Gladly
- GS2
- Guidewire
- Hitachi Solutions
- iLert
- Atlassian - Jira Service Management
- Karte
- Kloudless
- Mackerel
- MongoDB

- New Relic
- nOps
- Okta
- OneLogin
- Operata
- Opsgenie
- PagerDuty
- Payshield
- Rhythm Software
- Rightsline
- Rootly
- SaaSus Platform
- SailPoint
- Scalr
- Segment
- Shopify
- SignalFx
- Site24x7
- SnowcatCloud
- Snyk
- Stax
- Stripe
- SugarCRM
- Symantec
- Tealium
- Thundra
- TriggerMesh
- Whispir
- Zendesk

- Amazon Seller Partner API

Configuring Amazon EventBridge to receive events from a SaaS integration

Configuring EventBridge to receive partner events consists of two main steps:

- Creating the partner event source
- Associating that partner source with a partner event bus

> **Note**
>
> Any events published by a partner to a partner event source that has not been associated with an event bus will be immediately dropped. Those events will not be persisted at rest in EventBridge.

Create a partner event source (console only)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Partner event sources**.
3. Find the partner that you want and then choose **Set up** for that partner.
4. To copy your account ID to the clipboard, choose **Copy**.
5. In the navigation pane, choose **Partner event sources**.
6. Go to the partner's website and follow the instructions to create a partner event source using your account ID. The event source that you create is available to only your account.

Associate the partner source with a partner event bus (console)

1. In the EventBridge console, choose **Partner event sources** in the navigation pane.
2. Select the button next to the partner event source and then choose **Associate with event bus**.

   The status of the event source changes from `Pending` to `Active`, and the name of the event bus updates to match the partner event source name. You can now start creating rules that match events from the partner event source.

Associate the partner source with a partner event bus (AWS CLI)

- Use `create-event-bus` to create a partner event bus associated with the partner event source.

  Both `name` and `event-source-name` should be set to the partner event source name.

  For example:

```
aws events create-event-bus \
    --name "aws.partner/saas-integration/name" \
    --event-source-name "aws.partner/saas-integration/name" \
    --region us-east-1
```

  After EventBridge creates the event bus, you can call `describe-event-source` to return details about the partner source. The `State` of the partner source should be `ACTIVE`.

```
aws events describe-event-source
--name "aws.partner/saas-integration/name"
```

> **Note**
>
> Calling `put-permission` on a partner event bus returns an error. Only the partner account of the event source associated with the partner event bus is permitted to send events to it.

Associate the partner source with a partner event bus (CloudFormation)

1. Create a CloudFormation template that provisions an `AWS::Events::EventBus` resource with the partner event source.

   Both `Name` and `EventSourceName` should be set to the partner event source name. For example:

```
AWSTemplateFormatVersion: 2010-09-09

Description:
   Cloudformation template to create Event Bus for receiving partner events
```

```
Resources:
  ExamplePartnerEventBus:
    Type: AWS::Events::EventBus
    Properties:
      EventSourceName: 'aws.partner/saas-integration/name'
      Name: 'aws.partner/saas-integration/name'
```

2. Use `cloudformation create-stack` or the CloudFormation console to create a stack from the template. For example:

```
aws cloudformation create-stack --stack-name eventbridge-saas --template-
body file://template.yml --region us-east-1
```

> **Note**
>
> Including an `AWS::Events::EventBusPolicy` resource for the partner event bus in your template will result in an error. Only the partner account of the event source associated with the partner event bus is permitted to send events to it.

Receiving SaaS events from AWS Lambda function URLs in Amazon EventBridge

> **Note**
>
> In order for the Inbound Webhook to be accessible by our partners, we're creating an Open Lambda in your AWS account that is secured at the Lambda application level by verifying the authentication signature sent by the third-party partner. Please review this configuration with your security team. For more information, see Security and auth model for Lambda function URLs.

Your Amazon EventBridge event bus can use an AWS Lambda function URL created by an CloudFormation template to receive events from supported SaaS providers. With function URLs, the event data is sent to a Lambda function. The function then converts this data into an event that can be ingested by EventBridge and sent to an event bus for processing. Once the event is on an event bus, you can use rules to filter the events, apply any configured input transformations, and then route it to the correct target.

> **Note**
>
> Creating Lambda function URLs will increase your monthly costs. For more information, see AWS Lambda pricing.

To set up a connection to EventBridge, you first select the SaaS provider that you want to set up a connection with. Then, you provide a *signing secret* that you've created with that provider, and select the EventBridge event bus to send events to. Finally, you use an CloudFormation template and create the needed resources to complete the connection.

The following SaaS providers are currently available for use with EventBridge using Lambda function URLs:

- GitHub
- Twilio

Topics

- Step 1: Create the CloudFormation stack
- Step 2: Create a GitHub webhook
- Set up a connection to a Twilio
- Update webhook secret or auth token
- Update Lambda function
- Available event types
- Quotas, error codes, and retrying delivery

Step 1: Create the CloudFormation stack

First, use the Amazon EventBridge console to create a CloudFormation stack:

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. From the navigation pane, choose **Quick starts**.
3. Under **Inbound webhooks using Lambda fURLs**, choose **Get started**.
4. Under **GitHub**, choose **Set up**.
5. Under **Step 1: Select an event bus**, select an event bus from the dropdown list. This event bus receives data from the Lambda function URL that you provide to GitHub. You can also create an event bus by selecting **New event bus**.
6. Under **Step 2: Set up using CloudFormation**, choose **New GitHub webhook**.
7. Select **I acknowledge that the Inbound Webhook I create will be publicly accessible.** and choose **Confirm**.
8. Enter a name for the stack.
9. Under parameters, verify that the correct event bus is listed, then specify a secure token for the **GitHubWebhookSecret**. For more information on creating a secure token, see Setting your secret token in the GitHub documentation.
10. Under **Capabilities and transforms**, select each of the following:

    - **I acknowledge that CloudFormation might create IAM resources.**
    - **I acknowledge that CloudFormation might create IAM resources with custom names.**
    - **I acknowledge that CloudFormation might require the following capability: `CAPABILITY_AUTO_EXPAND`**

11. Choose **Create stack**.

Step 2: Create a GitHub webhook

Next, create the webhook on GitHub. You'll need both the secure token and the Lambda function URL you created in step 2 to complete this step. For more information, see Creating webhooks in the GitHub documentation.

Set up a connection to a Twilio

Step 1: Find your Twilio auth token

To set up a connection between Twilio and EventBridge, first set up the connection to Twilio with the auth token, or secret, for your Twilio account. For more information, see Auth Tokens and How To Change Them in the Twilio documentation.

Step 2: Create the CloudFormation stack

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Quick starts**.
3. Under **Inbound webhooks using Lambda fURLs**, choose **Get started**.
4. Under **Twilio**, choose **Set up**.
5. Under **Step 1: Select and event bus**, sselect an event bus from the dropdown list. This event bus receives data from the Lambda function URL that you provide to Twilio. You can also create an event bus by selecting **New event bus**.
6. Under **Step 2: Set up using CloudFormation**, choose **New Twilio webhook**.
7. Select **I acknowledge that the Inbound Webhook I create will be publicly accessible.** and choose **Confirm**.
8. Enter a name for the stack.
9. Under parameters, verify that the correct event bus is listed, then enter the **TwilioWebhookSecret** that you created in Step 1.
10. Under **Capabilities and transforms**, select each of the following:

    - **I acknowledge that CloudFormation might create IAM resources.**
    - **I acknowledge that CloudFormation might create IAM resources with custom names.**

- **I acknowledge that CloudFormation might require the following capability: CAPABILITY_AUTO_EXPAND**

11. Choose **Create stack**.

Step 3: Create a Twilio webhook

After you set up the Lambda function URL, you need to give it to Twilio so that event data can be sent. For more information, see Configure your public URL with Twilio in the Twilio documentation.

Update webhook secret or auth token

Update GitHub secret

> **Note**
>
> GitHub doesn't support having two secrets at the same time. You may experience resource downtime while the GitHub secret and the secret in the CloudFormation stack are out of sync. GitHub messages sent while the secrets are out of sync will fail becaue of incorrect signatures. Wait until the GitHub and CloudFormation secrets are in sync, then try again.

1. Create a new GitHub secret. For more information, see Encrypted secrets in the GitHub documentation.
2. Open the CloudFormation console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudformation.
3. From the navigation pane, choose **Stacks**.
4. Choose the stack for the webhook that includes the secret you want to update.
5. Choose **Update**.
6. Make sure **Use current template** is selected and choose **Next**.
7. Under **GitHubWebhookSecret**, clear **Use existing value**, enter the new GitHub secret you created in step 1, and choose **Next**.
8. Choose **Next**.
9. Choose **Update stack**.

It may take up to one hour for the secret to propagate. To reduce this downtime, you can refresh the Lambda execution context.

Update Twilio secret

> **Note**
>
> Twilio doesn't support having two secrets at the same time. You may experience resource downtime while the Twilio secret and the secret in the CloudFormation stack are out of sync. Twilio messages sent while the secrets are out of sync will fail because of incorrect signatures. Wait until the Twilio and CloudFormation secrets are in sync, then try again.

1. Create a new Twilio secret. For more information, see Auth Tokens and How To Change Them in the Twilio documentation.
2. Open the CloudFormation console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudformation.
3. From the navigation pane, choose **Stacks**.
4. Choose the stack for the webhook that includes the secret you want to update.
5. Choose **Update**.
6. Make sure **Use current template** is selected and choose **Next**.
7. Under **TwilioWebhookSecret**, clear **Use existing value**, enter the new Twilio secret you created in step 1, and choose **Next**.
8. Choose **Next**.
9. Choose **Update stack**.

It may take up to one hour for the secret to propagate. To reduce this downtime, you can refresh the Lambda execution context.

Update Lambda function

The Lambda function that's created by the CloudFormation stack creates the basic webhook. If you want to customize the Lambda function for a specific use case, such as customized logging, use the CloudFormation console to access the function and then use the Lambda console to update the Lambda function code.

Access the Lambda function

1. Open the CloudFormation console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudformation.

2. From the navigation pane, choose **Stacks**.
3. Choose the stack for the webhook that includes the Lambda function you want to update.
4. Choose **Resources** tab.
5. To open the Lambda function in the Lambda console, under **Physical ID**, choose the ID of the Lambda function.

Now that you've accessed the Lambda function, use the Lambda console to update the function code.

Update the Lambda function code

1. Under **Actions**, choose **Export function**.
2. Choose **Download deployment package** and save the file to your computer.
3. Unzip the deployment package .zip file, update the `app.py` file, and zip the updated deployment package, making sure all the files in the original .zip file are included.
4. In the Lambda console, choose the **Code** tab.
5. Under **Code source**, choose **Upload from**.
6. Choose **.zip file**, and then choose **Upload**.
   - In the file chooser, select the file you updated, choose **Open**, and then choose **Save**.
7. Under **Actions**, choose **Publish new version**.

Available event types

The following event types are currently supported by CloudFormation event buses:

- **GitHub** – All event types are supported.
- **Twilio** – Post-event webhooks are supported.

Quotas, error codes, and retrying delivery

Quotas

The number of incoming requests to the webhook is capped by the underlying AWS services. The following table includes the relevant quotas.

Service	Quota
AWS Lambda	Default: 10 concurrent executions For more information about quotas, including requestin g quota increases, see Lambda quotas.
AWS Secrets Manager	Default: 5,000 requests per second For more information about quotas, including requestin g quota increases, see AWS Secrets Manager quotas. **Note** The number of requests per second is minimized using the AWS Secrets Manager Python caching client.
Amazon EventBridge	1 MB maximum entry size for PutEvents actions. EventBridge enforces Region-based rate quotas. For more information, see ???.

Error codes

Each AWS service returns specific error codes when errors occur. The following table includes the relevant error codes.

Service	Error code	Description
AWS Lambda	429 “TooManyRequestsEx ption”	The concurrent execution quota is exceeded.
AWS Secrets Manager	500 “Internal Server Error”	The requests per second quota is exceeded.
Amazon EventBridge	500 “Internal Server Error”	The rate quota is exceeded for the Region.

Event redelivery

When errors happen you can retry delivery of the affected events. Each SaaS provider has different retry procedures.

GitHub

Use the GitHub webhooks API to check the deliver status of any webhook call and redeliver the event, if needed. For more information, see the following GitHub documentation:

- **Organization** – Redeliver a delivery for an organization webhook
- **Repository** – Redeliver a delivery for a repository webhook
- **App** – Redeliver a delivery for an app webhook

Twilio

Twilio users can customize event retry options using connection overrides. For more information, see Webhooks (HTTP callbacks): Connection Overrides in the Twilio documentation.

Receiving events from Salesforce in Amazon EventBridge

You can use Amazon EventBridge to receive events from Salesforce in following ways:

- By using Salesforce's Event Bus Relay feature to receive events directly on an EventBridge partner event bus.
- By configuring a flow in Amazon AppFlow that uses Salesforce as a data source. Amazon AppFlow then sends Salesforce events to EventBridge by using a partner event bus.

You can send event information to Salesforce using API destinations. Once the event is sent to Salesforce, it can be processed by Flows or Apex triggers. For more information about setting up a Salesforce API destination, see ???.

Topics

- Receiving events from Salesforce using Event Bus Relay
- Receiving events from Salesforce using Amazon AppFlow

Receiving events from Salesforce using Event Bus Relay

Step 1: Set up Salesforce Event Bus Relay and an EventBridge partner event source

When you create an event relay configuration on Salesforce, Salesforce creates a partner event source in EventBridge in the pending state.

To configure Salesforce Event Bus Relay

1. Set Up a REST API Tool
2. (Optional) Define a Platform Event
3. Create a Channel for a Custom Platform Event
4. Create a Channel Member to Associate the Custom Platform Event
5. Create a Named Credential
6. Create an Event Relay Configuration

Step 2: Activate Salesforce partner event source in the EventBridge console and start the event relay

1. Open the Partner event sources page in the EventBridge console.
2. Select the Salesforce partner event source that you created in Step 1.
3. Choose **Associate with event bus**.
4. Validate the name of the partner event bus.
5. Choose **Associate**.
6. Start the Event Relay

Now that you've set up and started the Event Bus Relay and configured the partner event source you can create an EventBridge rule that reacts to events to filter and send the data to a target.

Receiving events from Salesforce using Amazon AppFlow

Amazon AppFlow encapsulates events from Salesforce in an EventBridge event envelope. The following example shows a Salesforce event received by an EventBridge partner event bus.

```
{
    "version": "0",
    "id": "5c42b99e-e005-43b3-c744-07990c50d2cc",
    "detail-type": "AccountChangeEvent",
    "source": "aws.partner/appflow.test/salesforce.com/364228160620/CustomSF-Source-
Final",
    "account": "000000000",
    "time": "2020-08-20T18:25:51Z",
    "region": "us-west-2",
    "resources": [],
    "detail": {
        "ChangeEventHeader": {
            "commitNumber": 248197218874,
            "commitUser": "0056g000003XW7AAAW",
            "sequenceNumber": 1,
            "entityName": "Account",
            "changeType": "UPDATE",
            "changedFields": [
                "LastModifiedDate",
                "Region__c"
            ],
```

```
            "changeOrigin": "com/salesforce/api/soap/49.0;client=SfdcInternalAPI/",
            "transactionKey": "000035af-b239-0581-9f14-461e4187de11",
            "commitTimestamp": 1597947935000,
            "recordIds": [
                "0016g00000MLhLeAAL"
            ]
        },
        "LastModifiedDate": "2020-08-20T18:25:35.000Z",
        "Region__c": "America"
    }
}
```

Step 1: Configure Amazon AppFlow to use Salesforce as a partner event source

To send events to EventBridge, you first need to configure Amazon AppFlow to use Salesforce as a partner event source.

1. In the Amazon AppFlow console, choose **Create flow**.
2. In the **Flow details** section, in **Flow name** enter a name for your flow.
3. (Optional) Enter a description for the flow and then choose **Next**.
4. Under **Source details**, choose *Salesforce* from the **Source name** drop-down, and then choose **Connect** to create a new connection.
5. In the **Connect to Salesforce** dialog box, choose either **Production** or **Sandbox** for the Salesforce environment.
6. In the **Connection name** field, enter a unique name for the connection, and then choose **Continue**.
7. In the Salesforce dialog box, do the following:
   a. Enter your Salesforce sign-in credentials to log in to Salesforce.
   b. Select Salesforce events for the types of data for Amazon AppFlow to process.
8. In the **Choose Salesforce event** drop-down, select the type of event to send to EventBridge.
9. For a destination, select **Amazon EventBridge**.
10. Select **Create new partner event source**.
11. (Optional) Specify a unique suffix for the partner event source.
12. Choose **Generate partner event source**.
13. Choose an Amazon S3 bucket to store event payload files that are larger than 1 MB.

14. In the **Flow trigger** section, ensure that **Run flow on event** is selected. This setting ensures that the flow is executed when a new Salesforce event occurs.
15. Choose **Next**.
16. For field mapping, select **Map all fields directly**. Alternatively, you can select the fields that are of interest from the **Source field name** list.

    For more information about field mapping, see Map data fields.
17. Choose **Next**.
18. (Optional) Configure filters for data fields in Amazon AppFlow.
19. Choose **Next**.
20. Review the settings and then choose **Create flow**.

With the flow configured, Amazon AppFlow creates a new partner event source that you then need to associate with a partner event bus in your account.

Step 2: Configure EventBridge to receive Salesforce events

Ensure that the Amazon AppFlow flow that is triggered from Salesforce events with EventBridge as a destination is configured before following instructions in this section.

To configure EventBridge to receive Salesforce events

1. Open the Partner event sources page in the EventBridge console.
2. Select the Salesforce partner event source that you created in Step 1.
3. Choose **Associate with event bus**.
4. Validate the name of the partner event bus.
5. Choose **Associate**.
6. In the Amazon AppFlow console, open the flow you created and choose **Activate flow**.
7. Open the Rules page in the EventBridge console.
8. Choose **Create rule**.
9. Enter a unique name for the rule.
10. Choose **Event pattern** in the **Define pattern** section.
11. For **Event matching pattern**, select **Pre-defined pattern by service**.
12. For **Service provider** section, select **All Events**.

13. For **Select event bus**, choose **Custom or partner event bus**.
14. Select the event bus that you associated with the Amazon AppFlow partner event source.
15. For **Select targets**, choose the AWS service that is to act when the rule runs. One rule can have up to five targets.
16. Choose **Create**.

The target service receives all Salesforce events configured for your account. To filter the events or send some events to different targets, you can use content-based filtering with event patterns.

> **Note**
>
> For events larger than 1 MB, Amazon AppFlow doesn't send the full event to EventBridge. Instead, Amazon AppFlow puts the event into an S3 bucket in your account, and then sends an event to EventBridge with a pointer to the Amazon S3 bucket. You can use the pointer to get the full event from the bucket.

Amazon EventBridge event patterns

Chances are you won't want to process every single event that gets delivered to a given event bus or pipe. Rather, you'll likely want to select a subset of all the events delivered, based on the source of the event, the event type, and/or attributes of those events.

To specify which events to send to a target, you create an *event pattern*. An event pattern defines the data EventBridge uses to determine whether to send the event to the target. If the event pattern matches the event, EventBridge sends the event to the target. Event patterns have the same structure as the events they match. An event pattern either matches an event or it doesn't.

> **Tip**
>
> You can use the EventBridge Sandbox to test event patterns against sample events before creating or updating rules. For more information, see ???.

For example, consider the following event from Amazon EC2:

```
{
  "version": "0",
  "id": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "us-west-1",
  "resources": [
    "arn:aws:ec2:us-west-1:123456789012:instance/i-1234567890abcdef0"
  ],
  "detail": {
    "instance-id": "i-1234567890abcdef0",
    "state": "terminated"
  }
}
```

The following event pattern selects all Amazon EC2 `instance-termination` events. The event pattern does this by specifying three requirements to match an event:

1. The event source must be Amazon EC2.

2. The event must be an Amazon EC2 state-change notification.
3. The state of the Amazon EC2 instance must be `terminated`.

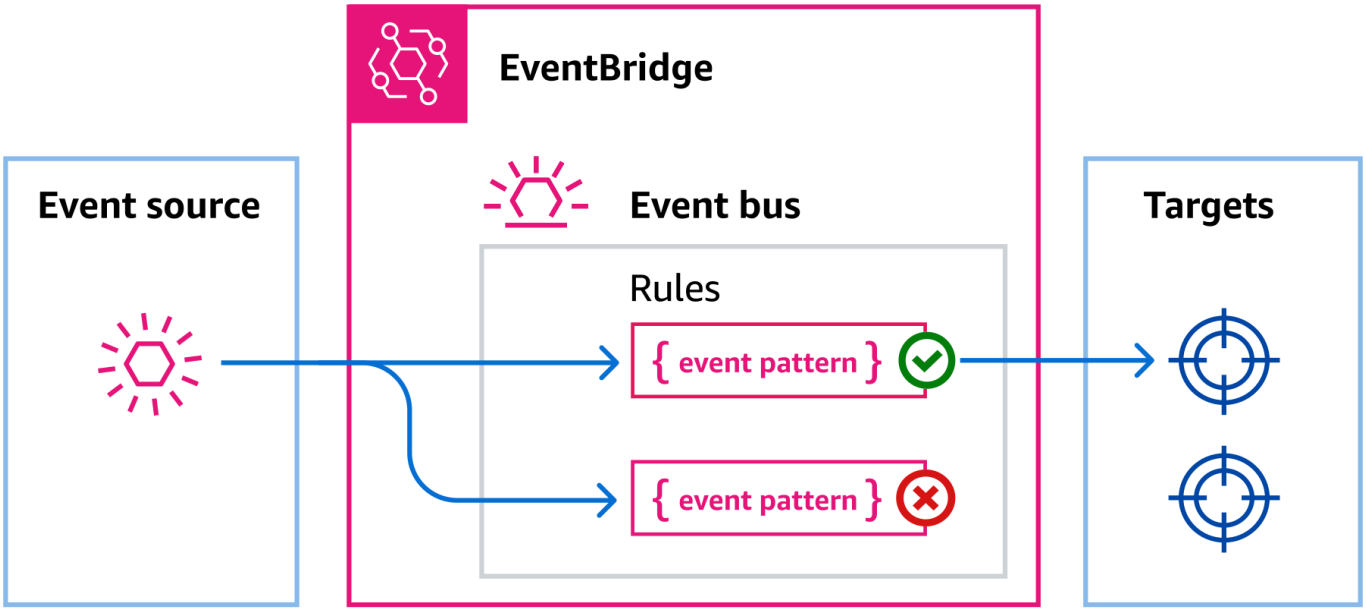
```
{
  "source": ["aws.ec2"],
  "detail-type": ["EC2 Instance State-change Notification"],
  "detail": {
    "state": ["terminated"]
  }
}
```

Note that in this example, the event pattern includes fields *about* the event--`source` and `detail-type`--as well as a field from the event body--`state`.

> **Important**
>
> In EventBridge, it is possible to create rules that can lead to higher-than-expected charges and throttling. For example, you can inadvertently create a rule that leads to an infinite loop, where a rule is fired recursively without end. Suppose you created a rule to detect that ACLs have changed on an Amazon S3 bucket, and trigger software to change them to the desired state. If the rule is not written carefully, the subsequent change to the ACLs fires the rule again, creating an infinite loop.
> For guidance on how to write precise rules and event patterns to minimize such unexpected results, see Best practices for rules and Best practices.

Event patterns for event buses

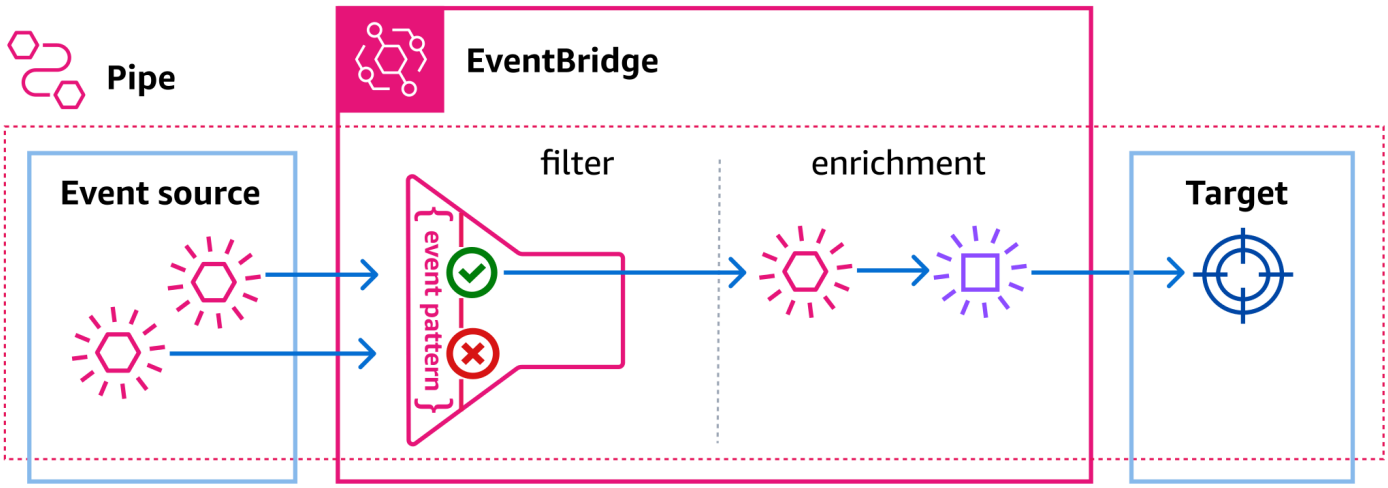
For event buses, you can specify an event pattern for each rule you create for the bus. In this way, you can select which events to send to specific targets. Event patterns for event buses can match against the event source, event metadata, and/or event detail values.



[Amazon EventBridge: How to filter events]

Event patterns for EventBridge Pipes

For EventBridge Pipes, you can specify event patterns to filter the events from the pipe source that you want delivered to the pipe target. Since each pipe has a single event source, event patterns for pipes can match against event metadata and/or detail values.



Not all event fields can be used to construct pipe event patterns. For more information, see Filtering.

Event pattern syntax

To create an event pattern, you specify the fields of an event that you want the event pattern to match. Only specify the fields that you use for matching.

For example, the following event pattern example only provides values for three fields: the top-level fields `"source"` and `"detail-type"`, and the `"state"` field inside the `"detail"` object field. EventBridge ignores all the other fields in the event when applying the rule.

```
{
  "source": ["aws.ec2"],
  "detail-type": ["EC2 Instance State-change Notification"],
  "detail": {
    "state": ["terminated"]
  }
}
```

For an event pattern to match an event, the event must contain all the field names listed in the event pattern. The field names must also appear in the event with the same nesting structure.

When you write event patterns to match events, you can use the `TestEventPattern` API or the `test-event-pattern` CLI command to test that your pattern matches the correct events. For more information, see TestEventPattern.

Matching event values

In an event pattern, the value to match is in a JSON array, surrounded by square brackets ("[", "]") so that you can provide multiple values. For example, to match events from Amazon EC2 or AWS Fargate, you could use the following pattern, which matches events where the value for the `"source"` field is either `"aws.ec2"` or `"aws.fargate"`.

```
{
    "source": ["aws.ec2", "aws.fargate"]
}
```

For more information, see Matching on multiple field values.

Using comparison operators in Amazon EventBridge event patterns

Amazon EventBridge supports declarative content filtering using event patterns. With content filtering, you can write complex event patterns that only match events under very specific conditions. For example, you can create an event pattern that matches an event when:

- A field of the event is within a specific numeric range.
- The event comes from a specific IP address.
- A specific field doesn't exist in the event JSON.

For more information, see Comparison operators.

Considerations when creating event patterns

Below are some things to consider when constructing your event patterns:

- EventBridge ignores the fields in the event that aren't included in the event pattern. The effect is that there is a `"*": "*"` wildcard for fields that don't appear in the event pattern.
- The values that event patterns match follow JSON rules. You can include strings enclosed in quotation marks ("), numbers, and the keywords `true`, `false`, and `null`.
- For strings, EventBridge uses exact character-by-character matching without case-folding or any other string normalization.
- For numbers, EventBridge uses string representation. For example, 300, 300.0, and 3.0e2 are not considered equal.
- If multiple patterns are specified for the same JSON field, EventBridge only uses the last one.
- Be aware that when EventBridge compiles event patterns for use, it uses dot (.) as the joining character.

  This means EventBridge will treat the following event patterns as identical:

```
## has no dots in keys
{ "detail" : { "state": { "status": [ "running" ] } } }

## has dots in keys
{ "detail" : { "state.status": [ "running" ] } }
```

  And that both event patterns will match the following two events:

```
## has no dots in keys
{ "detail" : { "state": { "status": "running" } } }

## has dots in keys
{ "detail" : { "state.status": "running"  } }
```

> **Note**
>
> This describes current EventBridge behavior, and should not be relied on to not change.

- Event patterns containing duplicate fields are invalid. If a pattern contains duplicate fields, EventBridge only considers the final field value.

  For example, the following event patterns will match the same event:

```
## has duplicate keys
{
  "source": ["aws.s3"],
  "source": ["aws.sns"],
  "detail-type": ["AWS API Call via CloudTrail"],
  "detail":  {
      "eventSource": ["s3.amazonaws.com"],
      "eventSource": ["sns.amazonaws.com"]
  }
}

## has unique keys
{
  "source": ["aws.sns"],
  "detail-type": ["AWS API Call via CloudTrail"],
  "detail": { "eventSource": ["sns.amazonaws.com"] }
}
```

  And EventBridge treats the following two events as identical:

```
## has duplicate keys
{
  "source": ["aws.s3"],
  "source": ["aws.sns"],
  "detail-type": ["AWS API Call via CloudTrail"],
```

```
  "detail":  [
    {
      "eventSource": ["s3.amazonaws.com"],
      "eventSource": ["sns.amazonaws.com"]
    }
  ]
}

## has unique keys
{
  "source": ["aws.sns"],
  "detail-type": ["AWS API Call via CloudTrail"],
  "detail": [
    { "eventSource": ["sns.amazonaws.com"] }
  ]
}
```

Note

This describes current EventBridge behavior, and should not be relied on to not change.

Matching events on event field values

You can use all of the JSON data types and values to match events. The following examples show events and the event patterns that match them.

Field matching

You can match on the value of a field. Consider the following Amazon EC2 Auto Scaling event.

```
{
  "version": "0",
  "id": "3e3c153a-8339-4e30-8c35-687ebef853fe",
  "detail-type": "EC2 Instance Launch Successful",
  "source": "aws.autoscaling",
  "account": "123456789012",
  "time": "2015-11-11T21:31:47Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
```

```
    "eventVersion": "",
    "responseElements": null
  }
}
```

For the preceding event, you can use the `"responseElements"` field to match.

```
{
  "source": ["aws.autoscaling"],
  "detail-type": ["EC2 Instance Launch Successful"],
  "detail": {
   "responseElements": [null]
  }
}
```

Value matching

Consider the following Amazon Macie event, which is truncated.

```
{
  "version": "0",
  "id": "0948ba87-d3b8-c6d4-f2da-732a1example",
  "detail-type": "Macie Finding",
  "source": "aws.macie",
  "account": "123456789012",
  "time": "2021-04-29T23:12:15Z",
  "region":"us-east-1",
  "resources": [

  ],
  "detail": {
    "schemaVersion": "1.0",
    "id": "64b917aa-3843-014c-91d8-937ffexample",
    "accountId": "123456789012",
    "partition": "aws",
    "region": "us-east-1",
    "type": "Policy:IAMUser/S3BucketEncryptionDisabled",
    "title": "Encryption is disabled for the S3 bucket",
    "description": "Encryption is disabled for the Amazon S3 bucket. The data in the
 bucket isn’t encrypted
        using server-side encryption.",
    "severity": {
```

```
        "score": 1,
        "description": "Low"
    },
    "createdAt": "2021-04-29T15:46:02Z",
    "updatedAt": "2021-04-29T23:12:15Z",
    "count": 2,
.
.
.
```

The following event pattern matches any event that has a severity score of 1 and a count of 2.

```
{
  "source": ["aws.macie"],
  "detail-type": ["Macie Finding"],
  "detail": {
    "severity": {
      "score": [1]
    },
    "count":[2]
  }
}
```

Matching events on null values and empty strings in Amazon EventBridge

Important

In EventBridge, it is possible to create rules that can lead to higher-than-expected charges and throttling. For example, you can inadvertently create a rule that leads to an infinite loop, where a rule is fired recursively without end. Suppose you created a rule to detect that ACLs have changed on an Amazon S3 bucket, and trigger software to change them to the desired state. If the rule is not written carefully, the subsequent change to the ACLs fires the rule again, creating an infinite loop.
For guidance on how to write precise rules and event patterns to minimize such unexpected results, see Best practices for rules and Best practices.

You can create an event pattern that matches a field in an event that has a null value or is an empty string. Consider the following example event.

See best practices to avoid higher than expected charges and throttling

```
{
  "version": "0",
  "id": "3e3c153a-8339-4e30-8c35-687ebef853fe",
  "detail-type": "EC2 Instance Launch Successful",
  "source": "aws.autoscaling",
  "account": "123456789012",
  "time": "2015-11-11T21:31:47Z",
  "region": "us-east-1",
  "resources": [
   ],
  "detail": {
    "eventVersion": "",
    "responseElements": null
   }
}
```

To match events where the value of `eventVersion` is an empty string, use the following event pattern, which matches the preceding event.

```
{
  "detail": {
     "eventVersion": [""]
  }
}
```

To match events where the value of `responseElements` is null, use the following event pattern, which matches the preceding event.

```
{
  "detail": {
     "responseElements": [null]
  }
}
```

> **Note**
>
> Null values and empty strings are not interchangeable in pattern matching. An event pattern that matches empty strings doesn't match values of `null`.

Using null values in AWS CloudFormation templates

AWS CloudFormation does not allow `null` values in templates. If you define an event pattern with a null value using YAML or JSON object syntax, the template validation fails with the error: `'null' values are not allowed in templates`.

To work around this limitation, specify the `EventPattern` property as a JSON string instead of a YAML or JSON object. The following example shows how to match on null values in a AWS CloudFormation template:

```
MyRule:
  Type: AWS::Events::Rule
  Properties:
    EventPattern: '{"detail":{"responseElements":[null]}}'
```

Matching on multiple values for an event field in Amazon EventBridge

The value of each field in an event pattern is an array containing one or more values. An event pattern matches the event if any of the values in the array match the value in the event. If the value in the event is an array, then the event pattern matches if the intersection of the event pattern array and the event array is non-empty.

> **Important**
>
> In EventBridge, it is possible to create rules that can lead to higher-than-expected charges and throttling. For example, you can inadvertently create a rule that leads to an infinite loop, where a rule is fired recursively without end. Suppose you created a rule to detect that ACLs have changed on an Amazon S3 bucket, and trigger software to change them to the desired state. If the rule is not written carefully, the subsequent change to the ACLs fires the rule again, creating an infinite loop.
>
> For guidance on how to write precise rules and event patterns to minimize such unexpected results, see Best practices for rules and Best practices.

For example, consider an event pattern that includes the following field.

```
"resources": [
   "arn:aws:ec2:us-east-1:123456789012:instance/i-b188560f",
   "arn:aws:ec2:us-east-1:111122223333:instance/i-b188560f",
   "arn:aws:ec2:us-east-1:444455556666:instance/i-b188560f",
]
```

The preceding event pattern matches an event that includes the following field because the first item in the event pattern array matches the second item in the event array.

```
"resources": [
   "arn:aws:autoscaling:us-east-1:123456789012:autoScalingGroup:eb56d16b-bbf0-401d-
b893-d5978ed4a025:autoScalingGroupName/ASGTerminate",
   "arn:aws:ec2:us-east-1:123456789012:instance/i-b188560f"
]
```

Comparison operators for use in event patterns in Amazon EventBridge

Below a summary of all the comparison operators available in EventBridge.

Comparison operators only work on leaf nodes, with the exception of `$or` and `anything-but`.

Comparison	Example	Rule syntax	Event bus support	Pipe support
And	Location is "New York" and Day is "Monday"	`"Location": [ "New York" ], "Day": ["Monday"]`	Yes	Yes
Anything-but	State is any value besides "initializing".	`"state": [ { "anything-but": "initializing" } ]`	Yes	Yes
Anything-but (begins with)	Region is not in the US.	`"Region": [ { "anything-but": {"prefix": "us-" } } ]`	Yes	No
Anything-but (ends with)	FileName does not end with a .png extension.	`"FileName": [ { "anything-but": { "suffix": ".png" } } ]`	Yes	No
Anything-but (ignore case)	State is any value besides "initializing" or any other casing variation, such as "INITIALIZING".	`"state": : [{ "anything-but": { "equals-ignore-case": "initializing" }}]}`	Yes	No
Anything-but using a wildcard	FileName is not a file path that includes `/lib/`.	`"FilePath" : [{ "anything-but":`	Yes	No

Comparison	Example	Rule syntax	Event bus support	Pipe support
		`{ "wildcard": "*/lib/*" }}]`		
Begins with	Region is in the US.	`"Region": [ {"prefix": "us-" } ]`	Yes	Yes
Begins with (ignore case)	Service name starts with the letters "eventb", regardless of case.	`{"service" : [{ "prefix": { "equals-ignore-case": "eventb" }}]}`	Yes	Yes
Empty	LastName is empty.	`"LastName": [""]`	Yes	Yes
Equals	Name is "Alice"	`"Name": [ "Alice" ]`	Yes	Yes
Equals (ignore case)	Name is "Alice"	`"Name": [ { "equals-ignore-case": "alice" } ]`	Yes	Yes
Ends with	FileName ends with a .png extension	`"FileName": [ { "suffix": ".png" } ]`	Yes	Yes
Ends with (ignore case)	Service name ends with the letters "tbridge", or any other casing variation, such as "TBRIDGE".	`{"service" : [{ "suffix": { "equals-ignore-case": "tBridge" }}]}`	Yes	Yes
Exists	ProductName exists	`"ProductName": [ { "exists": true } ]`	Yes	Yes

Comparison	Example	Rule syntax	Event bus support	Pipe support
Does not exist	ProductName does not exist	`"ProductName": [ { "exists": false } ]`	Yes	Yes
Not	Weather is anything but "Raining"	`"Weather": [ { "anything-but": [ "Raining" ] } ]`	Yes	Yes
Null	UserID is null	`"UserID": [ null ]`	Yes	Yes
Numeric (equals)	Price is 100	`"Price": [ { "numeric": [ "=", 100 ] } ]`	Yes	Yes
Numeric (range)	Price is more than 10, and less than or equal to 20	`"Price": [ { "numeric": [ ">", 10, "<=", 20 ] } ]`	Yes	Yes
Or	PaymentType is "Credit" or "Debit"	`"PaymentType": [ "Credit", "Debit"]`	Yes	Yes
Or (multiple fields)	Location is "New York", or Day is "Monday".	`"$or": [ { "Location": [ "New York" ] }, { "Day": [ "Monday" ] } ]`	Yes	Yes
Wildcard	Any file with a .png extension, located within the folder "dir"	`"FileName": [ { "wildcard": "dir/*.png" } ]`	Yes	No

Prefix matching

You can match an event depending on the prefix of a value in the event source. You can use prefix matching for string values.

For example, the following event pattern would match any event where the `"time"` field started with `"2017-10-02"` such as `"time": "2017-10-02T18:43:48Z"`.

```
{
  "time": [ { "prefix": "2017-10-02" } ]
}
```

Prefix matching while ignoring case

You can also match a prefix value regardless of the casing of the characters a value begins with, using `equals-ignore-case` in conjunction with `prefix`.

For example, the following event pattern would match any event where the `service` field started with the character string `EventB`, but also `EVENTB`, `eventb`, or any other capitalization of those characters.

```
{
  "detail": {"service" : [{ "prefix": { "equals-ignore-case": "EventB" }}]}
}
```

Suffix matching

You can match an event depending on the suffix of a value in the event source. You can use suffix matching for string values.

For example, the following event pattern would match any event where the `"FileName"` field ends with the `.png` file extension.

```
{
  "FileName": [ { "suffix": ".png" } ]
}
```

Suffix matching while ignoring case

You can also match a suffix value regardless of the casing of the characters a value ends with, using `equals-ignore-case` in conjunction with `suffix`.

For example, the following event pattern would match any event where the `FileName` field ended with the character string `.png`, but also `.PNG` or any other capitalization of those characters.

```
{
  "detail": {"FileName" : [{ "suffix": { "equals-ignore-case": ".png" }}]}
}
```

Anything-but matching

Anything-but matching matches anything except what's specified in the rule.

You can use anything-but matching with strings and numeric values, including lists that contain only strings, or only numbers.

The following event pattern shows anything-but matching with strings and numbers.

```
{
  "detail": {
    "state": [ { "anything-but": "initializing" } ]
  }
}

{
  "detail": {
    "x-limit": [ { "anything-but": 123 } ]
  }
}
```

The following event pattern shows anything-but matching with a list of strings.

```
{
  "detail": {
    "state": [ { "anything-but": [ "stopped", "overloaded" ] } ]
  }
}
```

The following event pattern shows anything-but matching with a list of numbers.

```
{
  "detail": {
    "x-limit": [ { "anything-but": [ 100, 200, 300 ] } ]
  }
}
```

Anything-but matching while ignoring case

You can also use `equals-ignore-case` in conjunction with `anything-but`, to match string values regardless of character casing.

The following event pattern matches `state` fields that do not contain the string "initializing", "INITIALIZING", "Initializing", or any other capitalization of those characters.

```
{
  "detail": {"state" : [{ "anything-but": { "equals-ignore-case": "initializing" }}]}
}
```

You can use `equals-ignore-case` in conjunction with `anything-but` to match against a list of values as well:

```
{
  "detail": {"state" : [{ "anything-but": { "equals-ignore-case": ["initializing",
 "stopped"] }}]}
}
```

Anything-but matching on prefixes

You can use `prefix` in conjunction with `anything-but` to match string values that do not start with the specified value. This includes single values, or a list of values.

The following event pattern shows anything-but matching that matches any event that does not have the prefix `"init"` in the `"state"` field.

```
{
  "detail": {
    "state": [ { "anything-but": { "prefix": "init" } } ]
  }
```

```
}
```

The following event pattern shows anything-but matching used with a list of prefix values. This event pattern matches any event that does not have either the prefix `"init"` or `"stop"` in the `"state"` field.

```
{
"detail": {
  "state" : [{ "anything-but": { "prefix": ["init", "stop"] } } ] }
  }
}
```

Anything-but matching on suffixes

You can use `suffix` in conjunction with `anything-but` to match string values that do not end with the specified value. This includes single values, or a list of values.

The following event pattern matches any values for the `FileName` field that do not end with `.txt`.

```
{
  "detail": {
    "FileName": [ { "anything-but": { "suffix": ".txt" } } ]
  }
}
```

The following event pattern shows anything-but matching used with a list of suffix values. This event pattern matches any values for the `FileName` field that do not end with either `.txt` or `.rtf`.

```
{
  "detail": {
    "FileName": [ { "anything-but": { "suffix": [".txt", ".rtf"] } } ]
  }
}
```

Anything-but matching using wildcards

You can use the wildcard character (*) within the values you specify for anything-but matching. This includes single values, or a list of values.

The following event pattern matches any values for the `FileName` field that do not contain `/lib/`.

```
{
"detail": {
  "FilePath" : [{ "anything-but": { "wildcard": "*/lib/*" }}]
  }
}
```

The following event pattern shows anything-but matching used with a list of values including wildcards. This event pattern matches any values for the `FileName` field that do not contain either `/lib/` or `/bin/`.

```
{
"detail": {
  "FilePath" : [{ "anything-but": { "wildcard": ["*/lib/*", "*/bin/*"] }}]
  }
}
```

For more information, see ???.

Numeric matching

Numeric matching compares values as 64-bit double-precision floating-point numbers (IEEE 754 binary64), which limits the integers that can be matched exactly. Integers are matched exactly from -9,007,199,254,740,992 to 9,007,199,254,740,992 (-2^{53} to $+2^{53}$). If a value falls outside this range or otherwise cannot be represented exactly as a double, EventBridge rejects the rule at creation time when the value is used in a numeric comparison, and an event carrying such a value is not treated as a number and will not match a numeric comparison.

The following shows numeric matching for an event pattern that only matches events that are true for all fields.

```
{
  "detail": {
    "c-count": [ { "numeric": [ ">", 0, "<=", 5 ] } ],
    "d-count": [ { "numeric": [ "<", 10 ] } ],
    "x-limit": [ { "numeric": [ "=", 3.018e2 ] } ]
  }
}
```

IP address matching

You can use IP address matching for IPv4 and IPv6 addresses. The following event pattern shows IP address matching to IP addresses that start with 10.0.0 and end with a number between 0 and 255.

```
{
  "detail": {
    "sourceIPAddress": [ { "cidr": "10.0.0.0/24" } ]
  }
}
```

Exists matching

Exists matching works on the presence or absence of a field in the JSON of the event.

Exists matching only works on leaf nodes. It does not work on intermediate nodes.

The following event pattern matches any event that has a `detail.state` field.

```
{
  "detail": {
    "state": [ { "exists": true  } ]
  }
}
```

The preceding event pattern matches the following event.

```
{
  "version": "0",
  "id": "7bf73129-1428-4cd3-a780-95db273d1602",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "123456789012",
  "time": "2015-11-11T21:29:54Z",
  "region": "us-east-1",
  "resources": ["arn:aws:ec2:us-east-1:123456789012:instance/i-abcd1111"],
  "detail": {
    "instance-id": "i-abcd1111",
    "state": "pending"
  }
```

```
}
```

The preceding event pattern does NOT match the following event because it doesn't have a `detail.state` field.

```
{
  "detail-type": [ "EC2 Instance State-change Notification" ],
  "resources": [ "arn:aws:ec2:us-east-1:123456789012:instance/i-02ebd4584a2ebd341" ],
  "detail": {
    "c-count" : {
       "c1" : 100
    }
  }
}
```

Equals-ignore-case matching

Equals-ignore-case matching works on string values regardless of case.

The following event pattern matches any event that has a `detail-type` field that matches the specified string, regardless of case.

```
{
  "detail-type": [ { "equals-ignore-case": "ec2 instance state-change notification" } ]
}
```

The preceding event pattern matches the following event.

```
{
  "detail-type": [ "EC2 Instance State-change Notification" ],
  "resources": [ "arn:aws:ec2:us-east-1:123456789012:instance/i-02ebd4584a2ebd341" ],
  "detail": {
    "c-count" : {
       "c1" : 100
    }
  }
}
```

Matching using wildcards

You can use the wildcard character (*) to match string values in event patterns.

> **Note**
>
> Currently the wildcard character is supported in event bus rules only.

Considerations when using wildcards in your event patterns:

- You can specify any number of wildcard characters in a given string value; however, consecutive wildcard characters are not supported.
- EventBridge supports using the backslash character (\) to specify the literal * and \ characters in wildcard filters:
  - The string * represents the literal * character
  - The string \\ represents the literal \ character

  Using the backslash to escape other characters is not supported.

Wildcards and event pattern complexity

There is a limit to how complex a rule using wildcards can be. If a rule is too complex, EventBridge returns an `InvalidEventPatternException` when attempting to create the rule. If your rule generates such an error, consider using the guidance below to reduce the complexity of the event pattern:

- **Reduce the number of wildcard characters used**

  Only use wildcard characters where you truly need to match against multiple possible values. For example, consider the following event pattern, where you want to match against event buses in the same Region:

  ```
  {
  "EventBusArn": [ { "wildcard": "*:*:*:*:*:event-bus/*" } ]
  }
  ```

  In the above case, many of the sections of the ARN will be directly based on the Region in which your event buses reside. So if you are using the `us-east-1` Region, a less complex pattern that still matches the desired values might be the following example:

  ```
  {
  ```

```
"EventBusArn": [ { "wildcard": "arn:aws:events:us-east-1:*:event-bus/*" } ]
}
```

- **Reduce repeating character sequences that occur after a wildcard character**

  Having the same character sequence appear multiple times after the use of a wildcard increases the complexity of processing the event pattern. Recast your event pattern to minimize repeated sequences. For example, consider the following example, that matches on the file name `doc.txt` file for any user:

  ```
  {
  "FileName": [ { "wildcard": "/Users/*/dir/dir/dir/dir/dir/doc.txt" } ]
  }
  ```

  If you knew that the `doc.txt` file would only occur in the specified path, you could reduce the repeated character sequence in this way:

  ```
  {
  "FileName": [ { "wildcard": "/Users/*/doc.txt" } ]
  }
  ```

Complex example with multiple matching

You can combine multiple matching criteria into a more complex event pattern. For example, the following event pattern combines `anything-but` and `numeric`.

```
{
  "time": [ { "prefix": "2017-10-02" } ],
  "detail": {
    "state": [ { "anything-but": "initializing" } ],
    "c-count": [ { "numeric": [ ">", 0, "<=", 5 ] } ],
    "d-count": [ { "numeric": [ "<", 10 ] } ],
    "x-limit": [ { "anything-but": [ 100, 200, 300 ] } ]
  }
}
```

Note

When building event patterns, if you include a key more than once the last reference will be the one used to evaluate events. For example, for the following pattern:

```
{
  "detail": {
    "location": [ { "prefix": "us-" } ],
    "location": [ { "anything-but": "us-east" } ]
  }
}
```

only `{ "anything-but": "us-east" }` will be taken into account when evaluating the `location`.

Complex example with `$or` matching

You can also create complex event patterns that check to see if *any* field values match, across multiple fields. Use `$or` to create an event pattern that matches if any of the values for multiple fields are matched.

Note that you can include other filter types, such as numeric matching and arrays, in your pattern matching for individual fields in your `$or` construct.

The following event pattern matches if any of the following conditions are met:

- The `c-count` field is greater than 0 or less than or equal to 5.
- The `d-count` field is less than 10.
- The `x-limit` field equals 3.018e2.

```
{
  "detail": {
    "$or": [
      { "c-count": [ { "numeric": [ ">", 0, "<=", 5 ] } ] },
      { "d-count": [ { "numeric": [ "<", 10 ] } ] },
      { "x-limit": [ { "numeric": [ "=", 3.018e2 ] } ] }
    ]
  }
```

```
}
```

> **Note**
>
> APIs that accept an event pattern (such as `PutRule`, `CreateArchive`, `UpdateArchive`, and `TestEventPattern`) will throw an `InvalidEventPatternException` if the use of `$or` results in over 1000 rule combinations.
> To determine the number of rule combinations in an event pattern, multiply the total number of arguments from each `$or` array in the event pattern. For example, the above pattern contains a single `$or` array with three arguments, so the total number of rule combinations is also three. If you added another `$or` array with two arguments, the total rule combinations would then be six.

Best practices for Amazon EventBridge event patterns

Below are some best practices to consider when defining event patterns in your event bus rules.

Avoid writing infinite loops

In EventBridge, it is possible to create rules that lead to infinite loops, where a rule is fired repeatedly. For example, a rule might detect that ACLs have changed on an S3 bucket, and trigger software to change them to the desired state. If the rule is not written carefully, the subsequent change to the ACLs fires the rule again, creating an infinite loop.

To prevent these issues, write the event patterns for your rules to be as precise as possible, so they only match the events you actually want sent to the target. In the above example, you would create an event pattern to match events so that the triggered actions do not re-fire the same rule. For example, create an event pattern in your rule that would match events only if ACLs are found to be in a bad state, instead of after any change. For more information, see ??? and ???.

An infinite loop can quickly cause higher than expected charges. It can also lead to throttling and delayed event delivery. You can monitor the upper bound of your invocation rates to be warned about unexpected spikes in volume.

Use budgeting to alert you when charges exceed your specified limit. For more information, see Managing Your Costs with Budgets.

Make event patterns precise as possible

The more precise your event pattern, the more likely it will match only the events you actually want it to, and avoid unexpected matches when new events are added to an event source, or existing events are updated to include new properties.

Event patterns can include filters that match on:

- Event metadata about the event, such as `source`, `detail-type`. `account`, or `region`.
- Event data, this is, the fields inside the `detail` object.
- Event content, or the actual values of the fields inside the `detail` object.

Most patterns are simple, such as specifying only `source` and `detail-type` filters. However, EventBridge patterns include the flexibility to filter on any key or value of the event. In addition, you can apply content filters such as `prefix` and `suffix` filters to improve the precision of your patterns. For more information, see ???.

Specify event source and detail type as filters

You can reduce generating infinite loops and matching undesired events by making your event patterns more precise using the `source` and `detail-type` metadata fields.

When you need to match specific values within two or more fields, use the `$or` comparison operator, rather than listing all possible values within a single array of values.

For events that are delivered through AWS CloudTrail, we recommend you use the `eventName` field as a filter.

The following event pattern example matches `CreateQueue` or `SetQueueAttributes` from the Amazon Simple Queue Service service, or `CreateKey` or `DisableKeyRotation` events from the AWS Key Management Service service.

```
{
  "detail-type": ["AWS API Call via CloudTrail"],
  "$or": [{
      "source": [
        "aws.sqs"
        ],
      "detail": {
```

```
        "eventName": [
          "CreateQueue",
          "SetQueueAttributes"
        ]
      }
    },
    {
      "source": [
        "aws.kms"
        ],
      "detail": {
        "eventName": [
          "CreateKey",
          "DisableKeyRotation"
        ]
      }
    }
  ]
}
```

Specify account and region as filters

Including `account` and `region` fields in your event pattern helps limit cross-account or cross-region event matching.

Specify content filters

Content-based filtering can help improve event pattern precision, while still keeping the length of the event pattern to a minimum. For example, matching based on a numeric range can be helpful instead of listing all possible numeric values.

For more information, see ???.

Scope your event patterns to account for event source updates

When creating event patterns, you should take into account that event schemas and event domains may evolve and expand over time. Here again, making your event patterns as precise as possible helps you limit unexpected matches if the event source changes or expands.

For example, suppose you are matching against events from a new micro-service that publishes payment-related events. Initially, the service uses the domain `acme.payments`, and publishes a single event, `Payment accepted`:

```
{
  "detail-type": "Payment accepted",
  "source": "acme.payments",
  "detail": {
    "type": "credit",
    "amount": "100",
    "date": "2023-06-10",
    "currency": "USD"
    }
  }
}
```

At this point, you could create a simple event pattern that matches Payment accepted events:

```
{ “source” : “acme.payments” }
```

However, suppose the service later introduces a new event for rejected payments:

```
{
  "detail-type": "Payment rejected",
  "source": "acme.payments",
  "detail": {
  }
}
```

In this case, the simple event pattern you created will now match against both `Payment accepted` and `Payment rejected` events. EventBridge routes both types of events to the specified target for processing, possibly introducing processing failures and additional processing cost.

To scope your event pattern to only `Payment accepted` events, you'd want to specify both `source` and `detail-type`, at a minimum:

```
{
  "detail-type": "Payment accepted",
  "source": "acme.payments"
  }
}
```

You can also specify account and Region in your event pattern, to further limit when cross-account or cross-Region events match this rule.

```
{
  "account": "012345678910",
  "source": "acme.payments",
  "region": "AWS-Region",
  "detail-type": "Payment accepted"
}
```

Validate event patterns

To ensure rules match the desired events, we strongly recommend you validate your event patterns. You can validate your event patterns using the EventBridge console or API:

- In the EventBridge console, you can create and test event patterns as part of creating a rule, or separately by using the Sandbox.
- You can test your event patterns in the AWS CLI using the test-event-pattern command.

Testing event patterns using the EventBridge Sandbox

Defining an event pattern is typically part of the larger process of creating a new rule or editing an existing one. Using the Sandbox in EventBridge, however, you can quickly define an event pattern and use a sample event to confirm the pattern matches the desired events, without having to create or edit a rule. Once you've got your event pattern tested, EventBridge give you the option of creating a new rule using that event pattern directly from the sandbox.

For more information about event patterns, see ???.

> **Important**
>
> In EventBridge, it is possible to create rules that can lead to higher-than-expected charges and throttling. For example, you can inadvertently create a rule that leads to an infinite loop, where a rule is fired recursively without end. Suppose you created a rule to detect that ACLs have changed on an Amazon S3 bucket, and trigger software to change them to the desired state. If the rule is not written carefully, the subsequent change to the ACLs fires the rule again, creating an infinite loop.
>
> For guidance on how to write precise rules and event patterns to minimize such unexpected results, see Best practices for rules and Best practices.

To test an event pattern using the EventBridge sandbox

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Developer resources,** then select **Sandbox**, and on the **Sandbox** page choose the **Event pattern** tab.
3. For **Event source**, choose **AWS events or EventBridge partner events**.
4. (Optional) In the **Sample events** section, choose a **Sample event type** against which you want to test your event pattern.

   The following sample event types are available:

   - **AWS events** – Select from events emitted from supported AWS services.
   - **EventBridge partner events** – Select from events emitted from third-party services that support EventBridge, such as Salesforce.
   - **Enter my own** – Enter your own event in JSON text.

     You can also use an AWS or partner event as the starting point for creating your own custom event.

     1. Select **AWS events** or **EventBridge partner events**.
     2. Use the **Sample events** dropdown to select the event you want to use as a starting point for your custom event.

        EventBridge displays the sample event.
     3. Select **Copy**.
     4. Select **Enter my own** for **Event type.**
     5. Delete the sample event structure in the JSON editing pane, and paste the AWS or partner event in its place.
     6. Edit the event JSON to create your own sample event.
5. Choose a **Creation method**. You can create an event pattern from an EventBridge schema or template, or you can create a custom event pattern.

   Existing schema

   To use an existing EventBridge schema to create the event pattern, do the following:

   1. In the **Creation method** section, for **Method**, select **Use schema**.

2. In the **Event pattern** section, for **Schema type**, select **Select schema from Schema registry**.
3. For **Schema registry**, choose the dropdown box and enter the name of a schema registry, such as `aws.events`. You can also select an option from the dropdown list that appears.
4. For **Schema**, choose the dropdown box and enter the name of the schema to use. For example, `aws.s3@ObjectDeleted`. You can also select an option from the dropdown list that appears.
5. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

6. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
7. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Custom schema

To write a custom schema and convert it to an event pattern, do the following:

1. In the **Creation method** section, for **Method**, choose **Use schema**.

2. In the **Event pattern** section, for **Schema type**, choose **Enter schema**.
3. Enter your schema into the text box. You must format the schema as valid JSON text.
4. In the **Models** section, choose the **Edit** button next to any attribute to open its properties. Set the **Relationship** and **Value** fields as needed, then choose **Set** to save the attribute.

   > **Note**
   >
   > For information about an attribute's definition, choose the **Info** icon next to the attribute's name. For a reference on how to set attribute properties in your event, open the **Note** section of the attribute properties dialog box.
   > To delete an attribute's properties, choose the **Edit** button for that attribute, then choose **Clear**.

5. Choose **Generate event pattern in JSON** to generate and validate your event pattern as JSON text.
6. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

   - **Copy** – Copy the event pattern to your device's clipboard.
   - **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.

Event pattern

To write a custom event pattern in JSON format, do the following:

1. In the **Creation method** section, for **Method**, choose **Custom pattern (JSON editor)**.
2. For **Event pattern**, enter your custom event pattern in JSON-formatted text.
3. (Optional) To test the sample event against your test pattern, choose **Test pattern**.

   EventBridge displays a message box stating whether your sample event matches the event pattern.

   You can also choose any of the following options:

- **Copy** – Copy the event pattern to your device's clipboard.
- **Prettify** – Makes the JSON text easier to read by adding line breaks, tabs, and spaces.
- **Event pattern form** – Opens the event pattern in Pattern Builder. If the pattern can't be rendered in Pattern Builder as-is, EventBridge warns you before it opens Pattern Builder.

6. (Optional) To create a rule with this event pattern, and assign the rule to a specific event bus, choose **Create rule with pattern**.

   EventBridge takes you to **Step 1** of **Create rule**, which you can use to create a rule and assign it to the event bus of your choice.

   Note that **Step 2 - Build event pattern** contains the event pattern information you've already specified, and which you can accept or update.

   For more on how to create rules, see ???.

Code examples for EventBridge using AWS SDKs

The following code examples show how to use EventBridge with an AWS software development kit (SDK).

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Basic examples for EventBridge using AWS SDKs
  - Hello EventBridge
  - Learn the basics of EventBridge with an AWS SDK
  - Actions for EventBridge using AWS SDKs
    - Use DeleteRule with an AWS SDK or CLI
    - Use DescribeRule with an AWS SDK or CLI
    - Use DisableRule with an AWS SDK or CLI
    - Use EnableRule with an AWS SDK or CLI
    - Use ListRuleNamesByTarget with an AWS SDK or CLI
    - Use ListRules with an AWS SDK or CLI
    - Use ListTargetsByRule with an AWS SDK or CLI
    - Use PutEvents with an AWS SDK or CLI
    - Use PutRule with an AWS SDK or CLI
    - Use PutTargets with an AWS SDK or CLI
    - Use RemoveTargets with an AWS SDK or CLI
- Scenarios for EventBridge using AWS SDKs

- Create and trigger a rule in Amazon EventBridge using an AWS SDK
- Send S3 event notifications to Amazon EventBridge using an AWS SDK
- Use scheduled events to invoke a Lambda function

Basic examples for EventBridge using AWS SDKs

The following code examples show how to use the basics of Amazon EventBridge with AWS SDKs.

Examples

- Hello EventBridge
- Learn the basics of EventBridge with an AWS SDK
- Actions for EventBridge using AWS SDKs
  - Use DeleteRule with an AWS SDK or CLI
  - Use DescribeRule with an AWS SDK or CLI
  - Use DisableRule with an AWS SDK or CLI
  - Use EnableRule with an AWS SDK or CLI
  - Use ListRuleNamesByTarget with an AWS SDK or CLI
  - Use ListRules with an AWS SDK or CLI
  - Use ListTargetsByRule with an AWS SDK or CLI
  - Use PutEvents with an AWS SDK or CLI
  - Use PutRule with an AWS SDK or CLI
  - Use PutTargets with an AWS SDK or CLI
  - Use RemoveTargets with an AWS SDK or CLI

Hello EventBridge

The following code examples show how to get started using EventBridge.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
using Amazon.EventBridge;
using Amazon.EventBridge.Model;

namespace EventBridgeActions;

public static class HelloEventBridge
{
    static async Task Main(string[] args)
    {
        var eventBridgeClient = new AmazonEventBridgeClient();

        Console.WriteLine($"Hello Amazon EventBridge! Following are some of your
 EventBuses:");
        Console.WriteLine();

        // You can use await and any of the async methods to get a response.
        // Let's get the first five event buses.
        var response = await eventBridgeClient.ListEventBusesAsync(
            new ListEventBusesRequest()
            {
                Limit = 5
            });

        foreach (var eventBus in response.EventBuses)
        {
            Console.WriteLine($"\tEventBus: {eventBus.Name}");
            Console.WriteLine($"\tArn: {eventBus.Arn}");
            Console.WriteLine($"\tPolicy: {eventBus.Policy}");
            Console.WriteLine();
        }
    }
```

```
}
```

- For API details, see ListEventBuses in *AWS SDK for .NET API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 */
public class HelloEventBridge {
    public static void main(String[] args) {
        Region region = Region.US_WEST_2;
        EventBridgeClient eventBrClient = EventBridgeClient.builder()
                .region(region)
                .build();

        listBuses(eventBrClient);
        eventBrClient.close();
    }

    public static void listBuses(EventBridgeClient eventBrClient) {
        try {
            ListEventBusesRequest busesRequest = ListEventBusesRequest.builder()
                    .limit(10)
                    .build();
```

```
            ListEventBusesResponse response =
 eventBrClient.listEventBuses(busesRequest);
            List<EventBus> buses = response.eventBuses();
            for (EventBus bus : buses) {
                System.out.println("The name of the event bus is: " +
 bus.name());
                System.out.println("The ARN of the event bus is: " + bus.arn());
            }

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see ListEventBuses in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import aws.sdk.kotlin.services.eventbridge.EventBridgeClient
import aws.sdk.kotlin.services.eventbridge.model.ListEventBusesRequest
import aws.sdk.kotlin.services.eventbridge.model.ListEventBusesResponse

suspend fun main() {
    listBusesHello()
}

suspend fun listBusesHello() {
    val request =
        ListEventBusesRequest {
            limit = 10
        }
```

```
    EventBridgeClient.fromEnvironment { region = "us-west-2" }.use
 { eventBrClient ->
        val response: ListEventBusesResponse =
 eventBrClient.listEventBuses(request)
        response.eventBuses?.forEach { bus ->
            println("The name of the event bus is ${bus.name}")
            println("The ARN of the event bus is ${bus.arn}")
        }
    }
}
```

- For API details, see ListEventBuses in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Learn the basics of EventBridge with an AWS SDK

The following code examples show how to:

- Create a rule and add a target to it.
- Enable and disable rules.
- List and update rules and targets.
- Send events, then clean up resources.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
public class EventBridgeScenario
{
    /*
    Before running this .NET code example, set up your development environment,
 including your credentials.

    This .NET example performs the following tasks with Amazon EventBridge:
    - Create a rule.
    - Add a target to a rule.
    - Enable and disable rules.
    - List rules and targets.
    - Update rules and targets.
    - Send events.
    - Delete the rule.
    */

    private static ILogger logger = null!;
    private static EventBridgeWrapper _eventBridgeWrapper = null!;
    private static IConfiguration _configuration = null!;

    private static IAmazonIdentityManagementService? _iamClient = null!;
    private static IAmazonSimpleNotificationService? _snsClient = null!;
    private static IAmazonS3 _s3Client = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for Amazon EventBridge.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonEventBridge>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddAWSService<IAmazonS3>()
            .AddAWSService<IAmazonSimpleNotificationService>()
            .AddTransient<EventBridgeWrapper>()
            )
            .Build();
```

```
        _configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load settings from .json file.
            .AddJsonFile("settings.local.json",
                true) // Optionally, load local settings.
            .Build();

        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<EventBridgeScenario>();

        ServicesSetup(host);

        string topicArn = "";
        string roleArn = "";

        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Welcome to the Amazon EventBridge example scenario.");
        Console.WriteLine(new string('-', 80));

        try
        {
            roleArn = await CreateRole();

            await CreateBucketWithEventBridgeEvents();

            await AddEventRule(roleArn);

            await ListEventRules();

            topicArn = await CreateSnsTopic();

            var email = await SubscribeToSnsTopic(topicArn);

            await AddSnsTarget(topicArn);

            await ListTargets();

            await ListRulesForTarget(topicArn);

            await UploadS3File(_s3Client);

            await ChangeRuleState(false);

            await GetRuleState();
```

```
            await UpdateSnsEventRule(topicArn);

            await ChangeRuleState(true);

            await UploadS3File(_s3Client);

            await UpdateToCustomRule(topicArn);

            await TriggerCustomRule(email);

            await CleanupResources(topicArn);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "There was a problem executing the scenario.");
            await CleanupResources(topicArn);
        }
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("The Amazon EventBridge example scenario is
complete.");
        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Populate the services for use within the console application.
    /// </summary>
    /// <param name="host">The services host.</param>
    private static void ServicesSetup(IHost host)
    {
        _eventBridgeWrapper =
host.Services.GetRequiredService<EventBridgeWrapper>();
        _snsClient =
host.Services.GetRequiredService<IAmazonSimpleNotificationService>();
        _s3Client = host.Services.GetRequiredService<IAmazonS3>();
        _iamClient =
host.Services.GetRequiredService<IAmazonIdentityManagementService>();
    }

    /// <summary>
    /// Create a role to be used by EventBridge.
    /// </summary>
    /// <returns>The role Amazon Resource Name (ARN).</returns>
    public static async Task<string> CreateRole()
```

```
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Creating a role to use with EventBridge and attaching
 managed policy AmazonEventBridgeFullAccess.");
        Console.WriteLine(new string('-', 80));

        var roleName = _configuration["roleName"];

        var assumeRolePolicy = "{" +
                               "\"Version\": \"2012-10-17\"," +
                               "\"Statement\": [{" +
                               "\"Effect\": \"Allow\"," +
                               "\"Principal\": {" +
                               $"\"Service\": \"events.amazonaws.com\"" +
                               "}," +
                               "\"Action\": \"sts:AssumeRole\"" +
                               "}]" +
                               "}";

        var roleResult = await _iamClient!.CreateRoleAsync(
            new CreateRoleRequest()
            {
                AssumeRolePolicyDocument = assumeRolePolicy,
                Path = "/",
                RoleName = roleName
            });

        await _iamClient.AttachRolePolicyAsync(
            new AttachRolePolicyRequest()
            {
                PolicyArn = "arn:aws:iam::aws:policy/
AmazonEventBridgeFullAccess",
                RoleName = roleName
            });
        // Allow time for the role to be ready.
        Thread.Sleep(10000);
        return roleResult.Role.Arn;
    }

    /// <summary>
    /// Create an Amazon Simple Storage Service (Amazon S3) bucket with
 EventBridge events enabled.
    /// </summary>
    /// <returns>Async task.</returns>
```

```
    private static async Task CreateBucketWithEventBridgeEvents()
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Creating an S3 bucket with EventBridge events
enabled.");

        var testBucketName = _configuration["testBucketName"];

        var bucketExists = await
Amazon.S3.Util.AmazonS3Util.DoesS3BucketExistV2Async(_s3Client,
            testBucketName);

        if (!bucketExists)
        {
            await _s3Client.PutBucketAsync(new PutBucketRequest()
            {
                BucketName = testBucketName,
                UseClientRegion = true
            });
        }

        await _s3Client.PutBucketNotificationAsync(new
PutBucketNotificationRequest()
        {
            BucketName = testBucketName,
            EventBridgeConfiguration = new EventBridgeConfiguration()
        });

        Console.WriteLine($"\tAdded bucket {testBucketName} with EventBridge
events enabled.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Create and upload a file to an S3 bucket to trigger an event.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task UploadS3File(IAmazonS3 s3Client)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Uploading a file to the test bucket. This will trigger
a subscription email.");
```

```
        var testBucketName = _configuration["testBucketName"];

        var fileName = $"example_upload_{DateTime.UtcNow.Ticks}.txt";

        // Create the file if it does not already exist.
        if (!File.Exists(fileName))
        {
            await using StreamWriter sw = File.CreateText(fileName);
            await sw.WriteLineAsync(
                "This is a sample file for testing uploads.");
        }

        await s3Client.PutObjectAsync(new PutObjectRequest()
        {
            FilePath = fileName,
            BucketName = testBucketName
        });

        Console.WriteLine($"\tPress Enter to continue.");
        Console.ReadLine();

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Create an Amazon Simple Notification Service (Amazon SNS) topic to use as
an EventBridge target.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task<string> CreateSnsTopic()
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine(
            "Creating an Amazon Simple Notification Service (Amazon SNS) topic
for email subscriptions.");

        var topicName = _configuration["topicName"];

        string topicPolicy = "{" +
                             "\"Version\": \"2012-10-17\"," +
                             "\"Statement\": [{" +
                             "\"Sid\": \"EventBridgePublishTopic\"," +
                             "\"Effect\": \"Allow\"," +
                             "\"Principal\": {" +
```

```
                            $"\"Service\": \"events.amazonaws.com\"" +
                            "}," +
                            "\"Resource\": \"*\"," +
                            "\"Action\": \"sns:Publish\"" +
                            "}]" +
                            "}";

        var topicAttributes = new Dictionary<string, string>()
        {
            { "Policy", topicPolicy }
        };

        var topicResponse = await _snsClient!.CreateTopicAsync(new
CreateTopicRequest()
        {
            Name = topicName,
            Attributes = topicAttributes

        });

        Console.WriteLine($"\tAdded topic {topicName} for email subscriptions.");

        Console.WriteLine(new string('-', 80));

        return topicResponse.TopicArn;
    }

    /// <summary>
    /// Subscribe a user email to an SNS topic.
    /// </summary>
    /// <param name="topicArn">The ARN of the SNS topic.</param>
    /// <returns>The user's email.</returns>
    private static async Task<string> SubscribeToSnsTopic(string topicArn)
    {
        Console.WriteLine(new string('-', 80));


        string email = "";
        while (string.IsNullOrEmpty(email))
        {
            Console.WriteLine("Enter your email to subscribe to the Amazon SNS
topic:");
            email = Console.ReadLine()!;
        }
```

```
        var subscriptions = new List<string>();
        var paginatedSubscriptions =
_snsClient!.Paginators.ListSubscriptionsByTopic(
            new ListSubscriptionsByTopicRequest()
            {
                TopicArn = topicArn
            });

        // Get the entire list using the paginator.
        await foreach (var subscription in paginatedSubscriptions.Subscriptions)
        {
            subscriptions.Add(subscription.Endpoint);
        }

        if (subscriptions.Contains(email))
        {
            Console.WriteLine($"\tYour email is already subscribed.");
            Console.WriteLine(new string('-', 80));
            return email;
        }

        await _snsClient.SubscribeAsync(new SubscribeRequest()
        {
            TopicArn = topicArn,
            Protocol = "email",
            Endpoint = email
        });

        Console.WriteLine($"Use the link in the email you received to confirm
your subscription, then press Enter to continue.");

        Console.ReadLine();

        Console.WriteLine(new string('-', 80));
        return email;
    }

    /// <summary>
    /// Add a rule which triggers when a file is uploaded to an S3 bucket.
    /// </summary>
    /// <param name="roleArn">The ARN of the role used by EventBridge.</param>
    /// <returns>Async task.</returns>
    private static async Task AddEventRule(string roleArn)
```

```
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Creating an EventBridge event that sends an email when
an Amazon S3 object is created.");

        var eventRuleName = _configuration["eventRuleName"];
        var testBucketName = _configuration["testBucketName"];

        await _eventBridgeWrapper.PutS3UploadRule(roleArn, eventRuleName,
testBucketName);
        Console.WriteLine($"\tAdded event rule {eventRuleName} for bucket
{testBucketName}.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Add an SNS target to the rule.
    /// </summary>
    /// <param name="topicArn">The ARN of the SNS topic.</param>
    /// <returns>Async task.</returns>
    private static async Task AddSnsTarget(string topicArn)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Adding a target to the rule to that sends an email
when the rule is triggered.");

        var eventRuleName = _configuration["eventRuleName"];
        var testBucketName = _configuration["testBucketName"];
        var topicName = _configuration["topicName"];
        await _eventBridgeWrapper.AddSnsTargetToRule(eventRuleName, topicArn);
        Console.WriteLine($"\tAdded event rule {eventRuleName} with Amazon SNS
target {topicName} for bucket {testBucketName}.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// List the event rules on the default event bus.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task ListEventRules()
    {
        Console.WriteLine(new string('-', 80));
```

```
        Console.WriteLine("Current event rules:");

        var rules = await _eventBridgeWrapper.ListAllRulesForEventBus();
        rules.ForEach(r => Console.WriteLine($"\tRule: {r.Name} Description:
{r.Description} State: {r.State}"));

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Update the event target to use a transform.
    /// </summary>
    /// <param name="topicArn">The SNS topic ARN target to update.</param>
    /// <returns>Async task.</returns>
    private static async Task UpdateSnsEventRule(string topicArn)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Let's update the event target with a transform.");

        var eventRuleName = _configuration["eventRuleName"];
        var testBucketName = _configuration["testBucketName"];

        await
_eventBridgeWrapper.UpdateS3UploadRuleTargetWithTransform(eventRuleName,
topicArn);
        Console.WriteLine($"\tUpdated event rule {eventRuleName} with Amazon SNS
target {topicArn} for bucket {testBucketName}.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Update the rule to use a custom event pattern.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task UpdateToCustomRule(string topicArn)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Updating the event pattern to be triggered by a custom
event instead.");

        var eventRuleName = _configuration["eventRuleName"];

        await _eventBridgeWrapper.UpdateCustomEventPattern(eventRuleName);
```

```
        Console.WriteLine($"\tUpdated event rule {eventRuleName} to custom
pattern.");
        await
_eventBridgeWrapper.UpdateCustomRuleTargetWithTransform(eventRuleName,
            topicArn);

        Console.WriteLine($"\tUpdated event target {topicArn}.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Send rule events for a custom rule using the user's email address.
    /// </summary>
    /// <param name="email">The email address to include.</param>
    /// <returns>Async task.</returns>
    private static async Task TriggerCustomRule(string email)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("Sending an event to trigger the rule. This will
trigger a subscription email.");

        await _eventBridgeWrapper.PutCustomEmailEvent(email);

        Console.WriteLine($"\tEvents have been sent. Press Enter to continue.");
        Console.ReadLine();

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// List all of the targets for a rule.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task ListTargets()
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("List all of the targets for a particular rule.");

        var eventRuleName = _configuration["eventRuleName"];
        var targets = await
_eventBridgeWrapper.ListAllTargetsOnRule(eventRuleName);
```

```
        targets.ForEach(t => Console.WriteLine($"\tTarget: {t.Arn} Id: {t.Id}
Input: {t.Input}"));

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// List all of the rules for a particular target.
    /// </summary>
    /// <param name="topicArn">The ARN of the SNS topic.</param>
    /// <returns>Async task.</returns>
    private static async Task ListRulesForTarget(string topicArn)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine("List all of the rules for a particular target.");

        var rules = await _eventBridgeWrapper.ListAllRuleNamesByTarget(topicArn);
        rules.ForEach(r => Console.WriteLine($"\tRule: {r}"));

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Enable or disable a particular rule.
    /// </summary>
    /// <param name="isEnabled">True to enable the rule, otherwise false.</param>
    /// <returns>Async task.</returns>
    private static async Task ChangeRuleState(bool isEnabled)
    {
        Console.WriteLine(new string('-', 80));
        var eventRuleName = _configuration["eventRuleName"];

        if (!isEnabled)
        {
            Console.WriteLine($"Disabling the rule: {eventRuleName}");
            await _eventBridgeWrapper.DisableRuleByName(eventRuleName);
        }
        else
        {
            Console.WriteLine($"Enabling the rule: {eventRuleName}");
            await _eventBridgeWrapper.EnableRuleByName(eventRuleName);
        }

        Console.WriteLine(new string('-', 80));
```

```
    }

    /// <summary>
    /// Get the current state of the rule.
    /// </summary>
    /// <returns>Async task.</returns>
    private static async Task GetRuleState()
    {
        Console.WriteLine(new string('-', 80));
        var eventRuleName = _configuration["eventRuleName"];

        var state = await
_eventBridgeWrapper.GetRuleStateByRuleName(eventRuleName);
        Console.WriteLine($"Rule {eventRuleName} is in current state {state}.");

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Clean up the resources from the scenario.
    /// </summary>
    /// <param name="topicArn">The ARN of the SNS topic to clean up.</param>
    /// <returns>Async task.</returns>
    private static async Task CleanupResources(string topicArn)
    {
        Console.WriteLine(new string('-', 80));
        Console.WriteLine($"Clean up resources.");

        var eventRuleName = _configuration["eventRuleName"];
        if (GetYesNoResponse($"\tDelete all targets and event rule
{eventRuleName}? (y/n)"))
        {
            Console.WriteLine($"\tRemoving all targets from the event rule.");
            await _eventBridgeWrapper.RemoveAllTargetsFromRule(eventRuleName);

            Console.WriteLine($"\tDeleting event rule.");
            await _eventBridgeWrapper.DeleteRuleByName(eventRuleName);
        }

        var topicName = _configuration["topicName"];
        if (GetYesNoResponse($"\tDelete Amazon SNS subscription topic
{topicName}? (y/n)"))
        {
            Console.WriteLine($"\tDeleting topic.");
```

```
            await _snsClient!.DeleteTopicAsync(new DeleteTopicRequest()
            {
                TopicArn = topicArn
            });
        }

        var bucketName = _configuration["testBucketName"];
        if (GetYesNoResponse($"\tDelete Amazon S3 bucket {bucketName}? (y/n)"))
        {
            Console.WriteLine($"\tDeleting bucket.");
            // Delete all objects in the bucket.
            var deleteList = await _s3Client.ListObjectsV2Async(new
 ListObjectsV2Request()
            {
                BucketName = bucketName
            });
            await _s3Client.DeleteObjectsAsync(new DeleteObjectsRequest()
            {
                BucketName = bucketName,
                Objects = deleteList.S3Objects
                    .Select(o => new KeyVersion { Key = o.Key }).ToList()
            });
            // Now delete the bucket.
            await _s3Client.DeleteBucketAsync(new DeleteBucketRequest()
            {
                BucketName = bucketName
            });
        }

        var roleName = _configuration["roleName"];
        if (GetYesNoResponse($"\tDelete role {roleName}? (y/n)"))
        {
            Console.WriteLine($"\tDetaching policy and deleting role.");

            await _iamClient!.DetachRolePolicyAsync(new DetachRolePolicyRequest()
            {
                RoleName = roleName,
                PolicyArn = "arn:aws:iam::aws:policy/
AmazonEventBridgeFullAccess",
            });

            await _iamClient!.DeleteRoleAsync(new DeleteRoleRequest()
            {
                RoleName = roleName
```

```
            });
        }

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Helper method to get a yes or no response from the user.
    /// </summary>
    /// <param name="question">The question string to print on the console.</
param>
    /// <returns>True if the user responds with a yes.</returns>
    private static bool GetYesNoResponse(string question)
    {
        Console.WriteLine(question);
        var ynResponse = Console.ReadLine();
        var response = ynResponse != null &&
                       ynResponse.Equals("y",
                           StringComparison.InvariantCultureIgnoreCase);
        return response;
    }
}
```

Create a class that wraps EventBridge operations.

```
/// <summary>
/// Wrapper for Amazon EventBridge operations.
/// </summary>
public class EventBridgeWrapper
{
    private readonly IAmazonEventBridge _amazonEventBridge;
    private readonly ILogger<EventBridgeWrapper> _logger;

    /// <summary>
    /// Constructor for the EventBridge wrapper.
    /// </summary>
    /// <param name="amazonEventBridge">The injected EventBridge client.</param>
    /// <param name="logger">The injected logger for the wrapper.</param>
    public EventBridgeWrapper(IAmazonEventBridge amazonEventBridge,
 ILogger<EventBridgeWrapper> logger)
```

```
    {
        _amazonEventBridge = amazonEventBridge;
        _logger = logger;
    }

    /// <summary>
    /// Get the state for a rule by the rule name.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="eventBusName">The optional name of the event bus. If empty,
uses the default event bus.</param>
    /// <returns>The state of the rule.</returns>
    public async Task<RuleState> GetRuleStateByRuleName(string ruleName, string?
eventBusName = null)
    {
        var ruleResponse = await _amazonEventBridge.DescribeRuleAsync(
            new DescribeRuleRequest()
            {
                Name = ruleName,
                EventBusName = eventBusName
            });
        return ruleResponse.State;
    }

    /// <summary>
    /// Enable a particular rule on an event bus.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> EnableRuleByName(string ruleName)
    {
        var ruleResponse = await _amazonEventBridge.EnableRuleAsync(
            new EnableRuleRequest()
            {
                Name = ruleName
            });
        return ruleResponse.HttpStatusCode == HttpStatusCode.OK;
    }

    /// <summary>
    /// Disable a particular rule on an event bus.
    /// </summary
    /// <param name="ruleName">The name of the rule.</param>
    /// <returns>True if successful.</returns>
```

```
    public async Task<bool> DisableRuleByName(string ruleName)
    {
        var ruleResponse = await _amazonEventBridge.DisableRuleAsync(
            new DisableRuleRequest()
            {
                Name = ruleName
            });
        return ruleResponse.HttpStatusCode == HttpStatusCode.OK;
    }

    /// <summary>
    /// List the rules on an event bus.
    /// </summary>
    /// <param name="eventBusArn">The optional ARN of the event bus. If empty,
uses the default event bus.</param>
    /// <returns>The list of rules.</returns>
    public async Task<List<Rule>> ListAllRulesForEventBus(string? eventBusArn =
null)
    {
        var results = new List<Rule>();
        var request = new ListRulesRequest()
        {
            EventBusName = eventBusArn
        };
        // Get all of the pages of rules.
        ListRulesResponse response;
        do
        {
            response = await _amazonEventBridge.ListRulesAsync(request);
            results.AddRange(response.Rules);
            request.NextToken = response.NextToken;

        } while (response.NextToken is not null);

        return results;
    }

    /// <summary>
    /// List all of the targets matching a rule by name.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <returns>The list of targets.</returns>
    public async Task<List<Target>> ListAllTargetsOnRule(string ruleName)
    {
```

```
        var results = new List<Target>();
        var request = new ListTargetsByRuleRequest()
        {
            Rule = ruleName
        };
        ListTargetsByRuleResponse response;
        do
        {
            response = await _amazonEventBridge.ListTargetsByRuleAsync(request);
            results.AddRange(response.Targets);
            request.NextToken = response.NextToken;

        } while (response.NextToken is not null);

        return results;
    }

    /// <summary>
    /// List names of all rules matching a target.
    /// </summary>
    /// <param name="targetArn">The ARN of the target.</param>
    /// <returns>The list of rule names.</returns>
    public async Task<List<string>> ListAllRuleNamesByTarget(string targetArn)
    {
        var results = new List<string>();
        var request = new ListRuleNamesByTargetRequest()
        {
            TargetArn = targetArn
        };
        ListRuleNamesByTargetResponse response;
        do
        {
            response = await
_amazonEventBridge.ListRuleNamesByTargetAsync(request);
            results.AddRange(response.RuleNames);
            request.NextToken = response.NextToken;

        } while (response.NextToken is not null);

        return results;
    }

    /// <summary>
```

```
    /// Create a new event rule that triggers when an Amazon S3 object is created
 in a bucket.
    /// </summary>
    /// <param name="roleArn">The ARN of the role.</param>
    /// <param name="ruleName">The name to give the rule.</param>
    /// <param name="bucketName">The name of the bucket to trigger the event.</
param>
    /// <returns>The ARN of the new rule.</returns>
    public async Task<string> PutS3UploadRule(string roleArn, string ruleName,
 string bucketName)
    {
        string eventPattern = "{" +
                               "\"source\": [\"aws.s3\"]," +
                                   "\"detail-type\": [\"Object Created\"]," +
                                   "\"detail\": {" +
                                       "\"bucket\": {" +
                                           "\"name\": [\"" + bucketName + "\"]"
 +
                                       "}" +
                                   "}" +
                               "}";

        var response = await _amazonEventBridge.PutRuleAsync(
            new PutRuleRequest()
            {
                Name = ruleName,
                Description = "Example S3 upload rule for EventBridge",
                RoleArn = roleArn,
                EventPattern = eventPattern
            });

        return response.RuleArn;
    }

    /// <summary>
    /// Update an Amazon S3 object created rule with a transform on the target.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="targetArn">The ARN of the target.</param>
    /// <param name="eventBusArn">Optional event bus ARN. If empty, uses the
 default event bus.</param>
    /// <returns>The ID of the target.</returns>
    public async Task<string> UpdateS3UploadRuleTargetWithTransform(string
 ruleName, string targetArn, string? eventBusArn = null)
```

```
    {
        var targetID = Guid.NewGuid().ToString();

        var targets = new List<Target>
        {
            new Target()
            {
                Id = targetID,
                Arn = targetArn,
                InputTransformer = new InputTransformer()
                {
                    InputPathsMap = new Dictionary<string, string>()
                    {
                        {"bucket", "$.detail.bucket.name"},
                        {"time", "$.time"}
                    },
                    InputTemplate = "\"Notification: an object was uploaded to
bucket <bucket> at <time>.\""
                }
            }
        };
        var response = await _amazonEventBridge.PutTargetsAsync(
            new PutTargetsRequest()
            {
                EventBusName = eventBusArn,
                Rule = ruleName,
                Targets = targets,
            });
        if (response.FailedEntryCount > 0)
        {
            response.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to add target {e.TargetId}: {e.ErrorMessage}, code
{e.ErrorCode}");
            });
        }
        return targetID;
    }

    /// <summary>
    /// Update a custom rule with a transform on the target.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
```

```
    /// <param name="targetArn">The ARN of the target.</param>
    /// <param name="eventBusArn">Optional event bus ARN. If empty, uses the
 default event bus.</param>
    /// <returns>The ID of the target.</returns>
    public async Task<string> UpdateCustomRuleTargetWithTransform(string
 ruleName, string targetArn, string? eventBusArn = null)
    {
        var targetID = Guid.NewGuid().ToString();

        var targets = new List<Target>
        {
            new Target()
            {
                Id = targetID,
                Arn = targetArn,
                InputTransformer = new InputTransformer()
                {
                    InputTemplate = "\"Notification: sample event was received.
\""
                }
            }
        };
        var response = await _amazonEventBridge.PutTargetsAsync(
            new PutTargetsRequest()
            {
                EventBusName = eventBusArn,
                Rule = ruleName,
                Targets = targets,
            });
        if (response.FailedEntryCount > 0)
        {
            response.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to add target {e.TargetId}: {e.ErrorMessage}, code
 {e.ErrorCode}");
            });
        }
        return targetID;
    }

    /// <summary>
    /// Add an event to the event bus that includes an email, message, and time.
    /// </summary>
```

```
    /// <param name="email">The email to use in the event detail of the custom
event.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> PutCustomEmailEvent(string email)
    {
        var eventDetail = new
        {
            UserEmail = email,
            Message = "This event was generated by example code.",
            UtcTime = DateTime.UtcNow.ToString("g")
        };
        var response = await _amazonEventBridge.PutEventsAsync(
            new PutEventsRequest()
            {
                Entries = new List<PutEventsRequestEntry>()
                {
                    new PutEventsRequestEntry()
                    {
                        Source = "ExampleSource",
                        Detail = JsonSerializer.Serialize(eventDetail),
                        DetailType = "ExampleType"
                    }
                }
            });

        return response.FailedEntryCount == 0;
    }

    /// <summary>
    /// Update a rule to use a custom defined event pattern.
    /// </summary>
    /// <param name="ruleName">The name of the rule to update.</param>
    /// <returns>The ARN of the updated rule.</returns>
    public async Task<string> UpdateCustomEventPattern(string ruleName)
    {
        string customEventsPattern = "{" +
                                     "\"source\": [\"ExampleSource\"]," +
                                     "\"detail-type\": [\"ExampleType\"]" +
                                     "}";

        var response = await _amazonEventBridge.PutRuleAsync(
            new PutRuleRequest()
            {
                Name = ruleName,
```

```
                Description = "Custom test rule",
                EventPattern = customEventsPattern
            });

        return response.RuleArn;
    }

    /// <summary>
    /// Add an Amazon SNS target topic to a rule.
    /// </summary>
    /// <param name="ruleName">The name of the rule to update.</param>
    /// <param name="targetArn">The ARN of the Amazon SNS target.</param>
    /// <param name="eventBusArn">The optional event bus name, uses default if
empty.</param>
    /// <returns>The ID of the target.</returns>
    public async Task<string> AddSnsTargetToRule(string ruleName, string
targetArn, string? eventBusArn = null)
    {
        var targetID = Guid.NewGuid().ToString();

        // Create the list of targets and add a new target.
        var targets = new List<Target>
        {
            new Target()
            {
                Arn = targetArn,
                Id = targetID
            }
        };

        // Add the targets to the rule.
        var response = await _amazonEventBridge.PutTargetsAsync(
            new PutTargetsRequest()
            {
                EventBusName = eventBusArn,
                Rule = ruleName,
                Targets = targets,
            });

        if (response.FailedEntryCount > 0)
        {
            response.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
```

```
                    $"Failed to add target {e.TargetId}: {e.ErrorMessage}, code
{e.ErrorCode}");
            });
        }

        return targetID;
    }

    /// <summary>
    /// Delete an event rule by name.
    /// </summary>
    /// <param name="ruleName">The name of the event rule.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> RemoveAllTargetsFromRule(string ruleName)
    {
        var targetIds = new List<string>();
        var request = new ListTargetsByRuleRequest()
        {
            Rule = ruleName
        };
        ListTargetsByRuleResponse targetsResponse;
        do
        {
            targetsResponse = await
_amazonEventBridge.ListTargetsByRuleAsync(request);
            targetIds.AddRange(targetsResponse.Targets.Select(t => t.Id));
            request.NextToken = targetsResponse.NextToken;

        } while (targetsResponse.NextToken is not null);

        var removeResponse = await _amazonEventBridge.RemoveTargetsAsync(
            new RemoveTargetsRequest()
            {
                Rule = ruleName,
                Ids = targetIds
            });

        if (removeResponse.FailedEntryCount > 0)
        {
            removeResponse.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to remove target {e.TargetId}: {e.ErrorMessage},
code {e.ErrorCode}");
```

```
            });
        }

        return removeResponse.HttpStatusCode == HttpStatusCode.OK;
    }

    /// <summary>
    /// Delete an event rule by name.
    /// </summary>
    /// <param name="ruleName">The name of the event rule.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> DeleteRuleByName(string ruleName)
    {
        var response = await _amazonEventBridge.DeleteRuleAsync(
            new DeleteRuleRequest()
            {
                Name = ruleName
            });

        return response.HttpStatusCode == HttpStatusCode.OK;
    }
}
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.
  - DeleteRule
  - DescribeRule
  - DisableRule
  - EnableRule
  - ListRuleNamesByTarget
  - ListRules
  - ListTargetsByRule
  - PutEvents
  - PutRule
  - PutTargets

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * This Java code example performs the following tasks:
 *
 * This Java V2 example performs the following tasks with Amazon EventBridge:
 *
 * 1. Creates an AWS Identity and Access Management (IAM) role to use with
 * Amazon EventBridge.
 * 2. Amazon Simple Storage Service (Amazon S3) bucket with EventBridge events
 * enabled.
 * 3. Creates a rule that triggers when an object is uploaded to Amazon S3.
 * 4. Lists rules on the event bus.
 * 5. Creates a new Amazon Simple Notification Service (Amazon SNS) topic and
 * lets the user subscribe to it.
 * 6. Adds a target to the rule that sends an email to the specified topic.
 * 7. Creates an EventBridge event that sends an email when an Amazon S3 object
 * is created.
 * 8. Lists Targets.
 * 9. Lists the rules for the same target.
 * 10. Triggers the rule by uploading a file to the Amazon S3 bucket.
 * 11. Disables a specific rule.
 * 12. Checks and print the state of the rule.
 * 13. Adds a transform to the rule to change the text of the email.
 * 14. Enables a specific rule.
 * 15. Triggers the updated rule by uploading a file to the Amazon S3 bucket.
```

```
 * 16. Updates the rule to be a custom rule pattern.
 * 17. Sending an event to trigger the rule.
 * 18. Cleans up resources.
 *
 */
public class EventbridgeMVP {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException,
 IOException {
        final String usage = """

                Usage:
                    <roleName> <bucketName> <topicName> <eventRuleName>

                Where:
                    roleName - The name of the role to create.
                    bucketName - The Amazon Simple Storage Service (Amazon S3)
 bucket name to create.
                    topicName - The name of the Amazon Simple Notification
 Service (Amazon SNS) topic to create.
                    eventRuleName - The Amazon EventBridge rule name to create.
                """;

        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String polJSON = "{" +
                "\"Version\": \"2012-10-17\"," +
                "\"Statement\": [{" +
                "\"Effect\": \"Allow\"," +
                "\"Principal\": {" +
                "\"Service\": \"events.amazonaws.com\"" +
                "}," +
                "\"Action\": \"sts:AssumeRole\"" +
                "}]" +
                "}";

        Scanner sc = new Scanner(System.in);
        String roleName = args[0];
        String bucketName = args[1];
```

```
        String topicName = args[2];
        String eventRuleName = args[3];

        Region region = Region.US_EAST_1;
        EventBridgeClient eventBrClient = EventBridgeClient.builder()
                .region(region)
                .build();

        S3Client s3Client = S3Client.builder()
                .region(region)
                .build();

        Region regionGl = Region.AWS_GLOBAL;
        IamClient iam = IamClient.builder()
                .region(regionGl)
                .build();

        SnsClient snsClient = SnsClient.builder()
                .region(region)
                .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the Amazon EventBridge example
scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out
                .println("1. Create an AWS Identity and Access Management (IAM)
role to use with Amazon EventBridge.");
        String roleArn = createIAMRole(iam, roleName, polJSON);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Create an S3 bucket with EventBridge events
enabled.");
        if (checkBucket(s3Client, bucketName)) {
            System.out.println("Bucket " + bucketName + " already exists. Ending
this scenario.");
            System.exit(1);
        }

        createBucket(s3Client, bucketName);
        Thread.sleep(3000);
```

```
        setBucketNotification(s3Client, bucketName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. Create a rule that triggers when an object is
uploaded to Amazon S3.");
        Thread.sleep(10000);
        addEventRule(eventBrClient, roleArn, bucketName, eventRuleName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. List rules on the event bus.");
        listRules(eventBrClient);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Create a new SNS topic for testing and let the
user subscribe to the topic.");
        String topicArn = createSnsTopic(snsClient, topicName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Add a target to the rule that sends an email to
the specified topic.");
        System.out.println("Enter your email to subscribe to the Amazon SNS
topic:");
        String email = sc.nextLine();
        subEmail(snsClient, topicArn, email);
        System.out.println(
                "Use the link in the email you received to confirm your
subscription. Then, press Enter to continue.");
        sc.nextLine();
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Create an EventBridge event that sends an email
when an Amazon S3 object is created.");
        addSnsEventRule(eventBrClient, eventRuleName, topicArn, topicName,
eventRuleName, bucketName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(" 8. List Targets.");
        listTargets(eventBrClient, eventRuleName);
```

```
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(" 9. List the rules for the same target.");
        listTargetRules(eventBrClient, topicArn);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(" 10. Trigger the rule by uploading a file to the S3
bucket.");
        System.out.println("Press Enter to continue.");
        sc.nextLine();
        uploadTextFiletoS3(s3Client, bucketName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("11. Disable a specific rule.");
        changeRuleState(eventBrClient, eventRuleName, false);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("12. Check and print the state of the rule.");
        checkRule(eventBrClient, eventRuleName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("13. Add a transform to the rule to change the text of
the email.");
        updateSnsEventRule(eventBrClient, topicArn, eventRuleName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("14. Enable a specific rule.");
        changeRuleState(eventBrClient, eventRuleName, true);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(" 15. Trigger the updated rule by uploading a file to
the S3 bucket.");
        System.out.println("Press Enter to continue.");
        sc.nextLine();
        uploadTextFiletoS3(s3Client, bucketName);
        System.out.println(DASHES);
```

```
        System.out.println(DASHES);
        System.out.println(" 16. Update the rule to be a custom rule pattern.");
        updateToCustomRule(eventBrClient, eventRuleName);
        System.out.println("Updated event rule " + eventRuleName + " to use a
custom pattern.");
        updateCustomRuleTargetWithTransform(eventBrClient, topicArn,
eventRuleName);
        System.out.println("Updated event target " + topicArn + ".");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("17. Sending an event to trigger the rule. This will
trigger a subscription email.");
        triggerCustomRule(eventBrClient, email);
        System.out.println("Events have been sent. Press Enter to continue.");
        sc.nextLine();
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("18. Clean up resources.");
        System.out.println("Do you want to clean up resources (y/n)");
        String ans = sc.nextLine();
        if (ans.compareTo("y") == 0) {
            cleanupResources(eventBrClient, snsClient, s3Client, iam, topicArn,
eventRuleName, bucketName, roleName);
        } else {
            System.out.println("The resources will not be cleaned up. ");
        }
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The Amazon EventBridge example scenario has
successfully completed.");
        System.out.println(DASHES);
    }

    public static void cleanupResources(EventBridgeClient eventBrClient,
SnsClient snsClient, S3Client s3Client,
            IamClient iam, String topicArn, String eventRuleName, String
bucketName, String roleName) {
        System.out.println("Removing all targets from the event rule.");
        deleteTargetsFromRule(eventBrClient, eventRuleName);
        deleteRuleByName(eventBrClient, eventRuleName);
        deleteSNSTopic(snsClient, topicArn);
```

```
        deleteS3Bucket(s3Client, bucketName);
        deleteRole(iam, roleName);
    }

    public static void deleteRole(IamClient iam, String roleName) {
        String policyArn = "arn:aws:iam::aws:policy/AmazonEventBridgeFullAccess";
        DetachRolePolicyRequest policyRequest = DetachRolePolicyRequest.builder()
                .policyArn(policyArn)
                .roleName(roleName)
                .build();

        iam.detachRolePolicy(policyRequest);
        System.out.println("Successfully detached policy " + policyArn + " from
role " + roleName);

        // Delete the role.
        DeleteRoleRequest roleRequest = DeleteRoleRequest.builder()
                .roleName(roleName)
                .build();

        iam.deleteRole(roleRequest);
        System.out.println("*** Successfully deleted " + roleName);
    }

    public static void deleteS3Bucket(S3Client s3Client, String bucketName) {
        // Remove all the objects from the S3 bucket.
        ListObjectsRequest listObjects = ListObjectsRequest.builder()
                .bucket(bucketName)
                .build();

        ListObjectsResponse res = s3Client.listObjects(listObjects);
        List<S3Object> objects = res.contents();
        ArrayList<ObjectIdentifier> toDelete = new ArrayList<>();

        for (S3Object myValue : objects) {
            toDelete.add(ObjectIdentifier.builder()
                    .key(myValue.key())
                    .build());
        }

        DeleteObjectsRequest dor = DeleteObjectsRequest.builder()
                .bucket(bucketName)
                .delete(Delete.builder()
                        .objects(toDelete).build())
```

```
                .build();

        s3Client.deleteObjects(dor);

        // Delete the S3 bucket.
        DeleteBucketRequest deleteBucketRequest = DeleteBucketRequest.builder()
                .bucket(bucketName)
                .build();

        s3Client.deleteBucket(deleteBucketRequest);
        System.out.println("You have deleted the bucket and the objects");
    }

    // Delete the SNS topic.
    public static void deleteSNSTopic(SnsClient snsClient, String topicArn) {
        try {
            DeleteTopicRequest request = DeleteTopicRequest.builder()
                    .topicArn(topicArn)
                    .build();

            DeleteTopicResponse result = snsClient.deleteTopic(request);
            System.out.println("\n\nStatus was " +
result.sdkHttpResponse().statusCode());

        } catch (SnsException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static void deleteRuleByName(EventBridgeClient eventBrClient, String
ruleName) {
        DeleteRuleRequest ruleRequest = DeleteRuleRequest.builder()
                .name(ruleName)
                .build();

        eventBrClient.deleteRule(ruleRequest);
        System.out.println("Successfully deleted the rule");
    }

    public static void deleteTargetsFromRule(EventBridgeClient eventBrClient,
String eventRuleName) {
        // First, get all targets that will be deleted.
        ListTargetsByRuleRequest request = ListTargetsByRuleRequest.builder()
```

```
                .rule(eventRuleName)
                .build();

        ListTargetsByRuleResponse response =
eventBrClient.listTargetsByRule(request);
        List<Target> allTargets = response.targets();

        // Get all targets and delete them.
        for (Target myTarget : allTargets) {
            RemoveTargetsRequest removeTargetsRequest =
RemoveTargetsRequest.builder()
                    .rule(eventRuleName)
                    .ids(myTarget.id())
                    .build();

            eventBrClient.removeTargets(removeTargetsRequest);
            System.out.println("Successfully removed the target");
        }
    }

    public static void triggerCustomRule(EventBridgeClient eventBrClient, String
email) {
        String json = "{" +
                "\"UserEmail\": \"" + email + "\"," +
                "\"Message\": \"This event was generated by example code.\"," +
                "\"UtcTime\": \"Now.\"" +
                "}";

        PutEventsRequestEntry entry = PutEventsRequestEntry.builder()
                .source("ExampleSource")
                .detail(json)
                .detailType("ExampleType")
                .build();

        PutEventsRequest eventsRequest = PutEventsRequest.builder()
                .entries(entry)
                .build();

        eventBrClient.putEvents(eventsRequest);
    }

    public static void updateCustomRuleTargetWithTransform(EventBridgeClient
eventBrClient, String topicArn,
            String ruleName) {
```

```
        String targetId = java.util.UUID.randomUUID().toString();
        InputTransformer inputTransformer = InputTransformer.builder()
                .inputTemplate("\"Notification: sample event was received.\"")
                .build();

        Target target = Target.builder()
                .id(targetId)
                .arn(topicArn)
                .inputTransformer(inputTransformer)
                .build();

        try {
            PutTargetsRequest targetsRequest = PutTargetsRequest.builder()
                    .rule(ruleName)
                    .targets(target)
                    .eventBusName(null)
                    .build();

            eventBrClient.putTargets(targetsRequest);
        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static void updateToCustomRule(EventBridgeClient eventBrClient, String
ruleName) {
        String customEventsPattern = "{" +
                "\"source\": [\"ExampleSource\"]," +
                "\"detail-type\": [\"ExampleType\"]" +
                "}";

        PutRuleRequest request = PutRuleRequest.builder()
                .name(ruleName)
                .description("Custom test rule")
                .eventPattern(customEventsPattern)
                .build();

        eventBrClient.putRule(request);
    }

    // Update an Amazon S3 object created rule with a transform on the target.
    public static void updateSnsEventRule(EventBridgeClient eventBrClient, String
topicArn, String ruleName) {
```

```
        String targetId = java.util.UUID.randomUUID().toString();
        Map<String, String> myMap = new HashMap<>();
        myMap.put("bucket", "$.detail.bucket.name");
        myMap.put("time", "$.time");

        InputTransformer inputTransformer = InputTransformer.builder()
                .inputTemplate("\"Notification: an object was uploaded to bucket
<bucket> at <time>.\"")
                .inputPathsMap(myMap)
                .build();

        Target target = Target.builder()
                .id(targetId)
                .arn(topicArn)
                .inputTransformer(inputTransformer)
                .build();

        try {
            PutTargetsRequest targetsRequest = PutTargetsRequest.builder()
                    .rule(ruleName)
                    .targets(target)
                    .eventBusName(null)
                    .build();

            eventBrClient.putTargets(targetsRequest);

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static void checkRule(EventBridgeClient eventBrClient, String
eventRuleName) {
        try {
            DescribeRuleRequest ruleRequest = DescribeRuleRequest.builder()
                    .name(eventRuleName)
                    .build();

            DescribeRuleResponse response =
eventBrClient.describeRule(ruleRequest);
            System.out.println("The state of the rule is " +
response.stateAsString());
```

```
        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static void changeRuleState(EventBridgeClient eventBrClient, String
eventRuleName, Boolean isEnabled) {
        try {
            if (!isEnabled) {
                System.out.println("Disabling the rule: " + eventRuleName);
                DisableRuleRequest ruleRequest = DisableRuleRequest.builder()
                        .name(eventRuleName)
                        .build();

                eventBrClient.disableRule(ruleRequest);
            } else {
                System.out.println("Enabling the rule: " + eventRuleName);
                EnableRuleRequest ruleRequest = EnableRuleRequest.builder()
                        .name(eventRuleName)
                        .build();
                eventBrClient.enableRule(ruleRequest);
            }

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    // Create and upload a file to an S3 bucket to trigger an event.
    public static void uploadTextFiletoS3(S3Client s3Client, String bucketName)
throws IOException {
        // Create a unique file name.
        String fileSuffix = new SimpleDateFormat("yyyyMMddHHmmss").format(new
Date());
        String fileName = "TextFile" + fileSuffix + ".txt";

        File myFile = new File(fileName);
        FileWriter fw = new FileWriter(myFile.getAbsoluteFile());
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write("This is a sample file for testing uploads.");
        bw.close();
```

```
        try {
            PutObjectRequest putOb = PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(fileName)
                    .build();

            s3Client.putObject(putOb, RequestBody.fromFile(myFile));

        } catch (S3Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void listTargetRules(EventBridgeClient eventBrClient, String
topicArn) {
        ListRuleNamesByTargetRequest ruleNamesByTargetRequest =
ListRuleNamesByTargetRequest.builder()
                .targetArn(topicArn)
                .build();

        ListRuleNamesByTargetResponse response =
eventBrClient.listRuleNamesByTarget(ruleNamesByTargetRequest);
        List<String> rules = response.ruleNames();
        for (String rule : rules) {
            System.out.println("The rule name is " + rule);
        }
    }

    public static void listTargets(EventBridgeClient eventBrClient, String
ruleName) {
        ListTargetsByRuleRequest ruleRequest = ListTargetsByRuleRequest.builder()
                .rule(ruleName)
                .build();

        ListTargetsByRuleResponse res =
eventBrClient.listTargetsByRule(ruleRequest);
        List<Target> targetsList = res.targets();
        for (Target target: targetsList) {
            System.out.println("Target ARN: "+target.arn());
        }
    }

    // Add a rule which triggers an SNS target when a file is uploaded to an S3
```

```
   // bucket.
   public static void addSnsEventRule(EventBridgeClient eventBrClient, String
ruleName, String topicArn,
           String topicName, String eventRuleName, String bucketName) {
       String targetID = java.util.UUID.randomUUID().toString();
       Target myTarget = Target.builder()
               .id(targetID)
               .arn(topicArn)
               .build();

       List<Target> targets = new ArrayList<>();
       targets.add(myTarget);
       PutTargetsRequest request = PutTargetsRequest.builder()
               .eventBusName(null)
               .targets(targets)
               .rule(ruleName)
               .build();

       eventBrClient.putTargets(request);
       System.out.println("Added event rule " + eventRuleName + " with Amazon
SNS target " + topicName + " for bucket "
               + bucketName + ".");
   }

   public static void subEmail(SnsClient snsClient, String topicArn, String
email) {
       try {
           SubscribeRequest request = SubscribeRequest.builder()
                   .protocol("email")
                   .endpoint(email)
                   .returnSubscriptionArn(true)
                   .topicArn(topicArn)
                   .build();

           SubscribeResponse result = snsClient.subscribe(request);
           System.out.println("Subscription ARN: " + result.subscriptionArn() +
"\n\n Status is "
                   + result.sdkHttpResponse().statusCode());

       } catch (SnsException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
           System.exit(1);
       }
   }
```

```
   public static void listRules(EventBridgeClient eventBrClient) {
       try {
           ListRulesRequest rulesRequest = ListRulesRequest.builder()
                   .eventBusName("default")
                   .limit(10)
                   .build();

           ListRulesResponse response = eventBrClient.listRules(rulesRequest);
           List<Rule> rules = response.rules();
           for (Rule rule : rules) {
               System.out.println("The rule name is : " + rule.name());
               System.out.println("The rule description is : " +
rule.description());
               System.out.println("The rule state is : " +
rule.stateAsString());
           }

       } catch (EventBridgeException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
           System.exit(1);
       }
   }

   public static String createSnsTopic(SnsClient snsClient, String topicName) {
       String topicPolicy = "{" +
               "\"Version\": \"2012-10-17\"," +
               "\"Statement\": [{" +
               "\"Sid\": \"EventBridgePublishTopic\"," +
               "\"Effect\": \"Allow\"," +
               "\"Principal\": {" +
               "\"Service\": \"events.amazonaws.com\"" +
               "}," +
               "\"Resource\": \"*\"," +
               "\"Action\": \"sns:Publish\"" +
               "}]" +
               "}";

       Map<String, String> topicAttributes = new HashMap<>();
       topicAttributes.put("Policy", topicPolicy);
       CreateTopicRequest topicRequest = CreateTopicRequest.builder()
               .name(topicName)
               .attributes(topicAttributes)
               .build();
```

```
        CreateTopicResponse response = snsClient.createTopic(topicRequest);
        System.out.println("Added topic " + topicName + " for email
subscriptions.");
        return response.topicArn();
    }

    // Create a new event rule that triggers when an Amazon S3 object is created
in
    // a bucket.
    public static void addEventRule(EventBridgeClient eventBrClient, String
roleArn, String bucketName,
            String eventRuleName) {
        String pattern = "{\n" +
                "  \"source\": [\"aws.s3\"],\n" +
                "  \"detail-type\": [\"Object Created\"],\n" +
                "  \"detail\": {\n" +
                "    \"bucket\": {\n" +
                "      \"name\": [\"" + bucketName + "\"]\n" +
                "    }\n" +
                "  }\n" +
                "}";

        try {
            PutRuleRequest ruleRequest = PutRuleRequest.builder()
                    .description("Created by using the AWS SDK for Java v2")
                    .name(eventRuleName)
                    .eventPattern(pattern)
                    .roleArn(roleArn)
                    .build();

            PutRuleResponse ruleResponse = eventBrClient.putRule(ruleRequest);
            System.out.println("The ARN of the new rule is " +
ruleResponse.ruleArn());

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    // Determine if the S3 bucket exists.
    public static Boolean checkBucket(S3Client s3Client, String bucketName) {
        try {
```

```
            HeadBucketRequest headBucketRequest = HeadBucketRequest.builder()
                    .bucket(bucketName)
                    .build();

            s3Client.headBucket(headBucketRequest);
            return true;
        } catch (S3Exception e) {
            System.err.println(e.awsErrorDetails().errorMessage());
        }
        return false;
    }

    // Set the S3 bucket notification configuration.
    public static void setBucketNotification(S3Client s3Client, String
bucketName) {
        try {
            EventBridgeConfiguration eventBridgeConfiguration =
EventBridgeConfiguration.builder()
                    .build();

            NotificationConfiguration configuration =
NotificationConfiguration.builder()
                    .eventBridgeConfiguration(eventBridgeConfiguration)
                    .build();

            PutBucketNotificationConfigurationRequest configurationRequest =
PutBucketNotificationConfigurationRequest
                    .builder()
                    .bucket(bucketName)
                    .notificationConfiguration(configuration)
                    .skipDestinationValidation(true)
                    .build();

            s3Client.putBucketNotificationConfiguration(configurationRequest);
            System.out.println("Added bucket " + bucketName + " with EventBridge
events enabled.");

        } catch (S3Exception e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static void createBucket(S3Client s3Client, String bucketName) {
```

```
        try {
            S3Waiter s3Waiter = s3Client.waiter();
            CreateBucketRequest bucketRequest = CreateBucketRequest.builder()
                    .bucket(bucketName)
                    .build();

            s3Client.createBucket(bucketRequest);
            HeadBucketRequest bucketRequestWait = HeadBucketRequest.builder()
                    .bucket(bucketName)
                    .build();

            // Wait until the bucket is created and print out the response.
            WaiterResponse<HeadBucketResponse> waiterResponse =
 s3Waiter.waitUntilBucketExists(bucketRequestWait);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println(bucketName + " is ready");

        } catch (S3Exception e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }

    public static String createIAMRole(IamClient iam, String rolename, String
 polJSON) {
        try {
            CreateRoleRequest request = CreateRoleRequest.builder()
                    .roleName(rolename)
                    .assumeRolePolicyDocument(polJSON)
                    .description("Created using the AWS SDK for Java")
                    .build();

            CreateRoleResponse response = iam.createRole(request);
            AttachRolePolicyRequest rolePolicyRequest =
 AttachRolePolicyRequest.builder()
                    .roleName(rolename)
                    .policyArn("arn:aws:iam::aws:policy/
AmazonEventBridgeFullAccess")
                    .build();

            iam.attachRolePolicy(rolePolicyRequest);
            return response.role().arn();

        } catch (IamException e) {
```

```
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
        return "";
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - DeleteRule
  - DescribeRule
  - DisableRule
  - EnableRule
  - ListRuleNamesByTarget
  - ListRules
  - ListTargetsByRule
  - PutEvents
  - PutRule
  - PutTargets

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/*
 Before running this Kotlin code example, set up your development environment,
 including your credentials.

 For more information, see the following documentation topic:
 https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/setup.html
```

```
 This Kotlin example performs the following tasks with Amazon EventBridge:

 1. Creates an AWS Identity and Access Management (IAM) role to use with Amazon
 EventBridge.
 2. Creates an Amazon Simple Storage Service (Amazon S3) bucket with EventBridge
 events enabled.
 3. Creates a rule that triggers when an object is uploaded to Amazon S3.
 4. Lists rules on the event bus.
 5. Creates a new Amazon Simple Notification Service (Amazon SNS) topic and lets
 the user subscribe to it.
 6. Adds a target to the rule that sends an email to the specified topic.
 7. Creates an EventBridge event that sends an email when an Amazon S3 object is
 created.
 8. Lists targets.
 9. Lists the rules for the same target.
 10. Triggers the rule by uploading a file to the S3 bucket.
 11. Disables a specific rule.
 12. Checks and prints the state of the rule.
 13. Adds a transform to the rule to change the text of the email.
 14. Enables a specific rule.
 15. Triggers the updated rule by uploading a file to the S3 bucket.
 16. Updates the rule to a custom rule pattern.
 17. Sends an event to trigger the rule.
 18. Cleans up resources.
*/
val DASHES: String = String(CharArray(80)).replace("\u0000", "-")

suspend fun main(args: Array<String>) {
    val usage = """
    Usage:
        <roleName> <bucketName> <topicName> <eventRuleName>

    Where:
        roleName - The name of the role to create.
        bucketName - The Amazon Simple Storage Service (Amazon S3) bucket name to
 create.
        topicName - The name of the Amazon Simple Notification Service (Amazon
 SNS) topic to create.
        eventRuleName - The Amazon EventBridge rule name to create.
    """
    val polJSON =
        "{" +
            "\"Version\": \"2012-10-17\"," +
            "\"Statement\": [{" +
```

```
            "\"Effect\": \"Allow\"," +
            "\"Principal\": {" +
            "\"Service\": \"events.amazonaws.com\"" +
            "}," +
            "\"Action\": \"sts:AssumeRole\"" +
            "}]" +
            "}"

    if (args.size != 4) {
        println(usage)
        exitProcess(1)
    }

    val sc = Scanner(System.`in`)
    val roleName = args[0]
    val bucketName = args[1]
    val topicName = args[2]
    val eventRuleName = args[3]

    println(DASHES)
    println("Welcome to the Amazon EventBridge example scenario.")
    println(DASHES)

    println(DASHES)
    println("1. Create an AWS Identity and Access Management (IAM) role to use
with Amazon EventBridge.")
    val roleArn = createIAMRole(roleName, polJSON)
    println(DASHES)

    println(DASHES)
    println("2. Create an S3 bucket with EventBridge events enabled.")
    if (checkBucket(bucketName)) {
        println("$bucketName already exists. Ending this scenario.")
        exitProcess(1)
    }

    createBucket(bucketName)
    delay(3000)
    setBucketNotification(bucketName)
    println(DASHES)

    println(DASHES)
    println("3. Create a rule that triggers when an object is uploaded to Amazon
S3.")
```

```
    delay(10000)
    addEventRule(roleArn, bucketName, eventRuleName)
    println(DASHES)

    println(DASHES)
    println("4. List rules on the event bus.")
    listRules()
    println(DASHES)

    println(DASHES)
    println("5. Create a new SNS topic for testing and let the user subscribe to
the topic.")
    val topicArn = createSnsTopic(topicName)
    println(DASHES)

    println(DASHES)
    println("6. Add a target to the rule that sends an email to the specified
topic.")
    println("Enter your email to subscribe to the Amazon SNS topic:")
    val email = sc.nextLine()
    subEmail(topicArn, email)
    println("Use the link in the email you received to confirm your subscription.
Then press Enter to continue.")
    sc.nextLine()
    println(DASHES)

    println(DASHES)
    println("7. Create an EventBridge event that sends an email when an Amazon S3
object is created.")
    addSnsEventRule(eventRuleName, topicArn, topicName, eventRuleName,
bucketName)
    println(DASHES)

    println(DASHES)
    println("8. List targets.")
    listTargets(eventRuleName)
    println(DASHES)

    println(DASHES)
    println(" 9. List the rules for the same target.")
    listTargetRules(topicArn)
    println(DASHES)

    println(DASHES)
```

```
    println("10. Trigger the rule by uploading a file to the S3 bucket.")
    println("Press Enter to continue.")
    sc.nextLine()
    uploadTextFiletoS3(bucketName)
    println(DASHES)

    println(DASHES)
    println("11. Disable a specific rule.")
    changeRuleState(eventRuleName, false)
    println(DASHES)

    println(DASHES)
    println("12. Check and print the state of the rule.")
    checkRule(eventRuleName)
    println(DASHES)

    println(DASHES)
    println("13. Add a transform to the rule to change the text of the email.")
    updateSnsEventRule(topicArn, eventRuleName)
    println(DASHES)

    println(DASHES)
    println("14. Enable a specific rule.")
    changeRuleState(eventRuleName, true)
    println(DASHES)

    println(DASHES)
    println("15. Trigger the updated rule by uploading a file to the S3 bucket.")
    println("Press Enter to continue.")
    sc.nextLine()
    uploadTextFiletoS3(bucketName)
    println(DASHES)

    println(DASHES)
    println("16. Update the rule to a custom rule pattern.")
    updateToCustomRule(eventRuleName)
    println("Updated event rule $eventRuleName to use a custom pattern.")
    updateCustomRuleTargetWithTransform(topicArn, eventRuleName)
    println("Updated event target $topicArn.")
    println(DASHES)

    println(DASHES)
    println("17. Send an event to trigger the rule. This will trigger a
 subscription email.")
```

```
    triggerCustomRule(email)
    println("Events have been sent. Press Enter to continue.")
    sc.nextLine()
    println(DASHES)

    println(DASHES)
    println("18. Clean up resources.")
    println("Do you want to clean up resources (y/n)")
    val ans = sc.nextLine()
    if (ans.compareTo("y") == 0) {
        cleanupResources(topicArn, eventRuleName, bucketName, roleName)
    } else {
        println("The resources will not be cleaned up. ")
    }
    println(DASHES)

    println(DASHES)
    println("The Amazon EventBridge example scenario has successfully
 completed.")
    println(DASHES)
}

suspend fun cleanupResources(
    topicArn: String?,
    eventRuleName: String?,
    bucketName: String?,
    roleName: String?,
) {
    println("Removing all targets from the event rule.")
    deleteTargetsFromRule(eventRuleName)
    deleteRuleByName(eventRuleName)
    deleteSNSTopic(topicArn)
    deleteS3Bucket(bucketName)
    deleteRole(roleName)
}

suspend fun deleteRole(roleNameVal: String?) {
    val policyArnVal = "arn:aws:iam::aws:policy/AmazonEventBridgeFullAccess"
    val policyRequest =
        DetachRolePolicyRequest {
            policyArn = policyArnVal
            roleName = roleNameVal
        }
    IamClient.fromEnvironment { region = "us-east-1" }.use { iam ->
```

```
        iam.detachRolePolicy(policyRequest)
        println("Successfully detached policy $policyArnVal from role
 $roleNameVal")

        // Delete the role.
        val roleRequest =
            DeleteRoleRequest {
                roleName = roleNameVal
            }

        iam.deleteRole(roleRequest)
        println("*** Successfully deleted $roleNameVal")
    }
}

suspend fun deleteS3Bucket(bucketName: String?) {
    // Remove all the objects from the S3 bucket.
    val listObjects =
        ListObjectsRequest {
            bucket = bucketName
        }
    S3Client.fromEnvironment { region = "us-east-1" }.use { s3Client ->
        val res = s3Client.listObjects(listObjects)
        val myObjects = res.contents
        val toDelete = mutableListOf<ObjectIdentifier>()

        if (myObjects != null) {
            for (myValue in myObjects) {
                toDelete.add(
                    ObjectIdentifier {
                        key = myValue.key
                    },
                )
            }
        }

        val delOb =
            Delete {
                objects = toDelete
            }

        val dor =
            DeleteObjectsRequest {
                bucket = bucketName
```

```
                delete = delOb
            }
        s3Client.deleteObjects(dor)

        // Delete the S3 bucket.
        val deleteBucketRequest =
            DeleteBucketRequest {
                bucket = bucketName
            }
        s3Client.deleteBucket(deleteBucketRequest)
        println("You have deleted the bucket and the objects")
    }
}

// Delete the SNS topic.
suspend fun deleteSNSTopic(topicArnVal: String?) {
    val request =
        DeleteTopicRequest {
            topicArn = topicArnVal
        }

    SnsClient.fromEnvironment { region = "us-east-1" }.use { snsClient ->
        snsClient.deleteTopic(request)
        println(" $topicArnVal was deleted.")
    }
}

suspend fun deleteRuleByName(ruleName: String?) {
    val ruleRequest =
        DeleteRuleRequest {
            name = ruleName
        }
    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.deleteRule(ruleRequest)
        println("Successfully deleted the rule")
    }
}

suspend fun deleteTargetsFromRule(eventRuleName: String?) {
    // First, get all targets that will be deleted.
    val request =
        ListTargetsByRuleRequest {
            rule = eventRuleName
```

```
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.listTargetsByRule(request)
        val allTargets = response.targets

        // Get all targets and delete them.
        if (allTargets != null) {
            for (myTarget in allTargets) {
                val removeTargetsRequest =
                    RemoveTargetsRequest {
                        rule = eventRuleName
                        ids = listOf(myTarget.id.toString())
                    }
                eventBrClient.removeTargets(removeTargetsRequest)
                println("Successfully removed the target")
            }
        }
    }
}

suspend fun triggerCustomRule(email: String) {
    val json =
        "{" +
            "\"UserEmail\": \"" + email + "\"," +
            "\"Message\": \"This event was generated by example code.\"" +
            "\"UtcTime\": \"Now.\"" +
            "}"

    val entry =
        PutEventsRequestEntry {
            source = "ExampleSource"
            detail = json
            detailType = "ExampleType"
        }

    val eventsRequest =
        PutEventsRequest {
            this.entries = listOf(entry)
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
```

```
        eventBrClient.putEvents(eventsRequest)
    }
}

suspend fun updateCustomRuleTargetWithTransform(
    topicArn: String?,
    ruleName: String?,
) {
    val targetId = UUID.randomUUID().toString()

    val inputTransformerOb =
        InputTransformer {
            inputTemplate = "\"Notification: sample event was received.\""
        }

    val target =
        Target {
            id = targetId
            arn = topicArn
            inputTransformer = inputTransformerOb
        }

    val targetsRequest =
        PutTargetsRequest {
            rule = ruleName
            targets = listOf(target)
            eventBusName = null
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putTargets(targetsRequest)
    }
}

suspend fun updateToCustomRule(ruleName: String?) {
    val customEventsPattern =
        "{" +
            "\"source\": [\"ExampleSource\"]," +
            "\"detail-type\": [\"ExampleType\"]" +
            "}"
    val request =
        PutRuleRequest {
            name = ruleName
```

```
            description = "Custom test rule"
            eventPattern = customEventsPattern
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putRule(request)
    }
}

// Update an Amazon S3 object created rule with a transform on the target.
suspend fun updateSnsEventRule(
    topicArn: String?,
    ruleName: String?,
) {
    val targetId = UUID.randomUUID().toString()
    val myMap = mutableMapOf<String, String>()
    myMap["bucket"] = "$.detail.bucket.name"
    myMap["time"] = "$.time"

    val inputTransOb =
        InputTransformer {
            inputTemplate = "\"Notification: an object was uploaded to bucket
 <bucket> at <time>.\""
            inputPathsMap = myMap
        }
    val targetOb =
        Target {
            id = targetId
            arn = topicArn
            inputTransformer = inputTransOb
        }

    val targetsRequest =
        PutTargetsRequest {
            rule = ruleName
            targets = listOf(targetOb)
            eventBusName = null
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putTargets(targetsRequest)
    }
```

```
}

suspend fun checkRule(eventRuleName: String?) {
    val ruleRequest =
        DescribeRuleRequest {
            name = eventRuleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.describeRule(ruleRequest)
        println("The state of the rule is $response")
    }
}

suspend fun changeRuleState(
    eventRuleName: String,
    isEnabled: Boolean?,
) {
    if (!isEnabled!!) {
        println("Disabling the rule: $eventRuleName")
        val ruleRequest =
            DisableRuleRequest {
                name = eventRuleName
            }
        EventBridgeClient { region = "us-east-1" }.use { eventBrClient ->
            eventBrClient.disableRule(ruleRequest)
        }
    } else {
        println("Enabling the rule: $eventRuleName")
        val ruleRequest =
            EnableRuleRequest {
                name = eventRuleName
            }
        EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
            eventBrClient.enableRule(ruleRequest)
        }
    }
}

// Create and upload a file to an S3 bucket to trigger an event.
@Throws(IOException::class)
suspend fun uploadTextFiletoS3(bucketName: String?) {
```

```
    val fileSuffix = SimpleDateFormat("yyyyMMddHHmmss").format(Date())
    val fileName = "TextFile$fileSuffix.txt"
    val myFile = File(fileName)
    val fw = FileWriter(myFile.absoluteFile)
    val bw = BufferedWriter(fw)
    bw.write("This is a sample file for testing uploads.")
    bw.close()

    val putOb =
        PutObjectRequest {
            bucket = bucketName
            key = fileName
            body = myFile.asByteStream()
        }

    S3Client.fromEnvironment { region = "us-east-1" }.use { s3Client ->
        s3Client.putObject(putOb)
    }
}

suspend fun listTargetRules(topicArnVal: String?) {
    val ruleNamesByTargetRequest =
        ListRuleNamesByTargetRequest {
            targetArn = topicArnVal
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response =
 eventBrClient.listRuleNamesByTarget(ruleNamesByTargetRequest)
        response.ruleNames?.forEach { rule ->
            println("The rule name is $rule")
        }
    }
}

suspend fun listTargets(ruleName: String?) {
    val ruleRequest =
        ListTargetsByRuleRequest {
            rule = ruleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
```

```
        val response = eventBrClient.listTargetsByRule(ruleRequest)
        response.targets?.forEach { target ->
            println("Target ARN: ${target.arn}")
        }
    }
}

// Add a rule that triggers an SNS target when a file is uploaded to an S3
 bucket.
suspend fun addSnsEventRule(
    ruleName: String?,
    topicArn: String?,
    topicName: String,
    eventRuleName: String,
    bucketName: String,
) {
    val targetID = UUID.randomUUID().toString()
    val myTarget =
        Target {
            id = targetID
            arn = topicArn
        }

    val targetsOb = mutableListOf<Target>()
    targetsOb.add(myTarget)

    val request =
        PutTargetsRequest {
            eventBusName = null
            targets = targetsOb
            rule = ruleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putTargets(request)
        println("Added event rule $eventRuleName with Amazon SNS target
 $topicName for bucket $bucketName.")
    }
}

suspend fun subEmail(
    topicArnVal: String?,
    email: String?,
```

```
) {
    val request =
        SubscribeRequest {
            protocol = "email"
            endpoint = email
            returnSubscriptionArn = true
            topicArn = topicArnVal
        }

    SnsClient.fromEnvironment { region = "us-east-1" }.use { snsClient ->
        val result = snsClient.subscribe(request)
        println(" Subscription ARN: ${result.subscriptionArn}")
    }
}

suspend fun createSnsTopic(topicName: String): String? {
    val topicPolicy = """
    {
        "Version":"2012-10-17",
        "Statement": [
            {
                "Sid": "EventBridgePublishTopic",
                "Effect": "Allow",
                "Principal": {
                    "Service": "events.amazonaws.com"
                },
                "Resource": "*",
                "Action": "sns:Publish"
            }
        ]
    }
    """.trimIndent()

    val topicAttributes = mutableMapOf<String, String>()
    topicAttributes["Policy"] = topicPolicy

    val topicRequest =
        CreateTopicRequest {
            name = topicName
            attributes = topicAttributes
        }

    SnsClient.fromEnvironment { region = "us-east-1" }.use { snsClient ->
        val response = snsClient.createTopic(topicRequest)
```

```
        println("Added topic $topicName for email subscriptions.")
        return response.topicArn
    }
}

suspend fun listRules() {
    val rulesRequest =
        ListRulesRequest {
            eventBusName = "default"
            limit = 10
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.listRules(rulesRequest)
        response.rules?.forEach { rule ->
            println("The rule name is ${rule.name}")
            println("The rule ARN is ${rule.arn}")
        }
    }
}

// Create a new event rule that triggers when an Amazon S3 object is created in a
 bucket.
suspend fun addEventRule(
    roleArnVal: String?,
    bucketName: String,
    eventRuleName: String?,
) {
    val pattern = """
    {
        "source": ["aws.s3"],
        "detail-type": ["Object Created"],
        "detail": {
            "bucket": {
                "name": ["$bucketName"]
            }
        }
    }
    """.trimIndent()

    val ruleRequest =
        PutRuleRequest {
            description = "Created by using the AWS SDK for Kotlin"
```

```
            name = eventRuleName
            eventPattern = pattern
            roleArn = roleArnVal
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val ruleResponse = eventBrClient.putRule(ruleRequest)
        println("The ARN of the new rule is ${ruleResponse.ruleArn}")
    }
}

// Set the Amazon S3 bucket notification configuration.
suspend fun setBucketNotification(bucketName: String) {
    val eventBridgeConfig =
        EventBridgeConfiguration {
        }

    val configuration =
        NotificationConfiguration {
            eventBridgeConfiguration = eventBridgeConfig
        }

    val configurationRequest =
        PutBucketNotificationConfigurationRequest {
            bucket = bucketName
            notificationConfiguration = configuration
            skipDestinationValidation = true
        }

    S3Client.fromEnvironment { region = "us-east-1" }.use { s3Client ->
        s3Client.putBucketNotificationConfiguration(configurationRequest)
        println("Added bucket $bucketName with EventBridge events enabled.")
    }
}

// Create an S3 bucket using a waiter.
suspend fun createBucket(bucketName: String) {
    val request =
        CreateBucketRequest {
            bucket = bucketName
        }

    S3Client.fromEnvironment { region = "us-east-1" }.use { s3 ->
```

```
        s3.createBucket(request)
        s3.waitUntilBucketExists {
            bucket = bucketName
        }
        println("$bucketName is ready")
    }
}

suspend fun checkBucket(bucketName: String?): Boolean {
    try {
        // Determine if the S3 bucket exists.
        val headBucketRequest =
            HeadBucketRequest {
                bucket = bucketName
            }

        S3Client.fromEnvironment { region = "us-east-1" }.use { s3Client ->
            s3Client.headBucket(headBucketRequest)
            return true
        }
    } catch (e: S3Exception) {
        System.err.println(e.message)
    }
    return false
}

suspend fun createIAMRole(
    rolenameVal: String?,
    polJSON: String?,
): String? {
    val request =
        CreateRoleRequest {
            roleName = rolenameVal
            assumeRolePolicyDocument = polJSON
            description = "Created using the AWS SDK for Kotlin"
        }

    val rolePolicyRequest =
        AttachRolePolicyRequest {
            roleName = rolenameVal
            policyArn = "arn:aws:iam::aws:policy/AmazonEventBridgeFullAccess"
        }

    IamClient.fromEnvironment { region = "us-east-1" }.use { iam ->
```

```
        val response = iam.createRole(request)
        iam.attachRolePolicy(rolePolicyRequest)
        return response.role?.arn
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
  - DeleteRule
  - DescribeRule
  - DisableRule
  - EnableRule
  - ListRuleNamesByTarget
  - ListRules
  - ListTargetsByRule
  - PutEvents
  - PutRule
  - PutTargets

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Actions for EventBridge using AWS SDKs

The following code examples demonstrate how to perform individual EventBridge actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

These excerpts call the EventBridge API and are code excerpts from larger programs that must be run in context. You can see actions in context in Scenarios for EventBridge using AWS SDKs.

The following examples include only the most commonly used actions. For a complete list, see the Amazon EventBridge API Reference.

Examples

- Use DeleteRule with an AWS SDK or CLI

- Use DescribeRule with an AWS SDK or CLI
- Use DisableRule with an AWS SDK or CLI
- Use EnableRule with an AWS SDK or CLI
- Use ListRuleNamesByTarget with an AWS SDK or CLI
- Use ListRules with an AWS SDK or CLI
- Use ListTargetsByRule with an AWS SDK or CLI
- Use PutEvents with an AWS SDK or CLI
- Use PutRule with an AWS SDK or CLI
- Use PutTargets with an AWS SDK or CLI
- Use RemoveTargets with an AWS SDK or CLI

Use `DeleteRule` with an AWS SDK or CLI

The following code examples show how to use `DeleteRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Delete a rule by its name.

```
/// <summary>
/// Delete an event rule by name.
/// </summary>
/// <param name="ruleName">The name of the event rule.</param>
/// <returns>True if successful.</returns>
```

```
    public async Task<bool> DeleteRuleByName(string ruleName)
    {
        var response = await _amazonEventBridge.DeleteRuleAsync(
            new DeleteRuleRequest()
            {
                Name = ruleName
            });

        return response.HttpStatusCode == HttpStatusCode.OK;
    }
```

- For API details, see DeleteRule in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To delete a CloudWatch Events rule

This example deletes the rule named EC2InstanceStateChanges:

```
aws events delete-rule --name "EC2InstanceStateChanges"
```

- For API details, see DeleteRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void deleteRuleByName(EventBridgeClient eventBrClient, String
ruleName) {
       DeleteRuleRequest ruleRequest = DeleteRuleRequest.builder()
               .name(ruleName)
               .build();
```

```
        eventBrClient.deleteRule(ruleRequest);
        System.out.println("Successfully deleted the rule");
    }
```

- For API details, see DeleteRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteRuleByName(ruleName: String?) {
    val ruleRequest =
        DeleteRuleRequest {
            name = ruleName
        }
    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.deleteRule(ruleRequest)
        println("Successfully deleted the rule")
    }
}
```

- For API details, see DeleteRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeRule` with an AWS SDK or CLI

The following code examples show how to use `DescribeRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get the state of a rule using the rule description.

```
    /// <summary>
    /// Get the state for a rule by the rule name.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="eventBusName">The optional name of the event bus. If empty,
uses the default event bus.</param>
    /// <returns>The state of the rule.</returns>
    public async Task<RuleState> GetRuleStateByRuleName(string ruleName, string?
eventBusName = null)
    {
        var ruleResponse = await _amazonEventBridge.DescribeRuleAsync(
            new DescribeRuleRequest()
            {
                Name = ruleName,
                EventBusName = eventBusName
            });
        return ruleResponse.State;
    }
```

- For API details, see DescribeRule in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To display information about a CloudWatch Events rule

This example displays information about the rule named DailyLambdaFunction:

```
aws events describe-rule --name "DailyLambdaFunction"
```

- For API details, see DescribeRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void checkRule(EventBridgeClient eventBrClient, String
eventRuleName) {
       try {
           DescribeRuleRequest ruleRequest = DescribeRuleRequest.builder()
                   .name(eventRuleName)
                   .build();

           DescribeRuleResponse response =
eventBrClient.describeRule(ruleRequest);
           System.out.println("The state of the rule is " +
response.stateAsString());

       } catch (EventBridgeException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
           System.exit(1);
       }
   }
```

- For API details, see DescribeRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun checkRule(eventRuleName: String?) {
    val ruleRequest =
        DescribeRuleRequest {
            name = eventRuleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.describeRule(ruleRequest)
        println("The state of the rule is $response")
    }
}
```

- For API details, see DescribeRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DisableRule` with an AWS SDK or CLI

The following code examples show how to use `DisableRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Disable a rule by its rule name.

```
/// <summary>
/// Disable a particular rule on an event bus.
/// </summary
/// <param name="ruleName">The name of the rule.</param>
/// <returns>True if successful.</returns>
public async Task<bool> DisableRuleByName(string ruleName)
{
    var ruleResponse = await _amazonEventBridge.DisableRuleAsync(
        new DisableRuleRequest()
        {
            Name = ruleName
        });
    return ruleResponse.HttpStatusCode == HttpStatusCode.OK;
}
```

- For API details, see DisableRule in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To disable a CloudWatch Events rule

This example disables the rule named DailyLambdaFunction. The rule is not deleted:

```
aws events disable-rule --name "DailyLambdaFunction"
```

- For API details, see DisableRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Disable a rule by using its rule name.

```
   public static void changeRuleState(EventBridgeClient eventBrClient, String
eventRuleName, Boolean isEnabled) {
       try {
           if (!isEnabled) {
               System.out.println("Disabling the rule: " + eventRuleName);
               DisableRuleRequest ruleRequest = DisableRuleRequest.builder()
                       .name(eventRuleName)
                       .build();

               eventBrClient.disableRule(ruleRequest);
           } else {
               System.out.println("Enabling the rule: " + eventRuleName);
               EnableRuleRequest ruleRequest = EnableRuleRequest.builder()
                       .name(eventRuleName)
                       .build();
               eventBrClient.enableRule(ruleRequest);
           }

       } catch (EventBridgeException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
           System.exit(1);
       }
   }
```

- For API details, see DisableRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun changeRuleState(
    eventRuleName: String,
    isEnabled: Boolean?,
) {
    if (!isEnabled!!) {
        println("Disabling the rule: $eventRuleName")
        val ruleRequest =
            DisableRuleRequest {
                name = eventRuleName
            }
        EventBridgeClient { region = "us-east-1" }.use { eventBrClient ->
            eventBrClient.disableRule(ruleRequest)
        }
    } else {
        println("Enabling the rule: $eventRuleName")
        val ruleRequest =
            EnableRuleRequest {
                name = eventRuleName
            }
        EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
            eventBrClient.enableRule(ruleRequest)
        }
    }
}
```

- For API details, see DisableRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `EnableRule` with an AWS SDK or CLI

The following code examples show how to use `EnableRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Enable a rule by its rule name.

```
/// <summary>
/// Enable a particular rule on an event bus.
/// </summary>
/// <param name="ruleName">The name of the rule.</param>
/// <returns>True if successful.</returns>
public async Task<bool> EnableRuleByName(string ruleName)
{
    var ruleResponse = await _amazonEventBridge.EnableRuleAsync(
        new EnableRuleRequest()
        {
            Name = ruleName
        });
    return ruleResponse.HttpStatusCode == HttpStatusCode.OK;
}
```

- For API details, see EnableRule in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To enable a CloudWatch Events rule

This example enables the rule named DailyLambdaFunction, which had been previously disabled:

```
aws events enable-rule --name "DailyLambdaFunction"
```

- For API details, see EnableRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Enable a rule by using its rule name.

```
   public static void changeRuleState(EventBridgeClient eventBrClient, String
eventRuleName, Boolean isEnabled) {
       try {
           if (!isEnabled) {
               System.out.println("Disabling the rule: " + eventRuleName);
               DisableRuleRequest ruleRequest = DisableRuleRequest.builder()
                       .name(eventRuleName)
                       .build();

               eventBrClient.disableRule(ruleRequest);
           } else {
               System.out.println("Enabling the rule: " + eventRuleName);
               EnableRuleRequest ruleRequest = EnableRuleRequest.builder()
```

```
                        .name(eventRuleName)
                        .build();
                eventBrClient.enableRule(ruleRequest);
            }

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }
```

- For API details, see EnableRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun changeRuleState(
    eventRuleName: String,
    isEnabled: Boolean?,
) {
    if (!isEnabled!!) {
        println("Disabling the rule: $eventRuleName")
        val ruleRequest =
            DisableRuleRequest {
                name = eventRuleName
            }
        EventBridgeClient { region = "us-east-1" }.use { eventBrClient ->
            eventBrClient.disableRule(ruleRequest)
        }
    } else {
        println("Enabling the rule: $eventRuleName")
        val ruleRequest =
            EnableRuleRequest {
                name = eventRuleName
```

```
            }
        EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
            eventBrClient.enableRule(ruleRequest)
        }
    }
}
```

- For API details, see EnableRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListRuleNamesByTarget` with an AWS SDK or CLI

The following code examples show how to use `ListRuleNamesByTarget`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List all of the rule names using the target.

```
    /// <summary>
    /// List names of all rules matching a target.
    /// </summary>
    /// <param name="targetArn">The ARN of the target.</param>
```

```
    /// <returns>The list of rule names.</returns>
    public async Task<List<string>> ListAllRuleNamesByTarget(string targetArn)
    {
        var results = new List<string>();
        var request = new ListRuleNamesByTargetRequest()
        {
            TargetArn = targetArn
        };
        ListRuleNamesByTargetResponse response;
        do
        {
            response = await
_amazonEventBridge.ListRuleNamesByTargetAsync(request);
            results.AddRange(response.RuleNames);
            request.NextToken = response.NextToken;

        } while (response.NextToken is not null);

        return results;
    }
```

- For API details, see ListRuleNamesByTarget in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To display all the rules that have a specified target

This example displays all rules that have the Lambda function named "MyFunctionName" as the target:

```
aws events list-rule-names-by-target --target-arn "arn:aws:lambda:us-
east-1:123456789012:function:MyFunctionName"
```

- For API details, see ListRuleNamesByTarget in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List all of the rule names by using the target.

```
   public static void listTargetRules(EventBridgeClient eventBrClient, String
topicArn) {
       ListRuleNamesByTargetRequest ruleNamesByTargetRequest =
ListRuleNamesByTargetRequest.builder()
               .targetArn(topicArn)
               .build();

       ListRuleNamesByTargetResponse response =
eventBrClient.listRuleNamesByTarget(ruleNamesByTargetRequest);
       List<String> rules = response.ruleNames();
       for (String rule : rules) {
           System.out.println("The rule name is " + rule);
       }
   }
```

- For API details, see ListRuleNamesByTarget in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun listTargetRules(topicArnVal: String?) {
    val ruleNamesByTargetRequest =
        ListRuleNamesByTargetRequest {
            targetArn = topicArnVal
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response =
 eventBrClient.listRuleNamesByTarget(ruleNamesByTargetRequest)
        response.ruleNames?.forEach { rule ->
            println("The rule name is $rule")
        }
    }
}
```

- For API details, see ListRuleNamesByTarget in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListRules` with an AWS SDK or CLI

The following code examples show how to use `ListRules`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List all of the rules for an event bus.

```
    /// <summary>
    /// List the rules on an event bus.
    /// </summary>
    /// <param name="eventBusArn">The optional ARN of the event bus. If empty,
uses the default event bus.</param>
    /// <returns>The list of rules.</returns>
    public async Task<List<Rule>> ListAllRulesForEventBus(string? eventBusArn =
null)
    {
        var results = new List<Rule>();
        var request = new ListRulesRequest()
        {
            EventBusName = eventBusArn
        };
        // Get all of the pages of rules.
        ListRulesResponse response;
        do
        {
            response = await _amazonEventBridge.ListRulesAsync(request);
            results.AddRange(response.Rules);
            request.NextToken = response.NextToken;

        } while (response.NextToken is not null);

        return results;
    }
```

- For API details, see ListRules in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To display a list of all CloudWatch Events rules

This example displays all CloudWatch Events rules in the region:

```
aws events list-rules
```

To display a list of CloudWatch Events rules beginning with a certain string.

This example displays all CloudWatch Events rules in the region that have a name starting with "Daily":

```
aws events list-rules --name-prefix "Daily"
```

- For API details, see ListRules in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Enable a rule by using its rule name.

```
    public static void listRules(EventBridgeClient eventBrClient) {
        try {
            ListRulesRequest rulesRequest = ListRulesRequest.builder()
                    .eventBusName("default")
                    .limit(10)
                    .build();

            ListRulesResponse response = eventBrClient.listRules(rulesRequest);
            List<Rule> rules = response.rules();
            for (Rule rule : rules) {
                System.out.println("The rule name is : " + rule.name());
                System.out.println("The rule description is : " +
rule.description());
                System.out.println("The rule state is : " +
rule.stateAsString());
            }

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
```

```
        }
    }
```

- For API details, see ListRules in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun listRules() {
    val rulesRequest =
        ListRulesRequest {
            eventBusName = "default"
            limit = 10
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.listRules(rulesRequest)
        response.rules?.forEach { rule ->
            println("The rule name is ${rule.name}")
            println("The rule ARN is ${rule.arn}")
        }
    }
}
```

- For API details, see ListRules in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListTargetsByRule` with an AWS SDK or CLI

The following code examples show how to use `ListTargetsByRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List all of the targets for a rule using the rule name.

```
/// <summary>
/// List all of the targets matching a rule by name.
/// </summary>
/// <param name="ruleName">The name of the rule.</param>
/// <returns>The list of targets.</returns>
public async Task<List<Target>> ListAllTargetsOnRule(string ruleName)
{
    var results = new List<Target>();
    var request = new ListTargetsByRuleRequest()
    {
        Rule = ruleName
    };
    ListTargetsByRuleResponse response;
    do
    {
        response = await _amazonEventBridge.ListTargetsByRuleAsync(request);
        results.AddRange(response.Targets);
        request.NextToken = response.NextToken;

    } while (response.NextToken is not null);
```

```
        return results;
    }
```

- For API details, see ListTargetsByRule in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To display all the targets for a CloudWatch Events rule

This example displays all the targets of the rule named DailyLambdaFunction:

```
aws events list-targets-by-rule --rule  "DailyLambdaFunction"
```

- For API details, see ListTargetsByRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List all of the targets for a rule by using the rule name.

```
   public static void listTargets(EventBridgeClient eventBrClient, String
ruleName) {
       ListTargetsByRuleRequest ruleRequest = ListTargetsByRuleRequest.builder()
               .rule(ruleName)
               .build();

       ListTargetsByRuleResponse res =
eventBrClient.listTargetsByRule(ruleRequest);
       List<Target> targetsList = res.targets();
```

```
        for (Target target: targetsList) {
            System.out.println("Target ARN: "+target.arn());
        }
    }
```

- For API details, see ListTargetsByRule in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun listTargets(ruleName: String?) {
    val ruleRequest =
        ListTargetsByRuleRequest {
            rule = ruleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.listTargetsByRule(ruleRequest)
        response.targets?.forEach { target ->
            println("Target ARN: ${target.arn}")
        }
    }
}
```

- For API details, see ListTargetsByRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutEvents` with an AWS SDK or CLI

The following code examples show how to use `PutEvents`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Create and trigger a rule

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send an event that matches a custom pattern for a rule.

```
    /// <summary>
    /// Add an event to the event bus that includes an email, message, and time.
    /// </summary>
    /// <param name="email">The email to use in the event detail of the custom
event.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> PutCustomEmailEvent(string email)
    {
        var eventDetail = new
        {
            UserEmail = email,
            Message = "This event was generated by example code.",
            UtcTime = DateTime.UtcNow.ToString("g")
        };
        var response = await _amazonEventBridge.PutEventsAsync(
            new PutEventsRequest()
            {
                Entries = new List<PutEventsRequestEntry>()
                {
```

```
                    new PutEventsRequestEntry()
                    {
                        Source = "ExampleSource",
                        Detail = JsonSerializer.Serialize(eventDetail),
                        DetailType = "ExampleType"
                    }
                }
            });

        return response.FailedEntryCount == 0;
    }
```

- For API details, see PutEvents in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Include the required files.

```
#include <aws/core/Aws.h>
#include <aws/events/EventBridgeClient.h>
#include <aws/events/model/PutEventsRequest.h>
#include <aws/events/model/PutEventsResult.h>
#include <aws/core/utils/Outcome.h>
#include <iostream>
```

Send the event.

```
        Aws::CloudWatchEvents::EventBridgeClient cwe;

        Aws::CloudWatchEvents::Model::PutEventsRequestEntry event_entry;
        event_entry.SetDetail(MakeDetails(event_key, event_value));
```

```
        event_entry.SetDetailType("sampleSubmitted");
        event_entry.AddResources(resource_arn);
        event_entry.SetSource("aws-sdk-cpp-cloudwatch-example");

        Aws::CloudWatchEvents::Model::PutEventsRequest request;
        request.AddEntries(event_entry);

        auto outcome = cwe.PutEvents(request);
        if (!outcome.IsSuccess())
        {
            std::cout << "Failed to post CloudWatch event: " <<
                outcome.GetError().GetMessage() << std::endl;
        }
        else
        {
            std::cout << "Successfully posted CloudWatch event" << std::endl;
        }
```

- For API details, see PutEvents in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To send a custom event to CloudWatch Events

This example sends a custom event to CloudWatch Events. The event is contained within the putevents.json file:

```
aws events put-events --entries file://putevents.json
```

Here are the contents of the putevents.json file:

```
[
  {
    "Source": "com.mycompany.myapp",
    "Detail": "{ \"key1\": \"value1\", \"key2\": \"value2\" }",
    "Resources": [
      "resource1",
      "resource2"
    ],
```

```
    "DetailType": "myDetailType"
  },
  {
    "Source": "com.mycompany.myapp",
    "Detail": "{ \"key1\": \"value3\", \"key2\": \"value4\" }",
    "Resources": [
      "resource1",
      "resource2"
    ],
    "DetailType": "myDetailType"
   }
]
```

- For API details, see PutEvents in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void triggerCustomRule(EventBridgeClient eventBrClient, String
email) {
       String json = "{" +
               "\"UserEmail\": \"" + email + "\"," +
               "\"Message\": \"This event was generated by example code.\"," +
               "\"UtcTime\": \"Now.\"" +
               "}";

       PutEventsRequestEntry entry = PutEventsRequestEntry.builder()
               .source("ExampleSource")
               .detail(json)
               .detailType("ExampleType")
               .build();

       PutEventsRequest eventsRequest = PutEventsRequest.builder()
               .entries(entry)
               .build();
```

```
        eventBrClient.putEvents(eventsRequest);
    }
```

- For API details, see PutEvents in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Import the SDK and client modules and call the API.

```
import {
  EventBridgeClient,
  PutEventsCommand,
} from "@aws-sdk/client-eventbridge";

export const putEvents = async (
  source = "eventbridge.integration.test",
  detailType = "greeting",
  resources = [],
) => {
  const client = new EventBridgeClient({});

  const response = await client.send(
    new PutEventsCommand({
      Entries: [
        {
          Detail: JSON.stringify({ greeting: "Hello there." }),
          DetailType: detailType,
          Resources: resources,
          Source: source,
        },
      ],
    }),
```

```
  );

  console.log("PutEvents response:");
  console.log(response);
  // PutEvents response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '3d0df73d-dcea-4a23-ae0d-f5556a3ac109',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   Entries: [ { EventId: '51620841-5af4-6402-d9bc-b77734991eb5' } ],
  //   FailedEntryCount: 0
  // }

  return response;
};
```

- For API details, see PutEvents in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });

var params = {
  Entries: [
    {
```

```
      Detail: '{ "key1": "value1", "key2": "value2" }',
      DetailType: "appRequestSubmitted",
      Resources: ["RESOURCE_ARN"],
      Source: "com.company.app",
    },
  ],
};

ebevents.putEvents(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Entries);
  }
});
```

- For API details, see PutEvents in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun triggerCustomRule(email: String) {
    val json =
        "{" +
            "\"UserEmail\": \"" + email + "\"," +
            "\"Message\": \"This event was generated by example code.\"" +
            "\"UtcTime\": \"Now.\"" +
            "}"

    val entry =
        PutEventsRequestEntry {
            source = "ExampleSource"
            detail = json
            detailType = "ExampleType"
```

```
        }

    val eventsRequest =
        PutEventsRequest {
            this.entries = listOf(entry)
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putEvents(eventsRequest)
    }
}
```

- For API details, see PutEvents in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutRule` with an AWS SDK or CLI

The following code examples show how to use `PutRule`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Create and trigger a rule
- Send event notifications to EventBridge

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a rule that triggers when an object is added to an Amazon Simple Storage Service bucket.

```
    /// <summary>
    /// Create a new event rule that triggers when an Amazon S3 object is created
 in a bucket.
    /// </summary>
    /// <param name="roleArn">The ARN of the role.</param>
    /// <param name="ruleName">The name to give the rule.</param>
    /// <param name="bucketName">The name of the bucket to trigger the event.</
param>
    /// <returns>The ARN of the new rule.</returns>
    public async Task<string> PutS3UploadRule(string roleArn, string ruleName,
 string bucketName)
    {
        string eventPattern = "{" +
                                "\"source\": [\"aws.s3\"]," +
                                    "\"detail-type\": [\"Object Created\"]," +
                                    "\"detail\": {" +
                                        "\"bucket\": {" +
                                            "\"name\": [\"" + bucketName + "\"]"
 +
                                        "}" +
                                    "}" +
                               "}";

        var response = await _amazonEventBridge.PutRuleAsync(
            new PutRuleRequest()
            {
                Name = ruleName,
                Description = "Example S3 upload rule for EventBridge",
                RoleArn = roleArn,
                EventPattern = eventPattern
            });

        return response.RuleArn;
    }
```

Create a rule that uses a custom pattern.

```
    /// <summary>
    /// Update a rule to use a custom defined event pattern.
```

```
    /// </summary>
    /// <param name="ruleName">The name of the rule to update.</param>
    /// <returns>The ARN of the updated rule.</returns>
    public async Task<string> UpdateCustomEventPattern(string ruleName)
    {
        string customEventsPattern = "{" +
                                     "\"source\": [\"ExampleSource\"]," +
                                     "\"detail-type\": [\"ExampleType\"]" +
                                     "}";

        var response = await _amazonEventBridge.PutRuleAsync(
            new PutRuleRequest()
            {
                Name = ruleName,
                Description = "Custom test rule",
                EventPattern = customEventsPattern
            });

        return response.RuleArn;
    }
```

- For API details, see PutRule in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Include the required files.

```
#include <aws/core/Aws.h>
#include <aws/events/EventBridgeClient.h>
#include <aws/events/model/PutRuleRequest.h>
#include <aws/events/model/PutRuleResult.h>
#include <aws/core/utils/Outcome.h>
#include <iostream>
```

Create the rule.

```
        Aws::CloudWatchEvents::EventBridgeClient cwe;
        Aws::CloudWatchEvents::Model::PutRuleRequest request;
        request.SetName(rule_name);
        request.SetRoleArn(role_arn);
        request.SetScheduleExpression("rate(5 minutes)");
        request.SetState(Aws::CloudWatchEvents::Model::RuleState::ENABLED);

        auto outcome = cwe.PutRule(request);
        if (!outcome.IsSuccess())
        {
            std::cout << "Failed to create CloudWatch events rule " <<
                rule_name << ": " << outcome.GetError().GetMessage() <<
                std::endl;
        }
        else
        {
            std::cout << "Successfully created CloudWatch events rule " <<
                rule_name << " with resulting Arn " <<
                outcome.GetResult().GetRuleArn() << std::endl;
        }
```

- For API details, see PutRule in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To create CloudWatch Events rules

This example creates a rule that triggers every day at 9:00am (UTC). If you use put-targets to add a Lambda function as a target of this rule, you could run the Lambda function every day at the specified time:

```
aws events put-rule --name "DailyLambdaFunction" --schedule-expression "cron(0 9
 * * ? *)"
```

This example creates a rule that triggers when any EC2 instance in the region changes state:

```
aws events put-rule --name "EC2InstanceStateChanges" --event-pattern "{\"source
\":[\"aws.ec2\"],\"detail-type\":[\"EC2 Instance State-change Notification\"]}"
 --role-arn "arn:aws:iam::123456789012:role/MyRoleForThisRule"
```

This example creates a rule that triggers when any EC2 instance in the region is stopped or terminated:

```
aws events put-rule --name "EC2InstanceStateChangeStopOrTerminate" --event-
pattern "{\"source\":[\"aws.ec2\"],\"detail-type\":[\"EC2 Instance State-change
 Notification\"],\"detail\":{\"state\":[\"stopped\",\"terminated\"]}}" --role-
arn "arn:aws:iam::123456789012:role/MyRoleForThisRule"
```

- For API details, see PutRule in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a scheduled rule.

```
   public static void createEBRule(EventBridgeClient eventBrClient, String
ruleName, String cronExpression) {
       try {
           PutRuleRequest ruleRequest = PutRuleRequest.builder()
                   .name(ruleName)
                   .eventBusName("default")
                   .scheduleExpression(cronExpression)
                   .state("ENABLED")
                   .description("A test rule that runs on a schedule created by
the Java API")
                   .build();

           PutRuleResponse ruleResponse = eventBrClient.putRule(ruleRequest);
           System.out.println("The ARN of the new rule is " +
ruleResponse.ruleArn());
```

```
        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }
```

Create a rule that triggers when an object is added to an Amazon Simple Storage Service bucket.

```
    // Create a new event rule that triggers when an Amazon S3 object is created
in
    // a bucket.
    public static void addEventRule(EventBridgeClient eventBrClient, String
roleArn, String bucketName,
            String eventRuleName) {
        String pattern = "{\n" +
                "  \"source\": [\"aws.s3\"],\n" +
                "  \"detail-type\": [\"Object Created\"],\n" +
                "  \"detail\": {\n" +
                "    \"bucket\": {\n" +
                "      \"name\": [\"" + bucketName + "\"]\n" +
                "    }\n" +
                "  }\n" +
                "}";

        try {
            PutRuleRequest ruleRequest = PutRuleRequest.builder()
                    .description("Created by using the AWS SDK for Java v2")
                    .name(eventRuleName)
                    .eventPattern(pattern)
                    .roleArn(roleArn)
                    .build();

            PutRuleResponse ruleResponse = eventBrClient.putRule(ruleRequest);
            System.out.println("The ARN of the new rule is " +
ruleResponse.ruleArn());

        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
```

```
    }
```

- For API details, see PutRule in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Import the SDK and client modules and call the API.

```
import { EventBridgeClient, PutRuleCommand } from "@aws-sdk/client-eventbridge";

export const putRule = async (
  ruleName = "some-rule",
  source = "some-source",
) => {
  const client = new EventBridgeClient({});

  const response = await client.send(
    new PutRuleCommand({
      Name: ruleName,
      EventPattern: JSON.stringify({ source: [source] }),
      State: "ENABLED",
      EventBusName: "default",
    }),
  );

  console.log("PutRule response:");
  console.log(response);
  // PutRule response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'd7292ced-1544-421b-842f-596326bc7072',
  //     extendedRequestId: undefined,
```

```
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   RuleArn: 'arn:aws:events:us-east-1:xxxxxxxxxxxx:rule/
EventBridgeTestRule-1696280037720'
  // }
  return response;
};
```

- For API details, see PutRule in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });

var params = {
  Name: "DEMO_EVENT",
  RoleArn: "IAM_ROLE_ARN",
  ScheduleExpression: "rate(5 minutes)",
  State: "ENABLED",
};

ebevents.putRule(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.RuleArn);
  }
});
```

- For API details, see PutRule in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create a scheduled rule.

```
suspend fun createScRule(
    ruleName: String?,
    cronExpression: String?,
) {
    val ruleRequest =
        PutRuleRequest {
            name = ruleName
            eventBusName = "default"
            scheduleExpression = cronExpression
            state = RuleState.Enabled
            description = "A test rule that runs on a schedule created by the
 Kotlin API"
        }

    EventBridgeClient.fromEnvironment { region = "us-west-2" }.use
 { eventBrClient ->
        val ruleResponse = eventBrClient.putRule(ruleRequest)
        println("The ARN of the new rule is ${ruleResponse.ruleArn}")
    }
}
```

Create a rule that triggers when an object is added to an Amazon Simple Storage Service bucket.

```
// Create a new event rule that triggers when an Amazon S3 object is created in a
 bucket.
suspend fun addEventRule(
    roleArnVal: String?,
    bucketName: String,
    eventRuleName: String?,
) {
    val pattern = """
    {
        "source": ["aws.s3"],
        "detail-type": ["Object Created"],
        "detail": {
            "bucket": {
                "name": ["$bucketName"]
            }
        }
    }
    """.trimIndent()

    val ruleRequest =
        PutRuleRequest {
            description = "Created by using the AWS SDK for Kotlin"
            name = eventRuleName
            eventPattern = pattern
            roleArn = roleArnVal
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val ruleResponse = eventBrClient.putRule(ruleRequest)
        println("The ARN of the new rule is ${ruleResponse.ruleArn}")
    }
}
```

- For API details, see PutRule in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutTargets` with an AWS SDK or CLI

The following code examples show how to use `PutTargets`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code examples:

- Learn the basics
- Send event notifications to EventBridge

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Add an Amazon SNS topic as a target for a rule.

```
    /// <summary>
    /// Add an Amazon SNS target topic to a rule.
    /// </summary>
    /// <param name="ruleName">The name of the rule to update.</param>
    /// <param name="targetArn">The ARN of the Amazon SNS target.</param>
    /// <param name="eventBusArn">The optional event bus name, uses default if
empty.</param>
    /// <returns>The ID of the target.</returns>
    public async Task<string> AddSnsTargetToRule(string ruleName, string
targetArn, string? eventBusArn = null)
    {
        var targetID = Guid.NewGuid().ToString();

        // Create the list of targets and add a new target.
        var targets = new List<Target>
        {
            new Target()
            {
                Arn = targetArn,
```

```
                Id = targetID
            }
        };

        // Add the targets to the rule.
        var response = await _amazonEventBridge.PutTargetsAsync(
            new PutTargetsRequest()
            {
                EventBusName = eventBusArn,
                Rule = ruleName,
                Targets = targets,
            });

        if (response.FailedEntryCount > 0)
        {
            response.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to add target {e.TargetId}: {e.ErrorMessage}, code
{e.ErrorCode}");
            });
        }

        return targetID;
    }
```

Add an input transformer to a target for a rule.

```
    /// <summary>
    /// Update an Amazon S3 object created rule with a transform on the target.
    /// </summary>
    /// <param name="ruleName">The name of the rule.</param>
    /// <param name="targetArn">The ARN of the target.</param>
    /// <param name="eventBusArn">Optional event bus ARN. If empty, uses the
default event bus.</param>
    /// <returns>The ID of the target.</returns>
    public async Task<string> UpdateS3UploadRuleTargetWithTransform(string
ruleName, string targetArn, string? eventBusArn = null)
    {
        var targetID = Guid.NewGuid().ToString();

        var targets = new List<Target>
```

```
        {
            new Target()
            {
                Id = targetID,
                Arn = targetArn,
                InputTransformer = new InputTransformer()
                {
                    InputPathsMap = new Dictionary<string, string>()
                    {
                        {"bucket", "$.detail.bucket.name"},
                        {"time", "$.time"}
                    },
                    InputTemplate = "\"Notification: an object was uploaded to
bucket <bucket> at <time>.\""
                }
            }
        };
        var response = await _amazonEventBridge.PutTargetsAsync(
            new PutTargetsRequest()
            {
                EventBusName = eventBusArn,
                Rule = ruleName,
                Targets = targets,
            });
        if (response.FailedEntryCount > 0)
        {
            response.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to add target {e.TargetId}: {e.ErrorMessage}, code
{e.ErrorCode}");
            });
        }
        return targetID;
    }
```

- For API details, see PutTargets in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Include the required files.

```
#include <aws/core/Aws.h>
#include <aws/events/EventBridgeClient.h>
#include <aws/events/model/PutTargetsRequest.h>
#include <aws/events/model/PutTargetsResult.h>
#include <aws/core/utils/Outcome.h>
#include <iostream>
```

Add the target.

```
        Aws::CloudWatchEvents::EventBridgeClient cwe;

        Aws::CloudWatchEvents::Model::Target target;
        target.SetArn(lambda_arn);
        target.SetId(target_id);

        Aws::CloudWatchEvents::Model::PutTargetsRequest request;
        request.SetRule(rule_name);
        request.AddTargets(target);

        auto putTargetsOutcome = cwe.PutTargets(request);
        if (!putTargetsOutcome.IsSuccess())
        {
            std::cout << "Failed to create CloudWatch events target for rule "
                << rule_name << ": " <<
                putTargetsOutcome.GetError().GetMessage() << std::endl;
        }
        else
        {
            std::cout <<
```

```
                "Successfully created CloudWatch events target for rule "
                << rule_name << std::endl;
        }
```

- For API details, see PutTargets in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To add targets for CloudWatch Events rules

This example adds a Lambda function as the target of a rule:

```
aws events put-targets --rule DailyLambdaFunction --targets
 "Id"="1","Arn"="arn:aws:lambda:us-east-1:123456789012:function:MyFunctionName"
```

This example sets an Amazon Kinesis stream as the target, so that events caught by this rule are relayed to the stream:

```
aws events put-targets --rule EC2InstanceStateChanges --targets
 "Id"="1","Arn"="arn:aws:kinesis:us-east-1:123456789012:stream/
MyStream","RoleArn"="arn:aws:iam::123456789012:role/MyRoleForThisRule"
```

This example sets two Amazon Kinesis streams as targets for one rule:

```
aws events put-targets --rule DailyLambdaFunction --targets
 "Id"="Target1","Arn"="arn:aws:kinesis:us-east-1:379642911888:stream/
MyStream1","RoleArn"="arn:aws:iam::379642911888:role/ MyRoleToAccessLambda"
  "Id"="Target2"," Arn"="arn:aws:kinesis:us-east-1:379642911888:stream/
MyStream2","RoleArn"="arn:aws:iam::379642911888:role/MyRoleToAccessLambda"
```

- For API details, see PutTargets in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Add an Amazon SNS topic as a target for a rule.

```
   // Add a rule which triggers an SNS target when a file is uploaded to an S3
   // bucket.
   public static void addSnsEventRule(EventBridgeClient eventBrClient, String
ruleName, String topicArn,
           String topicName, String eventRuleName, String bucketName) {
       String targetID = java.util.UUID.randomUUID().toString();
       Target myTarget = Target.builder()
               .id(targetID)
               .arn(topicArn)
               .build();

       List<Target> targets = new ArrayList<>();
       targets.add(myTarget);
       PutTargetsRequest request = PutTargetsRequest.builder()
               .eventBusName(null)
               .targets(targets)
               .rule(ruleName)
               .build();

       eventBrClient.putTargets(request);
       System.out.println("Added event rule " + eventRuleName + " with Amazon
SNS target " + topicName + " for bucket "
               + bucketName + ".");
   }
```

Add an input transformer to a target for a rule.

```
   public static void updateCustomRuleTargetWithTransform(EventBridgeClient
eventBrClient, String topicArn,
```

```
            String ruleName) {
        String targetId = java.util.UUID.randomUUID().toString();
        InputTransformer inputTransformer = InputTransformer.builder()
                .inputTemplate("\"Notification: sample event was received.\"")
                .build();

        Target target = Target.builder()
                .id(targetId)
                .arn(topicArn)
                .inputTransformer(inputTransformer)
                .build();

        try {
            PutTargetsRequest targetsRequest = PutTargetsRequest.builder()
                    .rule(ruleName)
                    .targets(target)
                    .eventBusName(null)
                    .build();

            eventBrClient.putTargets(targetsRequest);
        } catch (EventBridgeException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }
```

- For API details, see PutTargets in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Import the SDK and client modules and call the API.

```
import {
```

```
  EventBridgeClient,
  PutTargetsCommand,
} from "@aws-sdk/client-eventbridge";

export const putTarget = async (
  existingRuleName = "some-rule",
  targetArn = "arn:aws:lambda:us-east-1:000000000000:function:test-func",
  uniqueId = Date.now().toString(),
) => {
  const client = new EventBridgeClient({});
  const response = await client.send(
    new PutTargetsCommand({
      Rule: existingRuleName,
      Targets: [
        {
          Arn: targetArn,
          Id: uniqueId,
        },
      ],
    }),
  );

  console.log("PutTargets response:");
  console.log(response);
  // PutTargets response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'f5b23b9a-2c17-45c1-ad5c-f926c3692e3d',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   FailedEntries: [],
  //   FailedEntryCount: 0
  // }

  return response;
};
```

- For API details, see PutTargets in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });

var params = {
  Rule: "DEMO_EVENT",
  Targets: [
    {
      Arn: "LAMBDA_FUNCTION_ARN",
      Id: "myEventBridgeTarget",
    },
  ],
};

ebevents.putTargets(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- For API details, see PutTargets in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Add a rule that triggers an SNS target when a file is uploaded to an S3
 bucket.
suspend fun addSnsEventRule(
    ruleName: String?,
    topicArn: String?,
    topicName: String,
    eventRuleName: String,
    bucketName: String,
) {
    val targetID = UUID.randomUUID().toString()
    val myTarget =
        Target {
            id = targetID
            arn = topicArn
        }

    val targetsOb = mutableListOf<Target>()
    targetsOb.add(myTarget)

    val request =
        PutTargetsRequest {
            eventBusName = null
            targets = targetsOb
            rule = ruleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putTargets(request)
        println("Added event rule $eventRuleName with Amazon SNS target
 $topicName for bucket $bucketName.")
    }
```

```
}
```

Add an input transformer to a target for a rule.

```
suspend fun updateCustomRuleTargetWithTransform(
    topicArn: String?,
    ruleName: String?,
) {
    val targetId = UUID.randomUUID().toString()

    val inputTransformerOb =
        InputTransformer {
            inputTemplate = "\"Notification: sample event was received.\""
        }

    val target =
        Target {
            id = targetId
            arn = topicArn
            inputTransformer = inputTransformerOb
        }

    val targetsRequest =
        PutTargetsRequest {
            rule = ruleName
            targets = listOf(target)
            eventBusName = null
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        eventBrClient.putTargets(targetsRequest)
    }
}
```

- For API details, see PutTargets in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `RemoveTargets` with an AWS SDK or CLI

The following code examples show how to use `RemoveTargets`.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Remove all of the targets for a rule using the rule name.

```
    /// <summary>
    /// Delete an event rule by name.
    /// </summary>
    /// <param name="ruleName">The name of the event rule.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> RemoveAllTargetsFromRule(string ruleName)
    {
        var targetIds = new List<string>();
        var request = new ListTargetsByRuleRequest()
        {
            Rule = ruleName
        };
        ListTargetsByRuleResponse targetsResponse;
        do
        {
            targetsResponse = await
_amazonEventBridge.ListTargetsByRuleAsync(request);
            targetIds.AddRange(targetsResponse.Targets.Select(t => t.Id));
            request.NextToken = targetsResponse.NextToken;

        } while (targetsResponse.NextToken is not null);

        var removeResponse = await _amazonEventBridge.RemoveTargetsAsync(
            new RemoveTargetsRequest()
            {
                Rule = ruleName,
                Ids = targetIds
```

```
            });

        if (removeResponse.FailedEntryCount > 0)
        {
            removeResponse.FailedEntries.ForEach(e =>
            {
                _logger.LogError(
                    $"Failed to remove target {e.TargetId}: {e.ErrorMessage},
code {e.ErrorCode}");
            });
        }

        return removeResponse.HttpStatusCode == HttpStatusCode.OK;
    }
```

- For API details, see RemoveTargets in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To remove a target for an event

This example removes the Amazon Kinesis stream named MyStream1 from being a target of the rule DailyLambdaFunction. When DailyLambdaFunction was created, this stream was set as a target with an ID of Target1:

```
aws events remove-targets --rule "DailyLambdaFunction" --ids "Target1"
```

- For API details, see RemoveTargets in *AWS CLI Command Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Remove all of the targets for a rule by using the rule name.

```
   public static void deleteTargetsFromRule(EventBridgeClient eventBrClient,
String eventRuleName) {
       // First, get all targets that will be deleted.
       ListTargetsByRuleRequest request = ListTargetsByRuleRequest.builder()
               .rule(eventRuleName)
               .build();

       ListTargetsByRuleResponse response =
eventBrClient.listTargetsByRule(request);
       List<Target> allTargets = response.targets();

       // Get all targets and delete them.
       for (Target myTarget : allTargets) {
           RemoveTargetsRequest removeTargetsRequest =
RemoveTargetsRequest.builder()
                   .rule(eventRuleName)
                   .ids(myTarget.id())
                   .build();

           eventBrClient.removeTargets(removeTargetsRequest);
           System.out.println("Successfully removed the target");
       }
   }
```

- For API details, see RemoveTargets in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteTargetsFromRule(eventRuleName: String?) {
    // First, get all targets that will be deleted.
```

```
    val request =
        ListTargetsByRuleRequest {
            rule = eventRuleName
        }

    EventBridgeClient.fromEnvironment { region = "us-east-1" }.use
 { eventBrClient ->
        val response = eventBrClient.listTargetsByRule(request)
        val allTargets = response.targets

        // Get all targets and delete them.
        if (allTargets != null) {
            for (myTarget in allTargets) {
                val removeTargetsRequest =
                    RemoveTargetsRequest {
                        rule = eventRuleName
                        ids = listOf(myTarget.id.toString())
                    }
                eventBrClient.removeTargets(removeTargetsRequest)
                println("Successfully removed the target")
            }
        }
    }
}
```

- For API details, see RemoveTargets in *AWS SDK for Kotlin API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for EventBridge using AWS SDKs

The following code examples show you how to implement common scenarios in EventBridge with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within EventBridge or combined with other AWS services. Each scenario includes a link to the complete source code, where you can find instructions on how to set up and run the code.

Scenarios target an intermediate level of experience to help you understand service actions in context.

Examples

- Create and trigger a rule in Amazon EventBridge using an AWS SDK
- Send S3 event notifications to Amazon EventBridge using an AWS SDK
- Use scheduled events to invoke a Lambda function

Create and trigger a rule in Amazon EventBridge using an AWS SDK

The following code example shows how to create and trigger a rule in Amazon EventBridge.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Call the functions in the correct order.

```
require 'aws-sdk-sns'
require 'aws-sdk-iam'
require 'aws-sdk-cloudwatchevents'
require 'aws-sdk-ec2'
require 'aws-sdk-cloudwatch'
require 'aws-sdk-cloudwatchlogs'
require 'securerandom'
```

Checks whether the specified Amazon Simple Notification Service (Amazon SNS) topic exists among those provided to this function.

```
# Checks whether the specified Amazon SNS
# topic exists among those provided to this function.
# This is a helper function that is called by the topic_exists? function.
#
# @param topics [Array] An array of Aws::SNS::Types::Topic objects.
# @param topic_arn [String] The ARN of the topic to find.
```

```
# @return [Boolean] true if the topic ARN was found; otherwise, false.
# @example
#   sns_client = Aws::SNS::Client.new(region: 'us-east-1')
#   response = sns_client.list_topics
#   if topic_found?(
#     response.topics,
#     'arn:aws:sns:us-east-1:111111111111:aws-doc-sdk-examples-topic'
#   )
#     puts 'Topic found.'
#   end
def topic_found?(topics, topic_arn)
  topics.each do |topic|
    return true if topic.topic_arn == topic_arn
  end
  false
end
```

Checks whether the specified topic exists among those available to the caller in Amazon SNS.

```
# Checks whether the specified topic exists among those available to the
# caller in Amazon SNS.
#
# @param sns_client [Aws::SNS::Client] An initialized Amazon SNS client.
# @param topic_arn [String] The ARN of the topic to find.
# @return [Boolean] true if the topic ARN was found; otherwise, false.
# @example
#   exit 1 unless topic_exists?(
#     Aws::SNS::Client.new(region: 'us-east-1'),
#     'arn:aws:sns:us-east-1:111111111111:aws-doc-sdk-examples-topic'
#   )
def topic_exists?(sns_client, topic_arn)
  puts "Searching for topic with ARN '#{topic_arn}'..."
  response = sns_client.list_topics
  if response.topics.count.positive?
    if topic_found?(response.topics, topic_arn)
      puts 'Topic found.'
      return true
    end
    while response.next_page?
      response = response.next_page
      next unless response.topics.count.positive?
```

```
      if topic_found?(response.topics, topic_arn)
        puts 'Topic found.'
        return true
      end
    end
  end
  puts 'Topic not found.'
  false
rescue StandardError => e
  puts "Topic not found: #{e.message}"
  false
end
```

Create a topic in Amazon SNS and then subscribe an email address to receive notifications to that topic.

```
# Creates a topic in Amazon SNS
# and then subscribes an email address to receive notifications to that topic.
#
# @param sns_client [Aws::SNS::Client] An initialized Amazon SNS client.
# @param topic_name [String] The name of the topic to create.
# @param email_address [String] The email address of the recipient to notify.
# @return [String] The ARN of the topic that was created.
# @example
#   puts create_topic(
#     Aws::SNS::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-topic',
#     'mary@example.com'
#   )
def create_topic(sns_client, topic_name, email_address)
  puts "Creating the topic named '#{topic_name}'..."
  topic_response = sns_client.create_topic(name: topic_name)
  puts "Topic created with ARN '#{topic_response.topic_arn}'."
  subscription_response = sns_client.subscribe(
    topic_arn: topic_response.topic_arn,
    protocol: 'email',
    endpoint: email_address,
    return_subscription_arn: true
  )
  puts 'Subscription created with ARN ' \
    "'#{subscription_response.subscription_arn}'. Have the owner of the " \
```

```
    "email address '#{email_address}' check their inbox in a few minutes " \
    'and confirm the subscription to start receiving notification emails.'
  topic_response.topic_arn
rescue StandardError => e
  puts "Error creating or subscribing to topic: #{e.message}"
  'Error'
end
```

Check whether the specified AWS Identity and Access Management (IAM) role exists among those provided to this function.

```
# Checks whether the specified AWS Identity and Access Management (IAM)
# role exists among those provided to this function.
# This is a helper function that is called by the role_exists? function.
#
# @param roles [Array] An array of Aws::IAM::Role objects.
# @param role_arn [String] The ARN of the role to find.
# @return [Boolean] true if the role ARN was found; otherwise, false.
# @example
#   iam_client = Aws::IAM::Client.new(region: 'us-east-1')
#   response = iam_client.list_roles
#   if role_found?(
#     response.roles,
#     'arn:aws:iam::111111111111:role/aws-doc-sdk-examples-ec2-state-change'
#   )
#     puts 'Role found.'
#   end
def role_found?(roles, role_arn)
  roles.each do |role|
    return true if role.arn == role_arn
  end
  false
end
```

Check whether the specified role exists among those available to the caller in IAM.

```
# Checks whether the specified role exists among those available to the
# caller in AWS Identity and Access Management (IAM).
#
# @param iam_client [Aws::IAM::Client] An initialized IAM client.
# @param role_arn [String] The ARN of the role to find.
```

```
# @return [Boolean] true if the role ARN was found; otherwise, false.
# @example
#   exit 1 unless role_exists?(
#     Aws::IAM::Client.new(region: 'us-east-1'),
#     'arn:aws:iam::111111111111:role/aws-doc-sdk-examples-ec2-state-change'
#   )
def role_exists?(iam_client, role_arn)
  puts "Searching for role with ARN '#{role_arn}'..."
  response = iam_client.list_roles
  if response.roles.count.positive?
    if role_found?(response.roles, role_arn)
      puts 'Role found.'
      return true
    end
    while response.next_page?
      response = response.next_page
      next unless response.roles.count.positive?

      if role_found?(response.roles, role_arn)
        puts 'Role found.'
        return true
      end
    end
  end
  puts 'Role not found.'
  false
rescue StandardError => e
  puts "Role not found: #{e.message}"
  false
end
```

Create a role in IAM.

```
# Creates a role in AWS Identity and Access Management (IAM).
# This role is used by a rule in Amazon EventBridge to allow
# that rule to operate within the caller's account.
# This role is designed to be used specifically by this code example.
#
# @param iam_client [Aws::IAM::Client] An initialized IAM client.
# @param role_name [String] The name of the role to create.
# @return [String] The ARN of the role that was created.
# @example
```

```
#   puts create_role(
#     Aws::IAM::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-ec2-state-change'
#   )
def create_role(iam_client, role_name)
  puts "Creating the role named '#{role_name}'..."
  response = iam_client.create_role(
    assume_role_policy_document: {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Sid': '',
          'Effect': 'Allow',
          'Principal': {
            'Service': 'events.amazonaws.com'
          },
          'Action': 'sts:AssumeRole'
        }
      ]
    }.to_json,
    path: '/',
    role_name: role_name
  )
  puts "Role created with ARN '#{response.role.arn}'."
  puts 'Adding access policy to role...'
  iam_client.put_role_policy(
    policy_document: {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Sid': 'CloudWatchEventsFullAccess',
          'Effect': 'Allow',
          'Resource': '*',
          'Action': 'events:*'
        },
        {
          'Sid': 'IAMPassRoleForCloudWatchEvents',
          'Effect': 'Allow',
          'Resource': 'arn:aws:iam::*:role/AWS_Events_Invoke_Targets',
          'Action': 'iam:PassRole'
        }
      ]
    }.to_json,
    policy_name: 'CloudWatchEventsPolicy',
```

```
    role_name: role_name
  )
  puts 'Access policy added to role.'
  response.role.arn
rescue StandardError => e
  puts "Error creating role or adding policy to it: #{e.message}"
  puts 'If the role was created, you must add the access policy ' \
    'to the role yourself, or delete the role yourself and try again.'
  'Error'
end
```

Checks whether the specified EventBridge rule exists among those provided to this function.

```
# Checks whether the specified Amazon EventBridge rule exists among
# those provided to this function.
# This is a helper function that is called by the rule_exists? function.
#
# @param rules [Array] An array of Aws::CloudWatchEvents::Types::Rule objects.
# @param rule_arn [String] The name of the rule to find.
# @return [Boolean] true if the name of the rule was found; otherwise, false.
# @example
#   cloudwatchevents_client = Aws::CloudWatch::Client.new(region: 'us-east-1')
#   response = cloudwatchevents_client.list_rules
#   if rule_found?(response.rules, 'aws-doc-sdk-examples-ec2-state-change')
#     puts 'Rule found.'
#   end
def rule_found?(rules, rule_name)
  rules.each do |rule|
    return true if rule.name == rule_name
  end
  false
end
```

Checks whether the specified rule exists among those available to the caller in EventBridge.

```
# Checks whether the specified rule exists among those available to the
# caller in Amazon EventBridge.
#
# @param cloudwatchevents_client [Aws::CloudWatchEvents::Client]
#   An initialized Amazon EventBridge client.
# @param rule_name [String] The name of the rule to find.
```

```
# @return [Boolean] true if the rule name was found; otherwise, false.
# @example
#   exit 1 unless rule_exists?(
#     Aws::CloudWatch::Client.new(region: 'us-east-1')
#     'aws-doc-sdk-examples-ec2-state-change'
#   )
def rule_exists?(cloudwatchevents_client, rule_name)
  puts "Searching for rule with name '#{rule_name}'..."
  response = cloudwatchevents_client.list_rules
  if response.rules.count.positive?
    if rule_found?(response.rules, rule_name)
      puts 'Rule found.'
      return true
    end
    while response.next_page?
      response = response.next_page
      next unless response.rules.count.positive?

      if rule_found?(response.rules, rule_name)
        puts 'Rule found.'
        return true
      end
    end
  end
  puts 'Rule not found.'
  false
rescue StandardError => e
  puts "Rule not found: #{e.message}"
  false
end
```

Create a rule in EventBridge.

```
# Creates a rule in Amazon EventBridge.
# This rule is triggered whenever an available instance in
# Amazon EC2 changes to the specified state.
# This rule is designed to be used specifically by this code example.
#
# Prerequisites:
#
# - A role in AWS Identity and Access Management (IAM) that is designed
#   to be used specifically by this code example.
```

```
# - A topic in Amazon SNS.
#
# @param cloudwatchevents_client [Aws::CloudWatchEvents::Client]
#   An initialized Amazon EventBridge client.
# @param rule_name [String] The name of the rule to create.
# @param rule_description [String] Some description for this rule.
# @param instance_state [String] The state that available instances in
#   Amazon EC2 must change to, to
#   trigger this rule.
# @param role_arn [String] The Amazon Resource Name (ARN) of the IAM role.
# @param target_id [String] Some identifying string for the rule's target.
# @param topic_arn [String] The ARN of the Amazon SNS topic.
# @return [Boolean] true if the rule was created; otherwise, false.
# @example
#   exit 1 unless rule_created?(
#     Aws::CloudWatch::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-ec2-state-change',
#     'Triggers when any available EC2 instance starts.',
#     'running',
#     'arn:aws:iam::111111111111:role/aws-doc-sdk-examples-ec2-state-change',
#     'sns-topic',
#     'arn:aws:sns:us-east-1:111111111111:aws-doc-sdk-examples-topic'
#   )
def rule_created?(
  cloudwatchevents_client,
  rule_name,
  rule_description,
  instance_state,
  role_arn,
  target_id,
  topic_arn
)
  puts "Creating rule with name '#{rule_name}'..."
  put_rule_response = cloudwatchevents_client.put_rule(
    name: rule_name,
    description: rule_description,
    event_pattern: {
      'source': [
        'aws.ec2'
      ],
      'detail-type': [
        'EC2 Instance State-change Notification'
      ],
      'detail': {
```

```
        'state': [
          instance_state
        ]
      }
    }.to_json,
    state: 'ENABLED',
    role_arn: role_arn
  )
  puts "Rule created with ARN '#{put_rule_response.rule_arn}'."

  put_targets_response = cloudwatchevents_client.put_targets(
    rule: rule_name,
    targets: [
      {
        id: target_id,
        arn: topic_arn
      }
    ]
  )
  if put_targets_response.key?(:failed_entry_count) &&
     put_targets_response.failed_entry_count.positive?
    puts 'Error(s) adding target to rule:'
    put_targets_response.failed_entries.each do |failure|
      puts failure.error_message
    end
    false
  else
    true
  end
rescue StandardError => e
  puts "Error creating rule or adding target to rule: #{e.message}"
  puts 'If the rule was created, you must add the target ' \
    'to the rule yourself, or delete the rule yourself and try again.'
  false
end
```

Check to see whether the specified log group exists among those available to the caller in Amazon CloudWatch Logs.

```
# Checks to see whether the specified log group exists among those available
# to the caller in Amazon CloudWatch Logs.
#
```

```
# @param cloudwatchlogs_client [Aws::CloudWatchLogs::Client] An initialized
#   Amazon CloudWatch Logs client.
# @param log_group_name [String] The name of the log group to find.
# @return [Boolean] true if the log group name was found; otherwise, false.
# @example
#   exit 1 unless log_group_exists?(
#     Aws::CloudWatchLogs::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-cloudwatch-log'
#   )
def log_group_exists?(cloudwatchlogs_client, log_group_name)
  puts "Searching for log group with name '#{log_group_name}'..."
  response = cloudwatchlogs_client.describe_log_groups(
    log_group_name_prefix: log_group_name
  )
  if response.log_groups.count.positive?
    response.log_groups.each do |log_group|
      if log_group.log_group_name == log_group_name
        puts 'Log group found.'
        return true
      end
    end
  end
  puts 'Log group not found.'
  false
rescue StandardError => e
  puts "Log group not found: #{e.message}"
  false
end
```

Create a log group in CloudWatch Logs.

```
# Creates a log group in Amazon CloudWatch Logs.
#
# @param cloudwatchlogs_client [Aws::CloudWatchLogs::Client] An initialized
#   Amazon CloudWatch Logs client.
# @param log_group_name [String] The name of the log group to create.
# @return [Boolean] true if the log group name was created; otherwise, false.
# @example
#   exit 1 unless log_group_created?(
#     Aws::CloudWatchLogs::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-cloudwatch-log'
#   )
```

```
def log_group_created?(cloudwatchlogs_client, log_group_name)
  puts "Attempting to create log group with the name '#{log_group_name}'..."
  cloudwatchlogs_client.create_log_group(log_group_name: log_group_name)
  puts 'Log group created.'
  true
rescue StandardError => e
  puts "Error creating log group: #{e.message}"
  false
end
```

Write an event to a log stream in CloudWatch Logs.

```
# Writes an event to a log stream in Amazon CloudWatch Logs.
#
# Prerequisites:
#
# - A log group in Amazon CloudWatch Logs.
# - A log stream within the log group.
#
# @param cloudwatchlogs_client [Aws::CloudWatchLogs::Client] An initialized
#   Amazon CloudWatch Logs client.
# @param log_group_name [String] The name of the log group.
# @param log_stream_name [String] The name of the log stream within
#   the log group.
# @param message [String] The message to write to the log stream.
# @param sequence_token [String] If available, the sequence token from the
#   message that was written immediately before this message. This sequence
#   token is returned by Amazon CloudWatch Logs whenever you programmatically
#   write a message to the log stream.
# @return [String] The sequence token that is returned by
#   Amazon CloudWatch Logs after successfully writing the message to the
#   log stream.
# @example
#   puts log_event(
#     Aws::EC2::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-cloudwatch-log'
#     '2020/11/19/53f985be-199f-408e-9a45-fc242df41fEX',
#     "Instance 'i-033c48ef067af3dEX' restarted.",
#     '495426724868310740095796045676567882148068632824696073EX'
#   )
def log_event(
  cloudwatchlogs_client,
```

```
  log_group_name,
  log_stream_name,
  message,
  sequence_token
)
  puts "Attempting to log '#{message}' to log stream '#{log_stream_name}'..."
  event = {
    log_group_name: log_group_name,
    log_stream_name: log_stream_name,
    log_events: [
      {
        timestamp: (Time.now.utc.to_f.round(3) * 1_000).to_i,
        message: message
      }
    ]
  }
  event[:sequence_token] = sequence_token unless sequence_token.empty?

  response = cloudwatchlogs_client.put_log_events(event)
  puts 'Message logged.'
  response.next_sequence_token
rescue StandardError => e
  puts "Message not logged: #{e.message}"
end
```

Restart an Amazon Elastic Compute Cloud (Amazon EC2) instance and adds information about the related activity to a log stream in CloudWatch Logs.

```
# Restarts an Amazon EC2 instance
# and adds information about the related activity to a log stream
# in Amazon CloudWatch Logs.
#
# Prerequisites:
#
# - The Amazon EC2 instance to restart.
# - The log group in Amazon CloudWatch Logs to add related activity
#   information to.
#
# @param ec2_client [Aws::EC2::Client] An initialized Amazon EC2 client.
# @param cloudwatchlogs_client [Aws::CloudWatchLogs::Client]
#   An initialized Amazon CloudWatch Logs client.
# @param instance_id [String] The ID of the instance.
```

```
# @param log_group_name [String] The name of the log group.
# @return [Boolean] true if the instance was restarted and the information
#   was written to the log stream; otherwise, false.
# @example
#   exit 1 unless instance_restarted?(
#     Aws::EC2::Client.new(region: 'us-east-1'),
#     Aws::CloudWatchLogs::Client.new(region: 'us-east-1'),
#     'i-033c48ef067af3dEX',
#     'aws-doc-sdk-examples-cloudwatch-log'
#   )
def instance_restarted?(
  ec2_client,
  cloudwatchlogs_client,
  instance_id,
  log_group_name
)
  log_stream_name = "#{Time.now.year}/#{Time.now.month}/#{Time.now.day}/" \
    "#{SecureRandom.uuid}"
  cloudwatchlogs_client.create_log_stream(
    log_group_name: log_group_name,
    log_stream_name: log_stream_name
  )
  sequence_token = ''

  puts "Attempting to stop the instance with the ID '#{instance_id}'. " \
    'This might take a few minutes...'
  ec2_client.stop_instances(instance_ids: [instance_id])
  ec2_client.wait_until(:instance_stopped, instance_ids: [instance_id])
  puts 'Instance stopped.'
  sequence_token = log_event(
    cloudwatchlogs_client,
    log_group_name,
    log_stream_name,
    "Instance '#{instance_id}' stopped.",
    sequence_token
  )

  puts 'Attempting to restart the instance. This might take a few minutes...'
  ec2_client.start_instances(instance_ids: [instance_id])
  ec2_client.wait_until(:instance_running, instance_ids: [instance_id])
  puts 'Instance restarted.'
  sequence_token = log_event(
    cloudwatchlogs_client,
    log_group_name,
```

```
    log_stream_name,
    "Instance '#{instance_id}' restarted.",
    sequence_token
  )

  true
rescue StandardError => e
  puts 'Error creating log stream or stopping or restarting the instance: ' \
    "#{e.message}"
  log_event(
    cloudwatchlogs_client,
    log_group_name,
    log_stream_name,
    "Error stopping or starting instance '#{instance_id}': #{e.message}",
    sequence_token
  )
  false
end
```

Display information about activity for a rule in EventBridge.

```
# Displays information about activity for a rule in Amazon EventBridge.
#
# Prerequisites:
#
# - A rule in Amazon EventBridge.
#
# @param cloudwatch_client [Amazon::CloudWatch::Client] An initialized
#   Amazon CloudWatch client.
# @param rule_name [String] The name of the rule.
# @param start_time [Time] The timestamp that determines the first datapoint
#   to return. Can also be expressed as DateTime, Date, Integer, or String.
# @param end_time [Time] The timestamp that determines the last datapoint
#   to return. Can also be expressed as DateTime, Date, Integer, or String.
# @param period [Integer] The interval, in seconds, to check for activity.
# @example
#   display_rule_activity(
#     Aws::CloudWatch::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-ec2-state-change',
#     Time.now - 600, # Start checking from 10 minutes ago.
#     Time.now, # Check up until now.
#     60 # Check every minute during those 10 minutes.
```

```
#   )
def display_rule_activity(
  cloudwatch_client,
  rule_name,
  start_time,
  end_time,
  period
)
  puts 'Attempting to display rule activity...'
  response = cloudwatch_client.get_metric_statistics(
    namespace: 'AWS/Events',
    metric_name: 'Invocations',
    dimensions: [
      {
        name: 'RuleName',
        value: rule_name
      }
    ],
    start_time: start_time,
    end_time: end_time,
    period: period,
    statistics: ['Sum'],
    unit: 'Count'
  )

  if response.key?(:datapoints) && response.datapoints.count.positive?
    puts "The event rule '#{rule_name}' was triggered:"
    response.datapoints.each do |datapoint|
      puts "  #{datapoint.sum} time(s) at #{datapoint.timestamp}"
    end
  else
    puts "The event rule '#{rule_name}' was not triggered during the " \
      'specified time period.'
  end
rescue StandardError => e
  puts "Error getting information about event rule activity: #{e.message}"
end
```

Display log information for all of the log streams in a CloudWatch Logs log group.

```
# Displays log information for all of the log streams in a log group in
# Amazon CloudWatch Logs.
```

```
#
# Prerequisites:
#
# - A log group in Amazon CloudWatch Logs.
#
# @param cloudwatchlogs_client [Amazon::CloudWatchLogs::Client] An initialized
#   Amazon CloudWatch Logs client.
# @param log_group_name [String] The name of the log group.
# @example
#   display_log_data(
#     Amazon::CloudWatchLogs::Client.new(region: 'us-east-1'),
#     'aws-doc-sdk-examples-cloudwatch-log'
#   )
def display_log_data(cloudwatchlogs_client, log_group_name)
  puts 'Attempting to display log stream data for the log group ' \
    "named '#{log_group_name}'..."
  describe_log_streams_response = cloudwatchlogs_client.describe_log_streams(
    log_group_name: log_group_name,
    order_by: 'LastEventTime',
    descending: true
  )
  if describe_log_streams_response.key?(:log_streams) &&
     describe_log_streams_response.log_streams.count.positive?
    describe_log_streams_response.log_streams.each do |log_stream|
      get_log_events_response = cloudwatchlogs_client.get_log_events(
        log_group_name: log_group_name,
        log_stream_name: log_stream.log_stream_name
      )
      puts "\nLog messages for '#{log_stream.log_stream_name}':"
      puts '-' * (log_stream.log_stream_name.length + 20)
      if get_log_events_response.key?(:events) &&
         get_log_events_response.events.count.positive?
        get_log_events_response.events.each do |event|
          puts event.message
        end
      else
        puts 'No log messages for this log stream.'
      end
    end
  end
rescue StandardError => e
  puts 'Error getting information about the log streams or their messages: ' \
    "#{e.message}"
end
```

Display a reminder to the caller to manually clean up any associated AWS resources that they no longer need.

```
# Displays a reminder to the caller to manually clean up any associated
# AWS resources that they no longer need.
#
# @param topic_name [String] The name of the Amazon SNS topic.
# @param role_name [String] The name of the IAM role.
# @param rule_name [String] The name of the Amazon EventBridge rule.
# @param log_group_name [String] The name of the Amazon CloudWatch Logs log
 group.
# @param instance_id [String] The ID of the Amazon EC2 instance.
# @example
#   manual_cleanup_notice(
#     'aws-doc-sdk-examples-topic',
#     'aws-doc-sdk-examples-cloudwatch-events-rule-role',
#     'aws-doc-sdk-examples-ec2-state-change',
#     'aws-doc-sdk-examples-cloudwatch-log',
#     'i-033c48ef067af3dEX'
#   )
def manual_cleanup_notice(
  topic_name, role_name, rule_name, log_group_name, instance_id
)
  puts '-' * 10
  puts 'Some of the following AWS resources might still exist in your account.'
  puts 'If you no longer want to use this code example, then to clean up'
  puts 'your AWS account and avoid unexpected costs, you might want to'
  puts 'manually delete any of the following resources if they exist:'
  puts "- The Amazon SNS topic named '#{topic_name}'."
  puts "- The IAM role named '#{role_name}'."
  puts "- The Amazon EventBridge rule named '#{rule_name}'."
  puts "- The Amazon CloudWatch Logs log group named '#{log_group_name}'."
  puts "- The Amazon EC2 instance with the ID '#{instance_id}'."
end
```

- For API details, see the following topics in *AWS SDK for Ruby API Reference*.
  - PutEvents
  - PutRule

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Send S3 event notifications to Amazon EventBridge using an AWS SDK

The following code example shows how to enable a bucket to send S3 event notifications to EventBridge and route notifications to an Amazon SNS topic and Amazon SQS queue.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /** This method configures a bucket to send events to AWS EventBridge and
creates a rule
    * to route the S3 object created events to a topic and a queue.
    *
    * @param bucketName Name of existing bucket
    * @param topicArn ARN of existing topic to receive S3 event notifications
    * @param queueArn ARN of existing queue to receive S3 event notifications
    *
    *  An AWS CloudFormation stack sets up the bucket, queue, topic before the
method runs.
    */
   public static String setBucketNotificationToEventBridge(String bucketName,
String topicArn, String queueArn) {
       try {
           // Enable bucket to emit S3 Event notifications to EventBridge.
           s3Client.putBucketNotificationConfiguration(b -> b
                   .bucket(bucketName)
                   .notificationConfiguration(b1 -> b1
                           .eventBridgeConfiguration(
                                   SdkBuilder::build)
                   ).build()).join();

           // Create an EventBridge rule to route Object Created notifications.
```

```
            PutRuleRequest putRuleRequest = PutRuleRequest.builder()
                    .name(RULE_NAME)
                    .eventPattern("""
                            {
                              "source": ["aws.s3"],
                              "detail-type": ["Object Created"],
                              "detail": {
                                "bucket": {
                                  "name": ["%s"]
                                }
                              }
                            }
                            """.formatted(bucketName))
                    .build();

            // Add the rule to the default event bus.
            PutRuleResponse putRuleResponse =
eventBridgeClient.putRule(putRuleRequest)
                    .whenComplete((r, t) -> {
                        if (t != null) {
                            logger.error("Error creating event bus rule: " +
t.getMessage(), t);
                            throw new RuntimeException(t.getCause().getMessage(),
t);
                        }
                        logger.info("Event bus rule creation request sent
successfully. ARN is: {}", r.ruleArn());
                    }).join();

            // Add the existing SNS topic and SQS queue as targets to the rule.
            eventBridgeClient.putTargets(b -> b
                    .eventBusName("default")
                    .rule(RULE_NAME)
                    .targets(List.of (
                            Target.builder()
                                    .arn(queueArn)
                                    .id("Queue")
                                    .build(),
                            Target.builder()
                                    .arn(topicArn)
                                    .id("Topic")
                                    .build())
                            )
                    ).join();
```

```
            return putRuleResponse.ruleArn();
        } catch (S3Exception e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
        return null;
    }
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - PutBucketNotificationConfiguration
  - PutRule
  - PutTargets

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use scheduled events to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

SDK for Java 2.x

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to register an AWS Lambda function as the target of a scheduled Amazon EventBridge event. The Lambda handler writes a friendly message and the full event data to Amazon CloudWatch Logs for later retrieval.

- Deploys a Lambda function.

- Creates an EventBridge scheduled event and makes the Lambda function the target.
- Grants permission to let EventBridge invoke the Lambda function.
- Prints the latest data from CloudWatch Logs to show the result of the scheduled invocations.
- Cleans up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using EventBridge with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Amazon EventBridge security

Amazon EventBridge uses AWS Identity and Access Management to control access to other AWS services and resources. For an overview of how IAM works, see Overview of Access Management in the *IAM User Guide*. For an overview of security credentials, see AWS Security Credentials in the *Amazon Web Services General Reference*.

Topics

- Data protection in Amazon EventBridge
- Tag-based policies in Amazon EventBridge
- Amazon EventBridge and AWS Identity and Access Management
- Logging Amazon EventBridge API calls using AWS CloudTrail
- Compliance validation in Amazon EventBridge
- Amazon EventBridge resilience
- Infrastructure security in Amazon EventBridge
- Configuration and vulnerability analysis in Amazon EventBridge

Data protection in Amazon EventBridge

The AWS shared responsibility model applies to data protection in Amazon EventBridge. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with EventBridge or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption in EventBridge

EventBridge provides both *encryption at rest* and *encryption in transit* to protect your data:

- Encryption at rest

  EventBridge integrates with AWS Key Management Service (KMS) to encrypt stored data. By default, EventBridge uses an AWS owned key to encrypt data. You can also specify for EventBridge to use a customer managed key for specific features instead.

- Encryption in transit

  EventBridge encrypts data that passes between EventBridge and other services by using Transport layer Security (TLS).

  For event buses, this includes during an event being sent to EventBridge, as well as when EventBridge sends an event to a rule target.

Encryption at rest in Amazon EventBridge

EventBridge provides transparent server-side encryption by integrating with AWS Key Management Service (KMS). Encryption of data at rest by default helps reduce the operational overhead and complexity involved in protecting sensitive data. At the same time, it enables you to build secure applications that meet strict encryption compliance and regulatory requirements.

By default, EventBridge uses an AWS owned key to encrypt data. You can specify for EventBridge to use customer managed keys for specific resources instead.

The following table lists the artifacts that EventBridge encrypts at rest, by resource:

Resource	Details	AWS owned key	Customer managed key
API destinations	Connection authorization parameters are stored in an AWS Secrets Manager secret.	Supported	Supported
Archives		Supported	Supported
Connections	Connection authorization parameters are stored in an AWS Secrets Manager secret.	Supported	Supported
Event buses	Includes:	Supported	Supported

Resource	Details	AWS owned key	Customer managed key
	• Events from custom and partner sources Event data includes all fields contained in the `event-detail` element of the event. EventBridge does not encrypt event metadata. • Rule event patterns. • Rule target information, including input, input transformers, and configuration parameters. • For event bus logging, the `detail` and `error` sections of log records.		
Pipes	Includes: • Event patterns • Input transformers • Execution data in logs Events flowing through a pipe are never stored at rest.	Supported	Supported

Important

We strongly recommend that you never put confidential or sensitive information in the following artifacts, as they are not encrypted at rest:

- Event bus names
- Rule names

- Shared resources, such at tags

KMS key options for data encryption in Amazon EventBridge

EventBridge uses an AWS owned key to encrypt data stored in resources. For each resource, you can choose the type of KMS key EventBridge uses to encrypt the data:

- **AWS owned key**

  By default, EventBridge encrypts data using 256-bit Advanced Encryption Standard (AES-256) under an AWS owned key, which helps secure your data from unauthorized access.

  You can't view, manage, or use AWS owned keys, or audit their use. However, you don't have to take any action or change any programs to protect the keys that encrypt your data.

  In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice. AWS owned keys are completely free of charge (no monthly fees or usage fees), and they do not count against the AWS KMS quotas for your account. You don't need to create or maintain the key or its key policy.

  For more information, see AWS owned keys in the *AWS Key Management Service Developer Guide*.
- **Customer managed key**

  EventBridge supports the use of a symmetric customer managed key that you create, own, and manage. Because you have full control of this type of KMS key, you can perform such tasks as:
  - Establishing and maintaining key policies
  - Establishing and maintaining IAM policies and grants
  - Enabling and disabling key policies
  - Rotating key cryptographic material
  - Adding tags
  - Creating key aliases
  - Scheduling keys for deletion

  For more information, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

  EventBridge supports Multi-Region keys and cross account access of keys.

Customer managed keys incur a monthly fee. For details, see AWS Key Management Service Pricing, and Quotas in the *AWS Key Management Service Developer Guide*.

> **Note**
>
> EventBridge does not support schema discovery on event buses encrypted using customer managed keys.

Authorizing EventBridge to use a customer managed key

If you use a customer managed key in your account to protect your EventBridge resources, the policies on that KMS key must give EventBridge permission to use it on your behalf. You provide these permissions in a key policy.

EventBridge does not need additional authorization to use the default AWS owned key to protect the EventBridge resources in your AWS account.

EventBridge requires the following permissions to use customer managed keys:

- `kms:DescribeKey`

  EventBridge requires this permission to retrieve the KMS key ARN for the Key Id provided, and to verify that the key is symmetric.
- `kms:GenerateDataKey`

  EventBridge requires this permission to generate a data key as the encryption key for the data.
- `kms:Decrypt`

  EventBridge requires this permission to decrypt the data key that is encrypted and stored with the encrypted data.

  EventBridge uses this for event pattern matching; users never have access to the data.

Security when using customer managed keys for EventBridge encryption

As a security best practice, add an `aws:SourceArn`, `aws:sourceAccount`, or `kms:EncryptionContext:aws:events:event-bus:arn` condition key to the AWS KMS key

policy. The IAM global condition key helps ensure that EventBridge uses the KMS key only for the specified bus or account.

The following example demonstrates how to follow this best practice in your IAM policy for an event bus:

```
{
      "Sid": "Allow the use of key",
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
      ],
      "Resource": "*",
      "Condition" : {
        "StringEquals": {
          "aws:SourceAccount": "arn:aws:events:region:account-id",
          "aws:SourceArn": "arn:aws:events:region:account-id:event-bus/event-bus-name",
          "kms:EncryptionContext:aws:events:event-bus:arn":
 "arn:aws:events:region:account-id:event-bus/event-bus-arn"
        }
      }
```

Maintaining AWS KMS encryption key access in EventBridge

To ensure EventBridge always retains access to the necessary customer managed key:

- Do not delete a customer managed key until you are sure all resources encrypted with it have been processed.

  When you perform any of the following operations, retain the previous key material to ensure EventBridge can continue to use it for previously-encrypted resources:

  - Automatic key rotation
  - Manual key rotation
  - Updating a key alias

In general, If you are considering deleting a AWS KMS key, disable it first and set a CloudWatch alarm or similar mechanism to be certain that you'll never need to use the key to decrypt encrypted data.

- Do not delete the key policy that provides EventBridge the permissions to use the key.

Other considerations include:

- Specify customer managed keys for rule targets, as appropriate.

  When EventBridge sends an event to a rule target, the event is sent using Transport layer Security (TLS). However, what encryption is applied to the event as it is stored on the target depends on the encryption you have configured on the target itself.

Encrypting EventBridge event buses with AWS KMS keys

You can specify that EventBridge use an AWS KMS to encrypt your data stored on an event bus, rather than use an AWS owned key as is the default. You can specify a customer managed key when you create or update an event bus. You can also update the default event bus to use a customer managed key for encryption as well. For more information, see KMS key options.

When you specify a customer managed key for an event bus, EventBridge uses that key to encrypt the following:

- Custom and partner events stored on the event bus.

  Events from AWS service are encrypted using an AWS owned key.

  EventBridge does not encrypt event metadata. For more information on event metadata, see AWS service event metadata in the *Events Reference*.
- For each rule on the bus:
  - The rule event pattern.
  - Target information, including target input, input transformers, and configuration parameters.
- If event bus logging is enabled, the `detail` and `error` sections of log records.

If you specify a customer managed key for an event bus, you have the option of specifying a dead-letter queue (DLQ) for the event bus. EventBridge then delivers any custom or partner events

that generate encryption or decryption errors to that DLQ. For more information, see DLQs for encrypted events.

> **Note**
>
> We strongly recommend specifying a DLQ for event buses, to ensure events are preserved if encryption or decryption errors occur.

You can also specify using customer managed keys for encrypting event bus archives. For more information, see Encrypting archives.

> **Note**
>
> Schema discovery is not supported for event buses encrypted using a customer managed key. To enable schema discovery on an event bus, choose to use an AWS owned key. For more information, see KMS key options.

Event bus encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

You can also use the encryption context as a condition for authorization in policies and grants.

If you use a customer managed key to protect your EventBridge resources, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

For event buses, EventBridge uses the same encryption context in all AWS KMS cryptographic operations. The context includes a single key–value pair, which contains the event bus ARN.

```
"encryptionContext": {
    "kms:EncryptionContext:aws:events:event-bus:arn": "event-bus-arn"
}
```

AWS KMS key policy for event bus

The following example key policy provides the required permissions for an event bus:

- `kms:DescribeKey`
- `kms:GenerateDataKey`
- `kms:Decrypt`

As a security best practice, we recommend you include condition keys in the key policy to helps ensure that EventBridge uses the KMS key only for the specified resource or account. For more information, see Security considerations.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowEventBridgeToValidateKeyPermission",
      "Effect": "Allow",
      "Principal": {
      "Service": "events.amazonaws.com"
       },
      "Action": [
        "kms:DescribeKey"
        ],
      "Resource": "*"
    },
  {
    "Sid": "AllowEventBridgeToEncryptEvents",
    "Effect": "Allow",
    "Principal": {
      "Service": "events.amazonaws.com"
    },
    "Action": [
      "kms:GenerateDataKey",
      "kms:Decrypt"
    ],
    "Resource": "*",
    "Condition": {
      "StringEquals": {
```

```
          "kms:EncryptionContext:aws:events:event-bus:arn": "arn:aws-
eusc:events:eusc-de-east-1:123456789012:event-bus/event-bus-arn",
          "aws:SourceArn": "arn:aws-eusc:events:eusc-de-
east-1:123456789012:event-bus/event-bus-name"
        }
      }
    }
  ] }
```

AWS KMS key permissions for event bus actions

To create or update an event bus encrypted using a customer managed key, you must have the following permissions to the specified customer managed key:

- `kms:GenerateDataKeyWithoutPlaintext`
- `kms:Decrypt`
- `kms:Encrypt`
- `kms:ReEncryptFrom`
- `kms:ReEncryptTo`
- `kms:DescribeKey`

In addition, to perform certain event bus actions on an event bus encrypted using a customer managed key, you must have `kms:Decrypt` permission for the specified customer managed key. These actions include:

- DescribeRule
- DisableRule
- EnableRule
- ListRules
- ListTargetsByRule
- PutRule
- ListRuleNamesByTarget
- PutTargets

Configuring encryption on EventBridge event buses

You can specify the KMS key for EventBridge to use when you create or update an event bus. You can also update the default event bus to use a customer managed key as well.

Specifying the AWS KMS key used for encryption when creating an event bus

Choosing the AWS KMS key used for encryption is part of creating an event bus. The default is to use the AWS owned key provided by EventBridge.

To specify a customer managed key for encryption when creating an event bus (console)

- Follow these instructions:

  Creating an event bus.

To specify a customer managed key for encryption when creating an event bus (CLI)

- When calling `create-event-bus`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encryption on the event bus.

  Optionally, use `dead-letter-config` to specify a dead-letter queue (DLQ).

Updating the AWS KMS key used for encryption on an event bus

You can update the AWS KMS key being used for encryption at rest on an existing event bus. This includes:

- Changing from the default AWS owned key to a customer managed key.
- Changing from a customer managed key to the default AWS owned key.
- Changing from one customer managed key to another.

When you update an event bus to use a different AWS KMS key, EventBridge uses the new key to re-encrypt the data keys that protect data stored on the event bus. EventBridge does not re-encrypt the stored data itself.

To update the KMS key used for encryption on an event bus (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Choose the event bus you want to update.
4. On the events bus details page, choose the **Encryption** tab.
5. Choose the KMS key for EventBridge to use when encrypting the event data stored on the event bus:
   - Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

     This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

     This is the default.
   - Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

     Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

     a. Specify an existing customer managed key, or choose **Create a new KMS key**.

        EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

     b. Choose the Amazon SQS queue to use as the dead-letter queue (DLQ) for this event bus, if any.

        EventBridge sends events that aren't successfully encrypted to the DLQ, if configured, so you can process them later.

To update the KMS key used for encryption on an event bus (CLI)

- When calling `update-event-bus`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encryption on the event bus.

Optionally, use `dead-letter-config` to specify a dead-letter queue (DLQ).

To update the KMS key used for encryption on the default event bus, using CloudFormation

Because EventBridge provisions the default event bus into your account automatically, you cannot create it using a CloudFormation template, as you normally would for any resource you wanted to include in a CloudFormation stack. To include the default event bus in a CloudFormation stack, you must first *import* it into a stack. Once you have imported the default event bus into a stack, you can then update the event bus properties as desired.

- Follow these instructions:

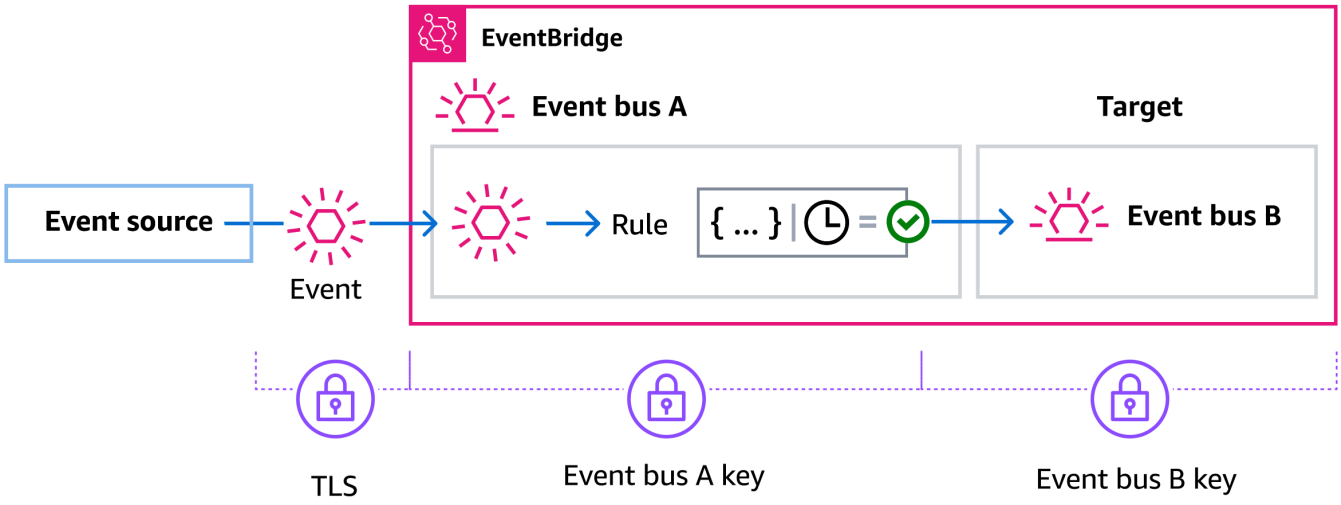
  Updating a default bus using CloudFormation.

Encryption in EventBridge when an event bus is the rule target

When a custom or partner event is sent to an event bus, EventBridge encrypts that event according to the encryption at rest KMS key configuration for that event bus - either the default AWS owned key or a customer managed key, if one has been specified. If an event matches a rule, EventBridge encrypts the event with the KMS key configuration for that event bus until the event is sent to the rule target, *unless the rule target is another event bus*.

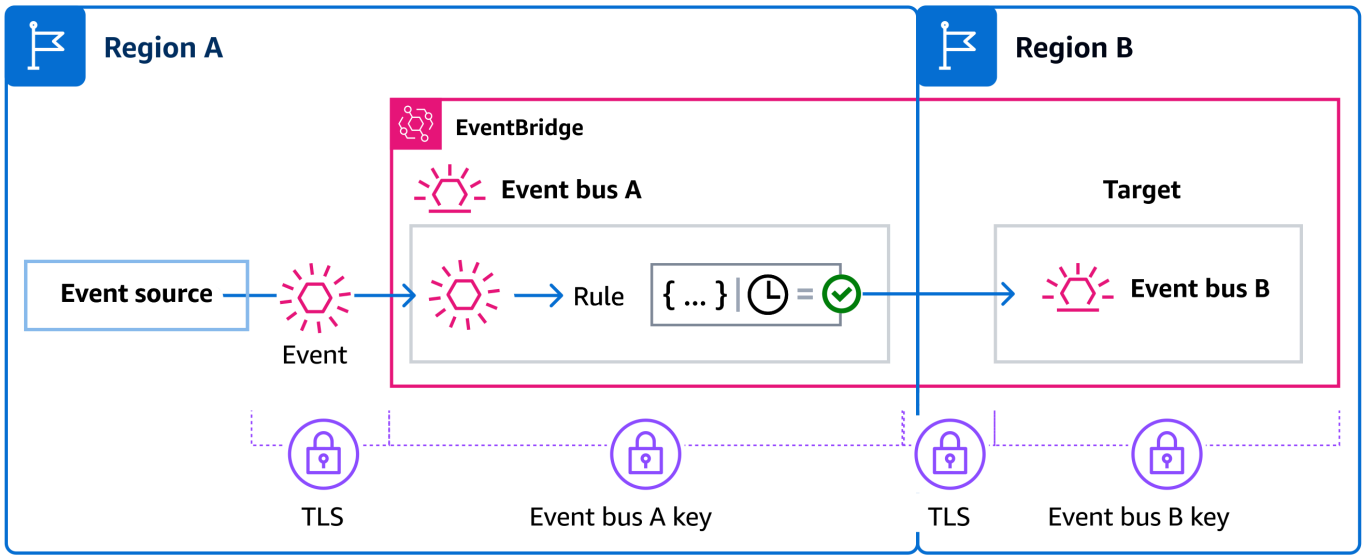
- If the target of a rule is another event bus in the same AWS Region:

  If the target event bus has a specified customer managed key, EventBridge encrypts the event with the customer managed key of the target event bus for delivery instead.



- If the target of a rule is another event bus in a different AWS Region:

  EventBridge encrypts the event at rest according to the KMS key configuration on the first event bus. EventBridge uses TLS to send the event to the second event bus in the different Region, where it is then encrypted according to the KMS key configuration specified for the target event bus.

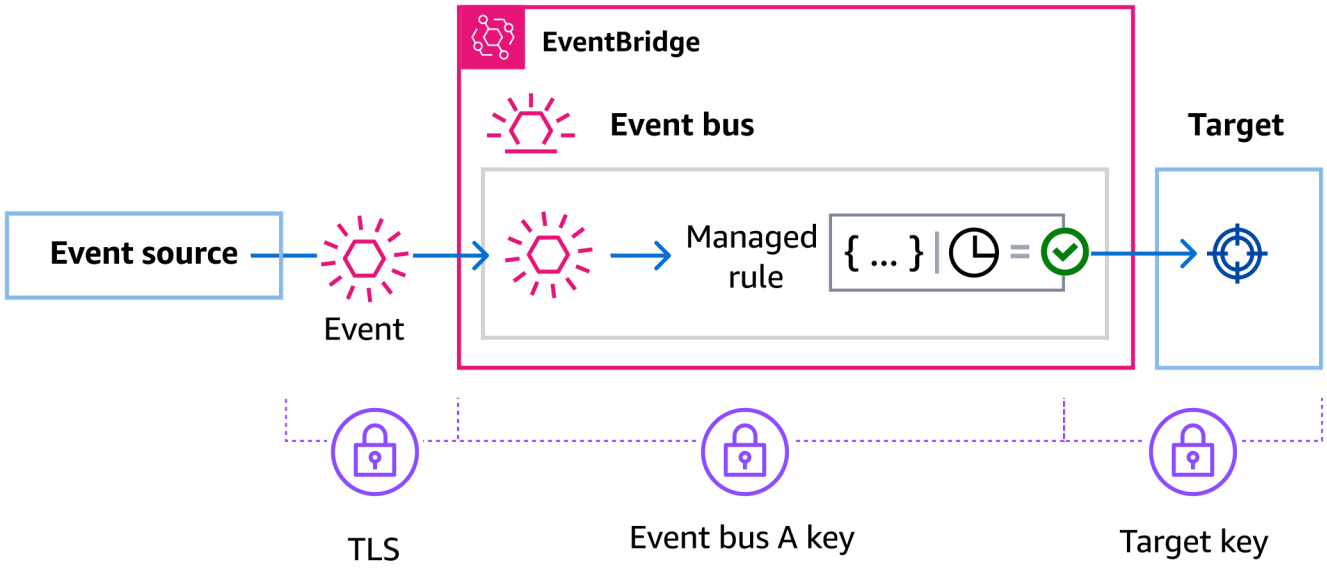


Event encryption for managed rules in EventBridge

AWS services can create and manage event bus rules in your AWS account that are needed for certain functions in those services. As part of a managed rule, the AWS service can specify that

EventBridge use the customer managed key specified for the rule target. This gives you the flexibility to specify which customer managed key to use based on the rule target.

In these cases, once a custom or partner event matches against the managed rule, EventBridge uses the target customer managed key specified by the managed rule to encrypt the event until it is sent to the rule target. This is regardless of whether the event bus has been configured to use its own customer managed key for encryption. This is the case even if the target of the managed rule is another event bus, and that event bus has its own customer managed key specified for encryption. EventBridge continues to use the target customer managed key specified in the managed rule until the event is sent to a target that is not an event bus.



For cases where the rule target is an event bus in another Region, you must provide a multi-Region key. The event bus in the first Region encrypts the event using the customer managed key specified in the managed rule. It then sends the event to the target event bus in the second Region. That event bus must be able to continue to use the customer managed key until it sends the event to its target.

Encrypting event bus logs with AWS KMS in EventBridge

When sending logs, EventBridge encrypts the `detail` and `error` sections of each log record with the KMS key specified for the event bus. If you have specified a customer managed key for the event bus, EventBridge uses that key for encryption in transit. Once delivered, the record is decrypted and then re-encrypted with the KMS key specified for the log destination.

Event bus logs encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

You can also use the encryption context as a condition for authorization in policies and grants.

If you use a customer managed key to protect your EventBridge resources, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

For event bus logs, EventBridge uses the same encryption context in all AWS KMS cryptographic operations.

```
"encryptionContext": {
    "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
}
```

AWS KMS key policy permissions for event bus logging

For event buses using a customer managed key, you must add the following permissions to the key policy.

- Allow EventBridge to encrypt logs using the customer managed key.

```
{
  "Sid": "Enable log service encryption",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": [
    "kms:GenerateDataKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
    }
```

```
  }
}
```

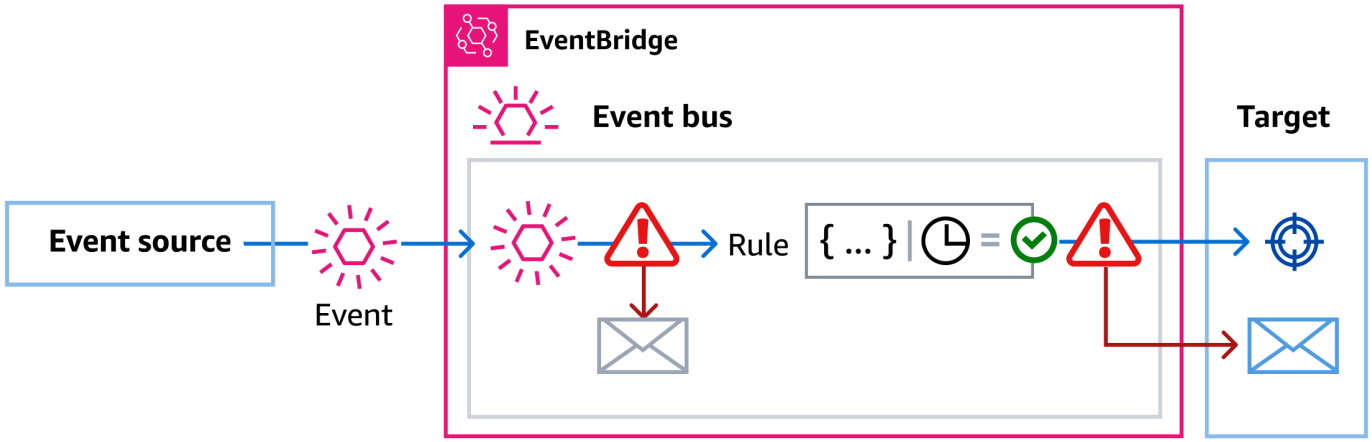
- Allow the logging service to decrypt logs sent by EventBridge.

```
{
  "Sid": "Enable log delivery decryption",
  "Effect": "Allow",
  "Principal": {
    "Service": "delivery.logs.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
    }
  }
}
```

Using dead-letter queues to capture encrypted event errors in EventBridge

If you configure customer managed key encryption on an event bus, we recommend that you specify a dead-letter queue (DLQ) for that event bus. EventBridge sends custom and partner events to this DLQ if it encounters a non-retriable error while processing the event on the event bus. A non-retriable error is one where user action is required to resolve the underlying issue, such as the specified customer managed key being disabled or missing.

- If a non-retriable encryption or decryption error occurs while EventBridge is processing the event on the event bus, the event is sent to the DLQ for the *event bus*, if one is specified.
- If a non-retriable encryption or decryption error occurs while EventBridge is attempting to send the event to a target, including input transformations and target-specific settings, the event is sent to the DLQ for the *target*, if one is specified.



For more information, including considerations when using DLQs, and instructions on setting permissions, see Using dead-letter queues.

Decrypting events in EventBridge dead-letter queues

Once you've resolved the underlying issue that is causing a non-retriable error, you can process the events sent to the event bus or target DLQs. For encrypted events, you must first decrypt the event in order to process it.

The following example demonstrates how to decrypt an event that EventBridge has delivered to an event bus or target DLQ.

```
// You will receive an encrypted event in the following json format.
        // ```
        //   {
        //     "version": "0",
        //     "id": "053afa53-cdd7-285b-e754-b0dfd0ac0bfb",  // New event id not the
 same as the original one
        //     "account": "123456789012",
        //     "time": "2020-02-10T10:22:00Z",
        //     "resources": [ ],
        //     "region": "us-east-1",
        //     "source": "aws.events",
        //     "detail-type": "Encrypted Events",
        //     "detail": {
        //       "event-bus-arn": "arn:aws:events:region:account:event-bus/bus-name",
        //       "rule-arn": "arn:aws:events:region:account:event-bus/bus-name/rule-
name",
        //       "kms-key-arn": "arn:aws:kms:region:account:key/key-arn",
```

```
        //       "encrypted-payload": "AgR4qiru/XNwTUyCgRHqP7rbbHn/
xpmVeVeRIAd12TDYYVwAawABABRhd3M6ZXZlbnRzOmV2ZW50LWJ1cwB
        //
 RYXJuOmF3czpldmVudHM6dXMtZWFzdC0xOjE0NjY4NjkwNDY3MzpldmVudC1idXMvY21rbXMtZ2EtY3Jvc3
        //
 MtYWNjb3VudC1zb3VyY2UtYnVzAAEAB2F3cy1rbXMAS2Fybjphd3M6a21zOnVzLWVhc3QtMToxNDY2ODY5"
        //     }
        //   }
        // ```

        // Construct an AwsCrypto object with the encryption algorithm
`ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY` which
        // is used by EventBridge for encryption operation. This object is an entry
point for decryption operation.
        // It can later use decryptData(MasterKeyProvider, byte[]) method to decrypt
data.
        final AwsCrypto crypto = AwsCrypto.builder()

.withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
                .build();

        // Construct AWS KMS master key provider with AWS KMS Client Supplier and AWS
KMS Key ARN. The KMS Client Supplier can
        // implement a RegionalClientSupplier interface. The AWS KMS Key ARN can be
fetched from kms-key-arn property in
        // encrypted event json detail.
        final KmsMasterKeyProvider kmsMasterKeyProvider =
KmsMasterKeyProvider.builder()
                .customRegionalClientSupplier(...)
                .buildStrict(KMS_KEY_ARN);

        // The string of encrypted-payload is base64 encoded. Decode it into byte
array, so it can be furthur
        // decrypted. The encrypted payload can be fetched from encrypted-payload field
in encrypted event json detail.
        byte[] encryptedByteArray = Base64.getDecoder().decode(ENCRYPTED_PAYLOAD);

        // The decryption operation. It retrieves the encryption context and encrypted
data key from the cipher
        // text headers, which is parsed from byte array encrypted data. Then it
decrypts the data key, and
        // uses it to finally decrypt event payload. This encryption/decryption
strategy is called envelope
```

```
        // encryption, https://docs.aws.amazon.com/kms/latest/developerguide/
concepts.html#enveloping
        final CryptoResult<byte[], KmsMasterKey> decryptResult =
 crypto.decryptData(kmsMasterKeyProvider, encryptedByteArray);

        final byte[] decryptedByteArray = decryptResult.getResult();

        // Decode the event json plaintext from byte array into string with UTF_8
 standard.
        String eventJson = new String(decryptedByteArray, StandardCharsets.UTF_8);
```

Encrypting EventBridge Pipes data with AWS KMS keys

You can specify that EventBridge use a customer managed key to encrypt pipe data stored at rest, rather than use an AWS owned key as is the default. You can specify a customer managed key when you create or update a pipe. For more information about key types, see KMS key options.

The pipe data EventBridge encrypts at rest includes:

- Event patterns
- Input transformers

Events flowing through a pipe are never stored at rest.

EventBridge Pipes encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

You can also use the encryption context as a condition for authorization in policies and grants.

If you use a customer managed key to protect your EventBridge resources, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

For EventBridge Pipes, EventBridge uses the same encryption context in all AWS KMS cryptographic operations. The context includes a single key–value pair, which contains the pipe ARN.

```
"encryptionContext": {
    "kms:EncryptionContext:aws:pipes:arn": "pipe-arn"
}
```

For vended logs, EventBridge uses the following encryption context.

```
"encryptionContext": {
    "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
}
```

AWS KMS key policy for EventBridge Pipes

The following example key policy provides the required permissions for a pipe:

- `kms:DescribeKey`
- `kms:GenerateDataKey`
- `kms:Decrypt`

As a security best practice, we recommend you include condition keys in the key policy to helps ensure that EventBridge uses the AWS KMS key only for the specified resource or account. For more information, see Security considerations.

JSON

```
{
  "Id": "CMKKeyPolicy",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
      "AWS": "arn:aws-eusc:iam::123456789012:role/pipe-execution-role"
      },
      "Action": [
        "kms:DescribeKey"
      ],
      "Resource": "*"
    },
    {
```

```
      "Effect": "Allow",
      "Principal": {
      "AWS": "arn:aws-eusc:iam::123456789012:role/pipe-execution-rolee"
      },
      "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
      ],
      "Resource": "*",
      "Condition": {
        "ArnLike": {
        "kms:EncryptionContext:aws:pipe:arn": "arn:aws-eusc:pipes:eusc-de-
east-1:123456789012:pipe/pipe-name"
        },
        "ForAnyValue:StringEquals": {
          "kms:EncryptionContextKeys": [
            "aws:pipe:arn"
          ]
        }
      }
    }
  ]
}
```

Permissions for pipe logs that include execution data

If you have configured pipes logging to include execution data, the key policy must include the following permissions for the logging service:

- `kms:Decrypt`
- `kms:GenerateDataKey`

For more information, see the section called “Including execution data in logs”.

The following example key policy provides the required permissions for pipes logging:

```
{
  "Sid": "Enable log service encryption",
  "Effect": "Allow",
  "Principal": {
    "Service": "delivery.logs.amazonaws.com"
```

```
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
    }
  }
}
```

In addition, the pipe execution role requires the `kms:GenerateDataKey` permisson.

```
{
  "Sid": "Enable log service encryption",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::account:role/pipe-execution-role"
  },
  "Action": [
    "kms:GenerateDataKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
    }
  }
}
```

The pipe execution role should also include:

```
"Action": [
    "kms:GenerateDataKey"
  ],
  "Resource": "key-arn",
  "Condition": {
    "StringLike": {
      "kms:EncryptionContext:SourceArn": "arn:partition:logs:region:account:*"
    }
```

```
}
```

Configuring encryption in EventBridge Pipes

You can specify the KMS key for EventBridge to use when you create or update a pipe.

Specifying the AWS KMS key used for encryption when creating a pipe

Choosing the AWS KMS key used for encryption is an option creating a pipe. The default is to use the AWS owned key provided by EventBridge.

To specify a customer managed key for encryption when creating a pipe (console)

- Follow these instructions:

  Creating a pipe.

To specify a customer managed key for encryption when creating a pipe (CLI)

- When calling `create-pipe`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encryption on the event bus.

Updating the AWS KMS key used for encryption on EventBridge Pipes

You can update the AWS KMS key being used for encryption at rest on an existing pipe. This includes:

- Changing from the default AWS owned key to a customer managed key.
- Changing from a customer managed key to the default AWS owned key.
- Changing from one customer managed key to another.

When you update a pipe to use a different AWS KMS key, EventBridge uses the new key to re-encrypt the data keys that protect data stored on the pipe. EventBridge does not re-encrypt the stored data itself.

To update the KMS key used for encryption on a pipe (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.

2. In the navigation pane, choose **Pipes**.
3. Choose the pipe you want to update.
4. On the pipe bus details page, choose the **Encryption** tab.
5. Choose the KMS key for EventBridge to use when encrypting the data stored on pipe:
   - Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

     This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

     This is the default.
   - Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

     Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

     - Specify an existing customer managed key, or choose **Create a new KMS key**.

       EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

To update the KMS key used for encryption on a pipe (CLI)

- When calling `update-pipe`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encrypting pipe data.

Encrypting EventBridge archives with AWS KMS keys

You can specify that EventBridge use a customer managed key to encrypt events stored in an archive, rather than use an AWS owned key as is the default. You can specify a customer managed key when you create or update an archive. For more information about key types, see KMS key options.

This includes:

- Events stored in the archive

- The event pattern, if any, specified to filter the events sent to the archive

This does not include archive metadata, such as the size of the archive or number of events it contains.

If you specify a customer managed key for an archive, EventBridge encrypts events prior to sending it to the archive, ensuring encryption in transit and at rest.

Archive encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

You can also use the encryption context as a condition for authorization in policies and grants.

If you use a customer managed key to protect your EventBridge resources, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

For event archives, EventBridge uses the same encryption context in all AWS KMS cryptographic operations. The context includes a single key–value pair, which contains the event bus ARN.

```
"encryptionContext": {
    "kms:EncryptionContext:aws:events:event-bus:arn": "event-bus-arn"
}
```

AWS KMS key policy for archives

The following example key policy provides the required permissions for an event archive:

- `kms:DescribeKey`
- `kms:GenerateDataKey`
- `kms:Decrypt`
- `kms:ReEncrypt`

As a security best practice, we recommend you include condition keys in the key policy to helps ensure that EventBridge uses the KMS key only for the specified resource or account. For more information, see Security considerations.

JSON

```
{
  "Id": "CMKKeyPolicy",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": [
        "kms:DescribeKey"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt",
        "kms:ReEncrypt*"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
         "kms:EncryptionContext:aws:events:event-bus:arn": "arn:aws-
eusc:events:eusc-de-east-1:123456789012:event-bus/event-bus-arn"
        }
      }
    }
  ]
}
```

Configuring encryption on archives

You can specify the KMS key for EventBridge to use when you create or update an archive.

Specifying encryption when creating an archive

Choosing the AWS KMS key used for encryption is an option creating an archive. The default is to use the AWS owned key provided by EventBridge.

To specify a customer managed key for encryption when creating an archive (console)

- Follow these instructions:

  Creating archives.

To specify a customer managed key for encryption when creating an archive (CLI)

- When calling `create-archive`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encrypting events stored in the archive.

Updating encryption on archives

You can update the AWS KMS key being used for encryption at rest on an existing archive. This includes:

- Changing from the default AWS owned key to a customer managed key.
- Changing from a customer managed key to the default AWS owned key.
- Changing from one customer managed key to another.

To update the KMS key used for encrypting events in an archive (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Navigate to the archive directly, or from the source event bus:
   - In the navigation pane, choose **Event buses**.

     On the events bus details page, choose the **Archives** tab.
   - In the navigation pane, choose **Archives**.

3. Choose the archive you want to update.
4. On the archive details page, choose the **Encryption** tab.
5. Choose the KMS key for EventBridge to use when encrypting the events stored in the archive.

> **Important**
>
> If you have specify that EventBridge use a customer managed key for encrypting the source event bus, we strongly recommend you also specify a customer managed key for any archives for the event bus as well.

- Choose **Use AWS owned key** for EventBridge to encrypt the data using an AWS owned key.

  This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

  This is the default.
- Choose **Use customer managed key** for EventBridge to encrypt the data using the customer managed key that you specify or create.

  Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

  - Specify an existing customer managed key, or choose **Create a new KMS key**.

    EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

To update the KMS key used for encrypting events stored in an archive (CLI)

- When calling `update-archive`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encrypting events stored in the archive.

Encrypting EventBridge connection authorization with AWS KMS keys

When you create or update a connection, you can specify authorization parameters for that connection. EventBridge then securely stores those parameters in a secret in AWS Secrets Manager. By default, EventBridge uses an AWS owned key to encrypt and decrypt this secret. You can specify that EventBridge use a customer managed key instead.

AWS KMS key policy for connections

The AWS KMS key policy must grant EventBridge the following permissions on your behalf:

- `kms:DescribeKey`
- `kms:GenerateDataKey`
- `kms:Decrypt`

The following policy example grants all AWS KMS permissions.

JSON

```
{
  "Id": "key-policy-example",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Enable IAM User Permissions",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::012345678901:root"
      },
      "Action": "kms:*",
      "Resource": "*"
    }
  ]
}
```

For EventBridge to use a customer managed key, you must add a resource tag to the key with a key of `EventBridgeApiDestinations` and a value of `true`. For more information on resource tags, see Add tags to a KMS key in the *AWS Key Management Service Developer Guide*.

As a security best practice, we recommend you include condition keys in the key policy to helps ensure that EventBridge uses the KMS key only for the specified resource or account. For more information, see Security considerations.

```
"Condition": {
  "StringLike": {
    "kms:ViaService": "secretsmanager.*.amazonaws.com",
    "kms:EncryptionContext:SecretARN": [
      "arn:aws:secretsmanager:*:*:secret:events!connection/*"
    ]
  },
  "StringEquals": {
    "kms:ResourceTag/EventBridgeApiDestinations": "true"
  }
}
```

Connection encryption context

An encryption context is a set of key–value pairs that contain arbitrary nonsecret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

You can also use the encryption context as a condition for authorization in policies and grants.

If you use a customer managed key to protect your EventBridge resources, you can use the encryption context to identify use of the KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail and Amazon CloudWatch Logs.

For connections, EventBridge uses the same encryption context in all AWS KMS cryptographic operations. The context includes a single key–value pair, which contains the secret ARN.

```
"encryptionContext": {
    "kms:EncryptionContext:SecretARN": "secret-arn"
}
```

Using cross-account or cross-Region customer managed keys for connections

You can allow users or roles in a different AWS account to use a KMS key in your account. Cross-account access requires permission in the key policy of the KMS key and in an IAM policy in the external user's account.

To use a customer managed key from another account, the account with the customer managed key must include the following policy:

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::account:role/
AmazonEventBridgeApiDestinationsInternalServiceRolePolicy"
  },
  "Action": [
    "kms:Decrypt",
    "kms:DescribeKey"
  ],
  "Resource": "*"
}
```

For more information, see Allowing users in other accounts to use a KMS key in the *AWS Key Management Service Developer Guide*.

Revoking customer managed key access to connections

Be aware that when you revoke a customer managed key--through disabling, deleting, or rotating the key, or updating the key policy--EventBridge may have cached the key value, and so that key may still retain access to a connection's secret for a short period of time.

To immediately revoke customer managed key access to a connection's secret, de-authorize or delete the connection. For more information, see De-authorizing connections and Deleting connections.

Connection de-authorization due to customer managed key errors

EventBridge de-authorizes a connection if it encounters the following errors when attempting to encrypt or decrypt the connection's secret:

- The customer managed key has been deleted.
- The customer managed key has been disabled.
- The connection does not have the necessary permissions to access the customer managed key.

For more information, see De-authorizing connections.

Configuring encryption on connections

You can specify the KMS key for EventBridge to use when you create or update a connection.

Specifying AWS KMS keys when creating connections

Choosing the AWS KMS key used for encryption is optional when creating a connection. By default, EventBridge uses an AWS owned key.

To specify a customer managed key for encryption when creating a connection (console)

- Follow these instructions:

  Creating connections.

To specify a customer managed key for encryption when creating a connection (CLI)

- When calling `create-connection`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encryption of the connection's secret.

Updating AWS KMS keys for connections

You can update the KMS key being used for encrypting an existing connection. This includes:

- Changing from the default AWS owned key to a customer managed key.
- Changing from a customer managed key to the default AWS owned key.
- Changing from one customer managed key to another.

When you update a connection to use a different KMS key , EventBridge decrypts the connection's secret and then encrypts it using the new key. Make sure the KMS key you specify has the necessary permissions. For more information, see Connection key policy.

To update the KMS key used for encryption on a connection (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Integration**, and then choose **Connections**.
3. Choose the connection you want to update.

4. On the connection details page, under **Encryption**, choose the KMS key for EventBridge to use when encrypting the connection's secret:
   - Choose **Use AWS owned key** for EventBridge to encrypt the secret using an AWS owned key.

     This AWS owned key is a KMS key that EventBridge owns and manages for use in multiple AWS accounts. In general, unless you are required to audit or control the encryption key that protects your resources, an AWS owned key is a good choice.

     This is the default.
   - Choose **Choose a different AWS KMS key (advanced)** for EventBridge to encrypt the secret using the customer managed key that you specify or create.

     Customer managed keys are KMS keys in your AWS account that you create, own, and manage. You have full control over these KMS keys.

     - Specify an existing customer managed key, or choose **Create a new KMS key**.

       Make sure the KMS key you specify has the necessary permissions. For more information, see Connection key policy.

       EventBridge displays the key status and any key aliases that have been associated with the specified customer managed key.

To update the KMS key used for encryption on a connection (CLI)

- When calling `update-connection`, use the `kms-key-identifier` option to specify the customer managed key for EventBridge to use for encrypting the connection secret.

Tag-based policies in Amazon EventBridge

In Amazon EventBridge, you can use policies based on tags to control access to resources.

For example, you could restrict access to resources that include a tag with the key `environment` and the value `production`. The following example policy denies any resource with this tag the ability to create, delete, or modify tags, rules, or event buses for resources that have been tagged `environment/production`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "events:PutRule",
                "events:DescribeRule",
                "events:DeleteRule",
                "events:CreateEventBus",
                "events:DescribeEventBus",
                "events:DeleteEventBus"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/environment": "production"
                }
            }
        }
    ]
}
```

For more information about tagging, see the following.

- Tagging resources in Amazon EventBridge
- Controlling Access Using IAM Tags

Amazon EventBridge and AWS Identity and Access Management

To access Amazon EventBridge, you need credentials that AWS can use to authenticate your requests. Your credentials must have permissions to access AWS resources, such as retrieving event data from other AWS resources. The following sections provide details on how you can use AWS Identity and Access Management (IAM) and EventBridge to help secure your resources by controlling who can access them.

Topics

- Authentication
- Access control
- Managing access permissions to your Amazon EventBridge resources
- Using identity-based policies (IAM policies) for Amazon EventBridge
- Using resource-based policies for Amazon EventBridge
- Cross-service confused deputy prevention in Amazon EventBridge
- Resource-based policies for Amazon EventBridge schemas
- Amazon EventBridge permissions reference
- Using IAM policy conditions in Amazon EventBridge
- Using service-linked roles for EventBridge

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you sign up for AWS, you provide an email address and password that is associated with your account. These are your *root credentials*, and they provide complete access to all of your AWS resources.

> **Important**
>
> For security reasons, we recommend that you use the root credentials only to create an *administrator*, which is an *IAM user* with full permissions to your account. Then you can use this administrator to create other users and roles with limited permissions. For more

information, see IAM Best Practices and Creating an Admin User and Group in the *IAM User Guide*.

- **IAM user** – An IAM user is an identity within your account that has specific permissions, for example, permission to send event data to a target in EventBridge. You can use an IAM sign-in credentials to sign in to secure AWS webpages such as the AWS Management Console, AWS Discussion Forums, or the AWS Support Center.

  In addition to sign-in credentials, you can also generate access keys for each user. You can use these keys when you access AWS services programmatically to cryptographically sign your request, either through one of the SDKs or by using the AWS Command Line Interface (AWS CLI). If you don't use AWS tools, you must sign the request yourself with *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *Amazon Web Services General Reference*.

- **IAM role** – An IAM role is another IAM identity that you can create in your account that has specific permissions. It's similar to an *IAM user*, but it isn't associated with a specific person. Using an IAM role, you can obtain temporary access keys to access AWS services and resources. IAM roles with temporary credentials are useful in the following situations:

  - **Federated user access** – Instead of creating a user, you can use identities from Directory Service, your enterprise user directory, or a web identity provider (IdP). These are known as *federated users*. AWS assigns a role to a federated user when the user requests access through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.
  - **Cross-account access** – You can use an IAM role in your account to grant another account permission to access your account's resources. For an example, see Tutorial: Delegate Access Across AWS Accounts Using IAM Roles in the *IAM User Guide*.
  - **AWS service access** – You can use an IAM role in your account to grant an AWS service permission to access your account's resources. For example, you can create a role that allows Amazon Redshift to load data stored in an Amazon S3 bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.
  - **Applications running on Amazon EC2** – For Amazon EC2 applications that need access to EventBridge, you can either store access keys in the EC2 instance or you can use an IAM role to manage temporary credentials. To assign an AWS role to an EC2 instance, you create an instance profile that is attached to the instance. An instance profile contains the role, and

it provides temporary credentials to applications running on the EC2 instance. For more information, see Using Roles for Applications on Amazon EC2 in the *IAM User Guide*.

Access control

To create or access EventBridge resources, you need both valid credentials and permissions. For example, to invoke AWS Lambda, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS) targets, you must have permissions to those services.

Managing access permissions to your Amazon EventBridge resources

You manage access to EventBridge resources such as rules or events by using identity-based or resource-based policies.

EventBridge resources

EventBridge resources and subresources have unique Amazon Resource Names (ARNs) associated with them. You use ARNs in EventBridge to create event patterns. For more information about ARNs, see Amazon Resource Names (ARN) and AWS Service Namespaces in the *Amazon Web Services General Reference*.

For a list of operations EventBridge provides for working with resources, see Amazon EventBridge permissions reference.

> **Note**
>
> Most services in AWS treat a colon (`:`) or a forward slash (`/`) as the same character in ARNs. However, EventBridge uses an exact match in event patterns and rules. Be sure to use the correct ARN characters when creating event patterns so that they match the ARN syntax in the event that you want to match.

The following table shows the resources in EventBridge.

Resource Type	ARN Format
Archive	`arn:aws:events:` *`region`*`:`*`account`*`:archive/` *`archive-name`*
Replay	`arn:aws:events:` *`region`*`:`*`account`*`:replay/`*`replay-name`*
Rule	`arn:aws:events:` *`region`*`:`*`account`*`:rule/[event-bus-name]/`*`rule-name`*
Event bus	`arn:aws:events:` *`region`*`:`*`account`*`:event-bus/` *`event-bus-name`*

Resource Type	ARN Format
All EventBridge resources	`arn:aws:events:*`
All EventBridge resources owned by the specified account in the specified Region	`arn:aws:events: region:account:*`

The following example shows how to indicate a specific rule (*myRule*) in your statement using its ARN.

```
"Resource": "arn:aws:events:us-east-1:123456789012:rule/myRule"
```

To specify all rules that belong to a specific account by using the asterisk (*) wildcard as follows.

```
"Resource": "arn:aws:events:us-east-1:123456789012:rule/*"
```

To specify all resources, or if a specific API action doesn't support ARNs, use the asterisk (*) wildcard in the `Resource` element as follows.

```
"Resource": "*"
```

To specify multiple resources or `PutTargets` in a single statement, separate their ARNs with commas as follows.

```
"Resource": ["arn1", "arn2"]
```

Resource ownership

An account owns the resources in the account, no matter who creates the resources. The resource owner is the account of the *principal entity*, the account root user, an IAM user or role that authenticates the request to create the resource. The following examples illustrate how this works:

- If you use the root user credentials of your account to create a rule, your account is the owner of the EventBridge resource.

- If you create an user in your account and grant permissions to create EventBridge resources to that user, the user can create EventBridge resources. However, your account, which the user belongs to, owns the EventBridge resources.
- If you create an IAM role in your account with permissions to create EventBridge resources, anyone who can assume the role can create EventBridge resources. Your account, which the role belongs to, owns the EventBridge resources.

Managing access to resources

A *permissions policy* describes who has access to what. The following section explains the available options for creating permissions policies.

> **Note**
>
> This section discusses using IAM in the context of EventBridge. It doesn't provide detailed information about the IAM service. For complete IAM documentation, see What is IAM? in the *IAM User Guide*. For information about IAM policy syntax and descriptions, see IAM policy reference in the *IAM User Guide*.

Policies attached to an IAM identity are referred to as *identity-based policies* (IAM policies) and policies attached to a resource are referred to as *resource-based policies*. In EventBridge, you can use both identity-based (IAM policies) and resource-based policies.

Topics

- Identity-based policies (IAM policies)
- Resource-based policies (IAM policies)

Identity-based policies (IAM policies)

You can attach policies to IAM identities. For example, you can do the following:

- **Attach a permissions policy to a user or a group in your account** – To grant a user permission to view rules in the Amazon CloudWatch console, attach a permissions policy to a user or group that the user belongs to.
- **Attach a permissions policy to a role (grant cross-account permissions)** – You can attach an identity-based permissions policy to an IAM role to grant cross-account permissions. For

example, the administrator in account A can create a role to grant cross-account permissions to another account B or an AWS service as follows:

1. Account A administrator creates an IAM role and attaches a permissions policy to the role that grants permission on resources in account A.
2. Account A administrator attaches a trust policy to the role identifying account B as the principal who can assume the role.
3. Account B administrator can then delegate permissions to assume the role to any users in account B. Doing this allows users in account B to create or access resources in account A. The principal in the trust policy can also be an AWS service principal to grant to an AWS service the permission needed to assume the role.

For more information about using IAM to delegate permissions, see Access Management in the *IAM User Guide*.

You can create specific IAM policies to restrict the calls and resources that users in your account have access to and then attach those policies to users. For more information about how to create IAM roles and to explore example IAM policy statements for EventBridge, see Managing access permissions to your Amazon EventBridge resources.

Resource-based policies (IAM policies)

When a rule runs in EventBridge, all of the targets associated with the rule are invoked, which means invoking the AWS Lambda functions, publishing to the Amazon SNS topics, or relaying the event to the Amazon Kinesis streams. To make API calls on the resources that you own, EventBridge needs the appropriate permission. For Lambda, Amazon SNS, and Amazon SQS resources, EventBridge uses resource-based policies. For Kinesis streams, EventBridge uses IAM roles.

For more information about how to create IAM roles and to explore example resource-based policy statements for EventBridge, see Using resource-based policies for Amazon EventBridge.

Specifying policy elements: actions, effects, and principals

For each EventBridge resource, EventBridge defines a set of API operations. To grant permissions for these API operations, EventBridge defines a set of actions that you can specify in a policy. Some API operations require permissions for more than one action to perform the API operation. For more information about resources and API operations, see EventBridge resources and Amazon EventBridge permissions reference.

The following are the basic policy elements:

- **Resource** – Use an Amazon Resource Name (ARN) to identify the resource that the policy applies to. For more information, see EventBridge resources.
- **Action** – Use keywords to identify resource operations that you want to allow or deny. For example, the `events:Describe` permission allows the user to perform the `Describe` operation.
- **Effect** – Specify either **allow** or **deny**. If you don't explicitly grant access to (allow) a resource, access is denied. You can also explicitly deny access to a resource, which you do to make sure that a user can't access it, even if a different policy grants access.
- **Principal** – In identity-based policies (IAM policies), the user that the policy is attached to is the implicit principal. For resource-based policies, you specify the user, account, service, or other entity that you want to receive permissions (applies to resource-based policies only).

For more information about IAM policy syntax and descriptions, see IAM JSON Policy Reference in the *IAM User Guide*.

For information about EventBridge API actions and the resources that they apply to, see Amazon EventBridge permissions reference.

Specifying conditions in a policy

When you grant permissions, you can use the access policy language to specify the conditions when a policy should take effect. For example, you might want a policy to be applied only after a specific date. For more information about specifying conditions in a policy language, see Condition in the *IAM User Guide*.

To define conditions, you use condition keys. There are AWS condition keys and EventBridge specific keys that you can use as appropriate. For a complete list of AWS keys, see Available Keys for Conditions in the *IAM User Guide*. For a complete list of EventBridge specific keys, see Using IAM policy conditions in Amazon EventBridge.

Using identity-based policies (IAM policies) for Amazon EventBridge

Identity-based policies are permissions policies that you can attach to IAM identities.

AWS managed policies for EventBridge

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. *Managed*, or predefined, policies grant the necessary permissions for common use cases, so you don't need to investigate what permissions are needed. For more information, see AWS managed policies in the *IAM User Guide*.

The following AWS managed policies that you can attach to users in your account are specific to EventBridge:

- **AmazonEventBridgeFullAccess** – Grants full access to EventBridge, including EventBridge Pipes, EventBridge Schemas and EventBridge Scheduler.
- **AmazonEventBridgeReadOnlyAccess** – Grants read-only access to EventBridge, including EventBridge Pipes, EventBridge Schemas and EventBridge Scheduler.

AWS managed policy: AmazonEventBridgeFullAccess

The AmazonEventBridgeFullAccess policy grants permissions to use all EventBridge actions, as well as the following permissions:

- `iam:CreateServiceLinkedRole` – EventBridge requires this permission to create the service role in your account for API destinations. This permission grants only the IAM service permissions to create a role in your account specifically for API destinations.
- `iam:PassRole` – EventBridge requires this permission to pass an invocation role to EventBridge to invoke the target of a rule.
- **Secrets Manager permissions** – EventBridge requires these permissions to manage secrets in your account when you provide credentials through the connection resource to authorize API Destinations.

To view the permissions for this policy, see AmazonEventBridgeFullAccess in the *AWS Managed Policy Reference*.

AWS managed policy: AmazonEventBridgeReadOnlyAccess

The AmazonEventBridgeReadOnlyAccess policy grants permissions to use all read EventBridge actions.

You can attach this policy to your users, groups, and roles. Use it when you need to view your EventBridge resources and their configuration without changing them. For example, use it to audit access or troubleshoot event delivery. You also get read access to EventBridge Pipes, EventBridge Schemas, and EventBridge Scheduler resources.

For the Custom Event Bus, you can use this policy to describe and list event buses, subscribers, and event sources. You can also read and list the resource policies attached to an event bus, and list the tags on EventBridge resources.

To view the permissions for this policy, see AmazonEventBridgeReadOnlyAccess in the *AWS Managed Policy Reference*.

AWS managed policy: AmazonEventBridgeApiDestinationsServiceRolePolicy

You can't attach AmazonEventBridgeApiDestinationsServiceRolePolicy to your IAM entities. This policy is attached to a service-linked role that allows EventBridge permissions to access AWS Secrets Manager resources on your behalf.

To view the permissions for this policy, see AmazonEventBridgeApiDestinationsServiceRolePolicy in the *AWS Managed Policy Reference*.

AWS managed policies: EventBridge Schemas

A schema defines the structure of events that are sent to EventBridge. EventBridge provides schemas for all events that are generated by AWS services. The following AWS managed policies specific to EventBridge Schemas are available:

- AmazonEventBridgeSchemasFullAccess

  You can attach the AmazonEventBridgeSchemasFullAccess policy to your IAM identities.

  Provides full access to EventBridge schemas.
- AmazonEventBridgeSchemasReadOnlyAccess

  You can attach the AmazonEventBridgeSchemasReadOnlyAccess policy to your IAM identities.

Provides read only access to EventBridge Schemas.

- AmazonEventBridgeSchemasServiceRolePolicy

  You can't attach AmazonEventBridgeSchemasServiceRolePolicy to your IAM entities. This policy is attached to a service-linked role that allows EventBridge permissions to managed rules created by EventBridge schemas.

AWS managed policies: EventBridge Scheduler

Amazon EventBridge Scheduler is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. For AWS managed policies that are specific to EventBridge Scheduler, see AWS managed policies for EventBridge Scheduler in the *EventBridge Scheduler User Guide*.

AWS managed policies: EventBridge Pipes

EventBridge Pipes connects event sources to targets. Pipes reduces the need for specialized knowledge and integration code when developing event driven architectures. This helps ensures consistency across your company's applications. The following AWS managed policies specific to EventBridge Pipes are available:

- AmazonEventBridgePipesFullAccess

  You can attach the AmazonEventBridgePipesFullAccess policy to your IAM identities.

  Provides full access to EventBridge Pipes.

  > **Note**
  >
  > This policy provides `iam:PassRole` – EventBridge Pipes requires this permission to pass an invocation role to EventBridge to create, and start pipes.

- AmazonEventBridgePipesReadOnlyAccess

  You can attach the AmazonEventBridgePipesReadOnlyAccess policy to your IAM identities.

  Provides read-only access to EventBridge Pipes.

- AmazonEventBridgePipesOperatorAccess

You can attach the AmazonEventBridgePipesOperatorAccess policy to your IAM identities.

Provides read-only and operator (that is, the ability to stop and start running Pipes) access to EventBridge Pipes.

Amazon EventBridge updates to AWS managed policies

View details about updates to AWS managed policies for EventBridge since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the EventBridge Document history page.

Change	Description	Date
AmazonEventBridgeReadOnlyAccess – Update to an existing policy	EventBridge added permissions to view Custom Event Bus subscribers and event bus resource policies, and to list the tags on EventBridge resources. The policy already granted permission to list tags on EventBridge Pipes, EventBridge Schemas, and EventBridge Scheduler resources. The following actions were added: • `events:DescribeSubscriber` • `events:ListSubscribers` • `events:GetResourcePolicy`	September 24, 2026

Change	Description	Date
	• `events:ListResourcePolicies` • `events:ListTagsForResource`	
AmazonEventBridgeApiDestinationsServiceRolePolicy – Updated policy	EventBridge updated policy to restrict the scope of permissions for Secrets Manager operations to the same account.	May 29, 2025
AmazonEventBridgeApiDestinationsServiceRolePolicy – Updated policy	EventBridge updated policy to grant AWS KMS encrypt and decrypt permissions via Secrets Manager. This enables EventBridge to update connection secret resources with new OAuth token value when access token refresh is required.	March 28, 2025
AmazonEventBridgeFullAccess – Updated policy	AWS GovCloud (US) Regions only The following permission is not included, as it is not used: • `iam:CreateServiceLinkedRole` permission for EventBridge Schema Registry	May 9, 2024

Change	Description	Date
AmazonEventBridgeSchemasFullAccess – Updated policy	AWS GovCloud (US) Regions only The following permission is not included, as it is not used: • `iam:CreateServiceLinkedRole` permission for EventBridge Schema Registry	May 9, 2024
AmazonEventBridgePipesFullAccess – New policy added	EventBridge added managed policy for full permissions for using EventBridge Pipes.	December 1, 2022
AmazonEventBridgePipesReadOnlyAccess – New policy added	EventBridge added managed policy for permissions to view EventBridge Pipes information resources.	December 1, 2022
AmazonEventBridgePipesOperatorAccess – New policy added	EventBridge added managed policy for permissions to view EventBridge Pipes information, as well as start and stop running pipes.	December 1, 2022
AmazonEventBridgeFullAccess – Update to an existing policy	EventBridge updated the policy to include permissions necessary for using EventBridge Pipes features.	December 1, 2022

Change	Description	Date
AmazonEventBridgeReadOnlyAccess – Update to an existing policy	EventBridge added permissions necessary for view EventBridge Pipes information resources. The following actions were added: • `pipes:DescribePipe` • `pipes:ListPipes` • `pipes:ListTagsForResource`	December 1, 2022
CloudWatchEventsReadOnlyAccess – Update to an existing policy	Updated to match AmazonEventBridgeReadOnlyAccess.	December 1, 2022
CloudWatchEventsFullAccess – Update to an existing policy	Updated to match AmazonEventBridgeFullAccess.	December 1, 2022

Change	Description	Date
AmazonEventBridgeFullAccess – Update to an existing policy	EventBridge updated the policy to include permissions necessary for using schemas and scheduler features. The following permissions were added: • EventBridge Schema Registry actions • EventBridge Scheduler actions • `iam:CreateServiceLinkedRole` permission for EventBridge Schema Registry • `iam:PassRole` permission for EventBridge Scheduler	November 10, 2022

Change	Description	Date
AmazonEventBridgeReadOnlyAccess – Update to an existing policy	EventBridge added permissions necessary for view schema and scheduler information resources. The following actions were added: • schemas:DescribeCodeBinding • schemas:DescribeDiscoverer • schemas:DescribeRegistry • schemas:DescribeSchema • schemas:ExportSchema • schemas:GetCodeBindingSource • schemas:GetDiscoveredSchema • schemas:GetResourcePolicy • schemas:ListDiscoverers • schemas:ListRegistries • schemas:ListSchemas • schemas:ListSchemaVersions	November 10, 2022

Change	Description	Date
	• schemas:ListTagsForResource • schemas:SearchSchemas • scheduler:GetSchedule • scheduler:GetScheduleGroup • scheduler:ListSchedules • scheduler:ListScheduleGroups • scheduler:ListTagsForResource	
AmazonEventBridgeReadOnlyAccess – Update to an existing policy	EventBridge added permissions necessary for view endpoint information. The following actions were added: • events:ListEndpoints • events:DescribeEndpoint	April 7, 2022

Change	Description	Date
AmazonEventBridgeReadOnlyAccess – Update to an existing policy	EventBridge added permissions necessary for view connection and API destination information. The following actions were added: • `events:DescribeConnection` • `events:ListConnections` • `events:DescribeApiDestination` • `events:ListApiDestinations`	March 4, 2021

Change	Description	Date
AmazonEventBridgeFullAccess – Update to an existing policy	EventBridge updated the policy to include `iam:CreateServiceLinkedRole` and AWS Secrets Manager permissions necessary for using API destinations. The following actions were added: • `secretsmanager:CreateSecret` • `secretsmanager:UpdateSecret` • `secretsmanager:DeleteSecret` • `secretsmanager:GetSecretValue` • `secretsmanager:PutSecretValue`	March 4, 2021
EventBridge started tracking changes	EventBridge started tracking changes for its AWS managed policies.	March 4, 2021

IAM roles for sending events to targets in Amazon EventBridge

To relay events to targets, EventBridge needs an IAM role.

To create an IAM role for sending events to EventBridge

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. To create an IAM role, follow the steps in Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide* . As you follow the steps, do the following:

- In **Role Name**, use a name that is unique within your account.
- In **Select Role Type**, choose **AWS Service Roles**, and then choose **Amazon EventBridge**. This grants EventBridge permissions to assume the role.
- In **Attach Policy**, choose **AmazonEventBridgeFullAccess**.

You can also create your own custom IAM policies to allow permissions for EventBridge actions and resources. You can attach these custom policies to the IAM users or groups that require those permissions. For more information about IAM policies, see Overview of IAM Policies in the *IAM User Guide*. For more information about managing and creating custom IAM policies, see Managing IAM Policies in the *IAM User Guide*.

Permissions required for EventBridge to access targets using IAM roles

EventBridge targets typically require IAM roles that grant permission to EventBridge to invoke the target. The following are some examples for various AWS services and targets. For others, use the EventBridge console to create a Rule and create a new Role which will be created with a policy with well-scoped permissions preconfigured.

Amazon SQS, Amazon SNS, and Lambda targets support both IAM execution roles and resource-based policies. CloudWatch Logs and EventBridge bus targets do not use roles, and permissions to EventBridge must be granted via a resource policy. API Gateway targets can use either resource policies or IAM roles.

API destinations

If the target is an API destination, the role that you specify must include a policy with the following statement:

- **Effect**: `Allow`
- **Action**: `events:InvokeApiDestination`
- **Resource**: `arn:aws:events:*:*:api-destination/*`

Kinesis streams

If the target is a Kinesis stream, the role used to send event data to that target must include a policy with the following statement:

- **Effect**: `Allow`

- **Action**: `kinesis:PutRecord`
- **Resource**: *

Systems Manager run commands

If the target is Systems Manager run command, and you specify one or more `InstanceIds` values for the command, the role that you specify must include a policy with the following statement:

- **Effect**: `Allow`
- **Action**: `ssm:SendCommand`
- **Resources**:
  - `arn:aws:ec2:region:accountId:instance/instanceId1`
  - `arn:aws:ec2:region:accountId:instance/instanceId2`
  - `arn:aws:ssm:region:*:document/documentName`

If the target is Systems Manager run command and you specify one or more tags to target instances, the role must include the following statement:

- **Effect**: `Allow`
- **Action**: `ssm:SendCommand`
- **Resources**:
  - `arn:aws:ec2:region:accountId:instance/*`
  - `arn:aws:ssm:region:accountId:managed-instance/*`
- **Condition**:

```
"StringEquals": {
  "aws:ResourceTag/tagKey": ["tagValues"]
}
```

The `aws:ResourceTag` condition key works for both EC2 instances and hybrid managed instances (on-premises nodes registered with Systems Manager). The `ssm:resourceTag/tagKey` condition key is also supported for the `SendCommand` action.

The role must also include the following statement to grant access to the Systems Manager document:

- **Effect**: `Allow`
- **Action**: `ssm:SendCommand`
- **Resource**: `arn:aws:ssm:region:*:document/documentName`

Step Functions state machines

If the target is an AWS Step Functions state machine, the role that you specify must include a policy with the following:

- **Effect**: `Allow`
- **Action**: `states:StartExecution`
- **Resource**: `arn:aws:states:*:*:stateMachine:*`

Amazon ECS tasks

If the target is an Amazon ECS task, the role that you specify must include the following policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask"
            ],
            "Resource": [
                "arn:aws-eusc:ecs:*:111122223333:task-definition/task-definition-
name"
            ],
            "Condition": {
                "ArnLike": {
                    "ecs:cluster": "arn:aws-
eusc:ecs:*:111122223333:cluster/cluster-name"
                }
            }
        },
        {
```

```
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": [
                "*"
            ],
            "Condition": {
                "StringLike": {
                    "iam:PassedToService": "ecs-tasks.amazonaws.com"
                }
            }
        }
    ]
}
```

The following policy allows built-in targets in EventBridge to perform Amazon EC2 actions on your behalf. You need to use the AWS Management Console to create rules with built-in targets.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "TargetInvocationAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:Describe*",
                "ec2:RebootInstances",
                "ec2:StopInstances",
                "ec2:TerminateInstances",
                "ec2:CreateSnapshot"
            ],
            "Resource": "*"
        }
    ]
}
```

The following policy allows EventBridge to relay events to the Kinesis streams in your account.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord"
            ],
            "Resource": "*"
        }
    ]
}
```

Customer-managed policy example: Using tagging to control access to rules

The following example shows a user policy that grant permissions for EventBridge actions. This policy works when you use the EventBridge API, AWS SDKs, or the AWS CLI.

You can grant users access to specific EventBridge rules while preventing them from accessing other rules. To do so, you tag both sets of rules and then use IAM policies that refer to those tags. For more information about tagging EventBridge resources, see Tagging resources in Amazon EventBridge.

You can grant an IAM policy to a user to allow access to only the rules with a particular tag. You choose which rules to grant access to by tagging them with that particular tag. For example, the following policy grants a user access to rules with the value of `Prod` for the tag key `Stack`.

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "events:*",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Stack": "Prod"
                }
```

```
            }
        }
    ]
}
```

For more information about using IAM policy statements, see Controlling Access Using Policies in the *IAM User Guide*.

Using resource-based policies for Amazon EventBridge

When a rule runs in EventBridge, all of the targets associated with the rule are invoked. Rules can invoke AWS Lambda functions, publish to Amazon SNS topics, or relay the event to Kinesis streams. To make API calls against the resources you own, EventBridge needs the appropriate permissions. For Amazon CloudWatch Logs resources, EventBridge uses resource-based policies. For Lambda, Amazon SNS, and Amazon SQS resources, EventBridge can use either an IAM execution role or a resource-based policy. For Kinesis streams, EventBridge uses identity-based policies.

> **Important**
>
> For Amazon CloudWatch Logs targets, do not specify a `RoleArn` in the target configuration. For Lambda, Amazon SNS, and Amazon SQS targets, you can use either an IAM execution role or a resource-based policy. If you specify a `RoleArn` for Lambda, Amazon SNS, or Amazon SQS targets, ensure the role has the required permissions (for example, `lambda:InvokeFunction`, `sns:Publish`, or `sqs:SendMessage`). If no execution role is configured, EventBridge uses resource-based policies on the target resource.

You use the AWS CLI to add permissions to your targets. For information about how to install and configure the AWS CLI, see Getting Set Up with the AWS Command Line Interface in the *AWS Command Line Interface User Guide*.

Topics

- Amazon API Gateway permissions
- CloudWatch Logs permissions
- AWS Lambda permissions
- Amazon SNS permissions
- Amazon SQS permissions
- EventBridge Pipes specifics

Amazon API Gateway permissions

To invoke your Amazon API Gateway endpoint by using a EventBridge rule, add the following permission to the policy of your API Gateway endpoint.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "events.amazonaws.com"
            },
            "Action": "execute-api:Invoke",
            "Condition": {
                "ArnEquals": {
                "aws:SourceArn": "arn:aws-eusc:events:eusc-de-
east-1:123456789012:rule/rule-name"
                }
            },
            "Resource": [
            "arn:aws-eusc:execute-api:eusc-de-east-1:123456789012:API-id/stage/
GET/api"
            ]
        }
    ]
}
```

CloudWatch Logs permissions

When CloudWatch Logs is the target of a rule, EventBridge creates log streams, and CloudWatch Logs stores the text from the events as log entries. To allow EventBridge to create the log stream and log the events, CloudWatch Logs must include a resource-based policy that enables EventBridge to write to CloudWatch Logs.

If you use the AWS Management Console to add CloudWatch Logs as the target of a rule, the resource-based policy is created automatically. If you use the AWS CLI to add the target, and the policy doesn't already exist, you must create it.

The following resource-based policy example allows EventBridge to write to all log groups that have names that start with `/aws/events/`. If you use a different naming policy for these types of logs, adjust the resource ARN accordingly.

```
{
  "Effect": "Allow",
  "Principal": {
    "Service": [
      "events.amazonaws.com",
      "delivery.logs.amazonaws.com"
    ]
  },
  "Action": [
    "logs:CreateLogStream",
    "logs:PutLogEvents"
  ],
  "Resource": "arn:aws:logs:region:account-id:log-group:/aws/events/*:*"
}
```

To create a resource policy for CloudWatch Logs using the AWS CLI

- At a command prompt, enter the following command.

```
aws logs put-resource-policy --policy-name EventBridgeToCWLogsPolicy \
--policy-document '{"Version":"2012-10-17",       "Statement":
[{"Effect":"Allow","Principal":{"Service":
["events.amazonaws.com","delivery.logs.amazonaws.com"]},"Action":
["logs:CreateLogStream","logs:PutLogEvents"],"Resource":"arn:aws:logs:region:account-
id:log-group:/aws/events/*:*"}]}'
```

For more information, see PutResourcePolicy in the *CloudWatch Logs API Reference guide*.

AWS Lambda permissions

To invoke your AWS Lambda function by using a EventBridge rule, add the following permission to the policy of your Lambda function.

> **Note**
>
> Alternatively, you can configure an IAM execution role on the target with `lambda:InvokeFunction` permission. Resource-based policies are required only when no execution role is configured on the target.

```
{
  "Effect": "Allow",
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:region:account-id:function:function-name",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:region:account-id:rule/rule-name"
    }
  },
  "Sid": "InvokeLambdaFunction"
}
```

To add the above permissions that enable EventBridge to invoke Lambda functions using the AWS CLI

- At a command prompt, enter the following command.

```
aws lambda add-permission --statement-id "InvokeLambdaFunction" \
--action "lambda:InvokeFunction" \
--principal "events.amazonaws.com" \
--function-name "arn:aws:lambda:region:account-id:function:function-name" \
--source-arn "arn:aws:events:region:account-id:rule/rule-name"
```

For more information about setting permissions that enable EventBridge to invoke Lambda functions, see AddPermission and Using Lambda with Scheduled Events in the *AWS Lambda Developer Guide*.

Amazon SNS permissions

To allow EventBridge to publish to an Amazon SNS topic, use the `aws sns get-topic-attributes` and the `aws sns set-topic-attributes` commands.

To add permissions that enable EventBridge to publish SNS topics

1. To list the attributes of an SNS topic, use the following command.

```
aws sns get-topic-attributes --topic-arn "arn:aws:sns:region:account-id:topic-name"
```

The following example shows the result of a new SNS topic.

```
{
    "Attributes": {
        "SubscriptionsConfirmed": "0",
        "DisplayName": "",
        "SubscriptionsDeleted": "0",
        "EffectiveDeliveryPolicy": "{\"http\":{\"defaultHealthyRetryPolicy\":
{\"minDelayTarget\":20,\"maxDelayTarget\":20,\"numRetries\":3,\"numMaxDelayRetries
\":0,\"numNoDelayRetries\":0,\"numMinDelayRetries\":0,\"backoffFunction\":\"linear
\"},\"disableSubscriptionOverrides\":false}}",
        "Owner": "account-id",
        "Policy": "{\"Version\":\"2012-10-17\",       \"Id\":\"__default_policy_ID
\",\"Statement\":[{\"Sid\":\"__default_statement_ID\",\"Effect\":\"Allow
\",\"Principal\":{\"AWS\":\"*\"},\"Action\":[\"SNS:GetTopicAttributes\",
\"SNS:SetTopicAttributes\",\"SNS:AddPermission\",\"SNS:RemovePermission\",
\"SNS:DeleteTopic\",\"SNS:Subscribe\",\"SNS:ListSubscriptionsByTopic\",
\"SNS:Publish\"],\"Resource\":\"arn:aws:sns:region:account-id:topic-name\",
\"Condition\":{\"StringEquals\":{\"AWS:SourceAccount\":\"account-id\"}}}]}",
        "TopicArn": "arn:aws:sns:region:account-id:topic-name",
        "SubscriptionsPending": "0"
    }
}
```

2. Use a JSON to string converter to convert the following statement to a string.

```
{
  "Sid": "PublishEventsToMyTopic",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "sns:Publish",
  "Resource": "arn:aws:sns:region:account-id:topic-name"
}
```

After you convert the statement to a string, it looks like the following example.

```
{\"Sid\":\"PublishEventsToMyTopic\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"events.amazonaws.com\"},\"Action\":\"sns:Publish\",\"Resource\":
\"arn:aws:sns:region:account-id:topic-name\"}
```

3. Add the string you created in the previous step to the `"Statement"` collection inside the `"Policy"` attribute.
4. Use the `aws sns set-topic-attributes` command to set the new policy.

```
aws sns set-topic-attributes --topic-arn "arn:aws:sns:region:account-id:topic-name"
 \
--attribute-name Policy \
--attribute-value "{\"Version\":\"2012-10-17\",       \"Id\":\"__default_policy_ID
\",\"Statement\":[{\"Sid\":\"__default_statement_ID\",\"Effect\":\"Allow
\",\"Principal\":{\"AWS\":\"*\"},\"Action\":[\"SNS:GetTopicAttributes\",
\"SNS:SetTopicAttributes\",\"SNS:AddPermission\",\"SNS:RemovePermission\",
\"SNS:DeleteTopic\",\"SNS:Subscribe\",\"SNS:ListSubscriptionsByTopic\",
\"SNS:Publish\"],\"Resource\":\"arn:aws:sns:region:account-id:topic-name\",
\"Condition\":{\"StringEquals\":{\"AWS:SourceAccount\":\"account-id\"}}},
 {\"Sid\":\"PublishEventsToMyTopic\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"events.amazonaws.com\"},\"Action\":\"sns:Publish\",\"Resource\":
\"arn:aws:sns:region:account-id:topic-name\"}]}"
```

For more information, see the SetTopicAttributes action in the *Amazon Simple Notification Service API Reference*.

Amazon SQS permissions

To allow an EventBridge rule to invoke an Amazon SQS queue, use the `aws sqs get-queue-attributes` and `aws sqs set-queue-attributes` commands.

If the policy for the SQS queue is empty, you first need to create a policy and then you can add the permissions statement to it. A new SQS queue has an empty policy.

If the SQS queue already has a policy, you need to copy the original policy and combine it with a new statement to add the permissions statement to it.

To add permissions that enable EventBridge rules to invoke an SQS queue

1. To list SQS queue attributes. At a command prompt, enter the following command.

```
aws sqs get-queue-attributes \
--queue-url https://sqs.region.amazonaws.com/account-id/queue-name \
--attribute-names Policy
```

2. Add the following statement.

```
{
      "Sid": "AWSEvents_custom-eventbus-ack-sqs-rule_dlq_sqs-rule-target",
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": "sqs:SendMessage",
      "Resource": "arn:aws:sqs:region:account-id:queue-name",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:events:region:account-id:rule/bus-name/rule-
name"
        }
      }
    }
```

3. Use a JSON to string converter to convert the preceding statement into a string. After you convert the policy to a string, it looks like the following.

```
{\"Sid\": \"EventsToMyQueue\", \"Effect\": \"Allow\", \"Principal\": {\"Service
\": \"events.amazonaws.com\"}, \"Action\": \"sqs:SendMessage\", \"Resource\":
 \"arn:aws:sqs:region:account-id:queue-name\", \"Condition\": {\"ArnEquals\":
 {\"aws:SourceArn\": \"arn:aws:events:region:account-id:rule/rule-name\"}}
```

4. Create a file called `set-queue-attributes.json` with the following content.

```
{
    "Policy": "{\"Version\":\"2012-10-17\",       \"Id\":
\"arn:aws:sqs:region:account-id:queue-name/SQSDefaultPolicy\",\"Statement\":
[{\"Sid\": \"EventsToMyQueue\", \"Effect\": \"Allow\", \"Principal\": {\"Service
\": \"events.amazonaws.com\"}, \"Action\": \"sqs:SendMessage\", \"Resource\":
 \"arn:aws:sqs:region:account-id:queue-name\", \"Condition\": {\"ArnEquals\":
 {\"aws:SourceArn\": \"arn:aws:events:region:account-id:rule/rule-name\"}}}]}"
}
```

5. Set the policy attribute by using the `set-queue-attributes.json` file you just created as the input, as shown in the following command.

```
aws sqs set-queue-attributes \
--queue-url https://sqs.region.amazonaws.com/account-id/queue-name \
--attributes file://set-queue-attributes.json
```

For more information, see Amazon SQS Policy Examples in the *Amazon Simple Queue Service Developer Guide*.

EventBridge Pipes specifics

EventBridge Pipes does not support resource-based policies and has no APIs which support resource based policy conditions.

However, if you configure pipe access through an interface VPC endpoint, that VPC endpoint supports resource policies that enable you to manage access to EventBridge Pipe APIs. For more information, see the section called “Interface VPC Endpoints”

Cross-service confused deputy prevention in Amazon EventBridge

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon EventBridge gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws-eusc:`*`servicename`*`:*:`*`123456789012`*`:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

Event buses

For EventBridge event bus rule targets, the value of `aws:SourceArn` must be the rule ARN.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in EventBridge to prevent the confused deputy problem. This example is for use in a role trust policy, for a role used by an EventBridge rule.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Sid": "ConfusedDeputyPreventionExamplePolicy",
        "Effect": "Allow",
        "Principal": {
            "Service": "events.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "ArnLike": {
                "aws:SourceArn": "arn:aws-eusc:events:*:123456789012:rule/myRule"
            },
            "StringEquals": {
                "aws:SourceAccount": "123456789012"
            }
        }
    }
}
```

EventBridge Pipes

For EventBridge Pipes, the value of `aws:SourceArn` must be the pipe ARN.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in EventBridge to prevent the confused deputy problem. This example is for use in a role trust policy, for a role used by EventBridge Pipes.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Sid": "ConfusedDeputyPreventionExamplePolicy",
        "Effect": "Allow",
        "Principal": {
            "Service": "pipes.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "ArnLike": {
                "aws:SourceArn": "arn:aws-eusc:pipes:*:123456789012:pipe/example"
            },
            "StringEquals": {
                "aws:SourceAccount": "123456789012"
            }
        }
    }
}
```

Resource-based policies for Amazon EventBridge schemas

The EventBridge schema registry supports resource-based policies. A *resource-based policy* is a policy that is attached to a resource rather than to an IAM identity. For example, in Amazon Simple Storage Service (Amazon S3), a resource policy is attached to an Amazon S3 bucket.

For more information about EventBridge Schemas and resource-based policies, see the following.

- Amazon EventBridge Schemas REST API Reference
- Identity-Based Policies and Resource-Based Policies in the IAM User Guide

Supported APIs for resource-based policies

You can use the following APIs with resource-based policies for the EventBridge schema registry.

- `DescribeRegistry`
- `UpdateRegistry`
- `DeleteRegistry`
- `ListSchemas`
- `SearchSchemas`
- `DescribeSchema`
- `CreateSchema`
- `DeleteSchema`
- `UpdateSchema`
- `ListSchemaVersions`
- `DeleteSchemaVersion`
- `DescribeCodeBinding`
- `GetCodeBindingSource`
- `PutCodeBinding`

Example policy granting all supported actions to an AWS account

For the EventBridge schema registry, you must always attach a resource-based policy to a registry. To grant access to a schema, you specify the schema ARN and the registry ARN in the policy.

To grant a user access to all available APIs for EventBridge Schemas, use a policy similar to the following, replacing the `"Principal"` with the account ID of the account you want to grant access.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Test",
            "Effect": "Allow",
            "Action": [
                "schemas:*"
            ],
            "Principal": {
                "AWS": [
                    "109876543210"
                ]
            },
            "Resource": [
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:registry/
default",
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:schema/
default*"
            ]
        }
    ]
}
```

Example policy granting read-only actions to an AWS account

The following example grants access to an account for only the read-only APIs for EventBridge schemas.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Sid": "Test",
            "Effect": "Allow",
            "Action": [
                "schemas:DescribeRegistry",
                "schemas:ListSchemas",
                "schemas:SearchSchemas",
                "schemas:DescribeSchema",
                "schemas:ListSchemaVersions",
                "schemas:DescribeCodeBinding",
                "schemas:GetCodeBindingSource"
            ],
            "Principal": {
                "AWS": [
                    "109876543210"
                ]
            },
            "Resource": [
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:registry/
default",
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:schema/
default*"
            ]
        }
    ]
}
```

Example policy granting all actions to an organization

You can use resource-based policies with the EventBridge schema registry to grant access to an organization. For more information, see the AWS Organizations User Guide. The following example grants organization with an ID of `o-a1b2c3d4e5` access to the schema registry.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Test",
            "Effect": "Allow",
```

```
            "Action": [
                "schemas:*"
            ],
            "Principal": "*",
            "Resource": [
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:registry/
default",
                "arn:aws-eusc:schemas:eusc-de-east-1:012345678901:schema/
default*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": [
                        "o-a1b2c3d4e5"
                    ]
                }
            }
        }
    ]
}
```

Amazon EventBridge permissions reference

To specify an action in an EventBridge policy, use the `events:` prefix followed by the API operation name, as shown in the following example.

```
"Action": "events:PutRule"
```

To specify multiple actions in a single statement, separate them with commas as follows.

```
"Action": ["events:action1", "events:action2"]
```

To specify multiple actions, you can also use wildcards. For example, you can specify all actions that begin with the word `"Put"` as follows.

```
"Action": "events:Put*"
```

To specify all EventBridge API actions, use the * wildcard as follows.

```
"Action": "events:*"
```

The following table lists the EventBridge API operations and corresponding actions that you can specify in an IAM policy.

EventBridge API operation	Required permissions	Description
DeleteRule	`events:DeleteRule`	Required to delete a rule.
DescribeEventBus	`events:DescribeEventBus`	Required to list accounts that are allowed to write events to the current account's event bus.
DescribeRule	`events:DescribeRule`	Required to list the details about a rule.
DisableRule	`events:DisableRule`	Required to disable a rule.
EnableRule	`events:EnableRule`	Required to enable a rule.

EventBridge API operation	Required permissions	Description
ListRuleNamesByTarget	events:ListRuleNamesByTarget	Required to list rules associated with a target.
ListRules	events:ListRules	Required to list all rules in your account.
ListTagsForResource	events:ListTagsForResource	Required to list all tags associated with an EventBridge resource. Currently, only rules can be tagged.
ListTargetsByRule	events:ListTargetsByRule	Required to list all targets associated with a rule.
PutEvents	events:PutEvents	Required to add custom events that can be matched to rules.
PutPermission	events:PutPermission	Required to give another account permission to write events to this account's default event bus.
PutRule	events:PutRule	Required to create or update a rule.
PutTargets	events:PutTargets	Required to add targets to a rule.
RemovePermission	events:RemovePermission	Required to revoke another account's permissions for writing events to this account's default event bus.
RemoveTargets	events:RemoveTargets	Required to remove a target from a rule.

EventBridge API operation	Required permissions	Description
TestEventPattern	`events:TestEventPattern`	Required to test an event pattern against a given event.

Using IAM policy conditions in Amazon EventBridge

To grant permissions, you use the IAM policy language in a policy statement to specify the conditions when a policy should take effect. For example, you can have a policy that is applied only after a specific date.

A condition in a policy consists of key-value pairs. Condition keys aren't case sensitive.

If you specify multiple conditions or keys in a single condition, all conditions and keys must be met for EventBridge to grant permission. If you specify a single condition with multiple values for one key, EventBridge grants permission if one of the values is met.

You can use placeholders or *policy variables* when you specify conditions. For more information, see Policy Variables in the *IAM User Guide*. For more information about specifying conditions in an IAM policy language, see Condition in the *IAM User Guide*.

By default, IAM users and roles can't access the events in your account. To access events, a user must be authorized for the `PutRule` API action. If an IAM user or role is authorized for the `events:PutRule` action, they can create a rule that matches certain events. However, for the rule to be useful, the user must also have permissions for the `events:PutTargets` action because, if you want the rule to do more than publish a CloudWatch metric, you must also add a target to a rule.

You can provide a condition in the policy statement of an IAM user or role that allows the user or role to create a rule that only matches a specific set of sources and event types. To grant access to specific sources and types of events, use the `events:source` and `events:detail-type` condition keys.

Similarly, you can provide a condition in the policy statement of an IAM user or role that allows the user or role to create a rule that only matches a specific resource in your accounts. To grant access to a specific resource, use the `events:TargetArn` condition key.

EventBridge condition keys

The following table shows the condition keys and key and value pairs that you can use in a policy in EventBridge.

Condition key	Key value pair	Evaluation types
aws:SourceAccount	The account in which the rule specified by `aws:SourceArn` exists.	Account Id, Null
aws:SourceArn	The ARN of the rule that is sending the event.	ARN, Null
events:creatorAccount	`"events:creatorAccount":"` *`creatorAccount`* `"` For *`creatorAccount`* , use the account ID for the account that created the rule. Use this condition to authorize API calls on rules from a specific account.	creatorAccount, Null
events:detail-type	`"events:detail-type":"` *`detail-type`* `"` Where *`detail-type`* is the literal string for the **detail-type** field of the event such as `"AWS API Call via CloudTrail"` and `"EC2 Instance State-change Notification"` .	Detail Type, Null
events: detail.eventTypeCode	`"events:detail.eventTypeCode":"` *`eventTypeCode`* `"` For *`eventTypeCode`* , use the literal string for the **detail.eventTypeCode** field of the event, such as `"AWS_ABUSE_DOS_REPORT"` .	eventTypeCode, Null

Condition key	Key value pair	Evaluation types
events: detail.service	`"events:detail.service":" service"` For *service*, use the literal string for the **detail.service** field of the event, such as `"ABUSE"`.	service, Null
events: detail.userIdentity.principalId	`"events:detail.userIdentity.principalId":" principal-id "` For *principal-id* , use the literal string for the **detail.userIdentity.principalId** field of the event with detail-type `"AWS API Call via CloudTrail"` such as `"AROAIDPPEZS35WEXAMPLE:AssumedRoleSessionName."` .	Principal Id, Null
events:eventBusInvocation	`"events:eventBusInvocation":" boolean"` For *boolean*, use true when a rule sends an event to a target that is an event bus in another account. Use false when when a `PutEvents` API call is used.	eventBusInvocation, Null

Condition key	Key value pair	Evaluation types
events:ManagedBy	Used internally by AWS services. For a rule created by an AWS service on your behalf, the value is the principal name of the service that created the rule.	Used to scope permissions to rules created by AWS services on your behalf (managed rules). For example, the `AmazonS3FilesFullAccess` managed policy uses this key to allow creation of only managed EventBridge rules.
events:source	`"events:source":" source "` Use *source* for the literal string for the source field of the event such as `"aws.ec2"` or `"aws.s3"`. For more possible values for *source*, see the example events in Events from AWS services.	Source, Null
events:TargetArn	`"events:TargetArn":" target-arn "` For *target-arn* , use the ARN of the target for the rule, for example `"arn:aws:lambda:*:*:function:*"` .	ArrayOfARN, Null

For example policy statements for EventBridge, see Managing access permissions to your Amazon EventBridge resources.

Topics

- EventBridge Pipes specifics
- Example: Using the creatorAccount condition

- Example: Using the eventBusInvocation condition
- Example: Limiting access to a specific source
- Example: Defining multiple sources that can be used in an event pattern individually
- Example: Ensuring that the source is defined in the event pattern
- Example: Defining a list of allowed sources in an event pattern with multiple sources
- Example: Limiting PutRule access by detail.service
- Example: Limiting PutRule access by detail.eventTypeCode
- Example: Ensuring that only AWS CloudTrail events for API calls from a certain PrincipalId are allowed
- Example: Limiting access to targets

EventBridge Pipes specifics

EventBridge Pipes does not support any additional IAM policy condition keys.

Example: Using the `creatorAccount` condition

The following example policy statement shows how to use the `creatorAccount` condition in a policy to only allow rules to be created if the account specified as the `creatorAccount` is the account that created the rule.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleForOwnedRules",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*",
            "Condition": {
                "StringEqualsIfExists": {
                    "events:creatorAccount": "${aws:PrincipalAccount}"
                }
            }
```

```
        }
    ]
}
```

Example: Using the `eventBusInvocation` condition

The `eventBusInvocation` indicates whether the invocation originates from a cross-account target or a `PutEvents` API request. The value is **true** when the invocation results from a rule that include a cross-account target, such as when the target is an event bus in another account. The value is **false** when the invocation results from a `PutEvents` API request. The following example indicates an invocation from a cross-account target.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCrossAccountInvocationEventsOnly",
      "Effect": "Allow",
      "Action": "events:PutEvents",
      "Resource": "*",
      "Condition": {
        "BoolIfExists": {
          "events:eventBusInvocation": "true"
        }
      }
    }
  ]
}
```

Example: Limiting access to a specific source

The following example policies can be attached to an IAM user. Policy A allows the `PutRule` API action for all events, whereas Policy B allows `PutRule` only if the event pattern of the rule being created matches Amazon EC2 events.

Policy A: allow all events

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleForAllEvents",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*"
        }
    ]
    }
```

Policy B:—allow events only from Amazon EC2

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
    {
    "Sid": "AllowPutRuleForAllEC2Events",
    "Effect": "Allow",
    "Action": "events:PutRule",
    "Resource": "*",
    "Condition": {
    "ForAllValues:StringEquals": {
    "events:source": "aws.ec2"
    }
    }
    }
    ]
    }
```

`EventPattern` is a mandatory argument to `PutRule`. Hence, if the user with Policy B calls `PutRule` with an event pattern like the following.

```
{
```

```
    "source": [ "aws.ec2" ]
}
```

The rule would be created because the policy allows for this specific source: that is, `"aws.ec2"`. However, if the user with Policy B calls `PutRule` with an event pattern like the following, the rule creation would be denied because the policy doesn't allow for this specific source: that is, `"aws.s3"`.

```
{
    "source": [ "aws.s3" ]
}
```

Essentially, the user with Policy B is only allowed to create a rule that would match the events originating from Amazon EC2; hence, they're only allowed access to the events from Amazon EC2.

See the following table for a comparison of Policy A and Policy B.

Event Pattern	Allowed by Policy A	Allowed by Policy B
`{ "source": [ "aws.ec2" ] }`	Yes	Yes
`{ "source": [ "aws.ec2", "aws.s3" ] }`	Yes	No (Source aws.s3 isn't allowed)
`{ "source": [ "aws.ec2" ], "detail-type": [ "EC2 Instance State-change Notification" ] }`	Yes	Yes

Event Pattern	Allowed by Policy A	Allowed by Policy B
`{ "detail-type": [ "EC2 Instance State-change Notification" ] }`	Yes	Yes. `ForAllValues` evaluates an empty set and returns true because the event pattern omits `events:source` . See the following **Important** note.

⚠ Policy B source restriction can be bypassed without a `Null` check

Policy B restricts the events a rule can match only when the rule's event pattern includes the `events:source` field. Because the condition uses the `ForAllValues` set operator without a `Null` check, an event pattern that omits `source` entirely causes the condition to evaluate over an empty set and return true. For example, an event pattern that specifies only `detail-type` triggers this behavior. A principal can therefore bypass the intended source restriction by creating rules whose event patterns don't specify a source.
To require that a rule's event pattern always includes a `source`, combine `ForAllValues` with a `Null` condition on `events:source`, as shown in Example: Ensuring that the source is defined in the event pattern. For more information about how set operators evaluate a missing condition key, see Creating a condition with multiple keys or values in the *IAM User Guide*.

Example: Defining multiple sources that can be used in an event pattern individually

The following policy allows an IAM user or role to create a rule where the source in the `EventPattern` is either Amazon EC2 or Amazon ECS.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
    {
    "Sid": "AllowPutRuleIfSourceIsEC2OrECS",
```

```
    "Effect": "Allow",
    "Action": "events:PutRule",
    "Resource": "*",
    "Condition": {
    "ForAllValues:StringEquals": {
    "events:source": [
    "aws.ec2",
    "aws.ecs"
    ]
    }
    }
    }
    ]
    }
```

The following table shows some examples of event patterns that are allowed or denied by this policy.

Event pattern	Allowed by the policy
`{ "source": [ "aws.ec2" ] }`	Yes
`{ "source": [ "aws.ecs" ] }`	Yes
`{ "source": [ "aws.s3" ] }`	No
`{ "source": [ "aws.ec2", "aws.ecs" ] }`	No
`{`	No

Event pattern	Allowed by the policy

<pre>"detail-type": ["AWS API Call via CloudTrail"]
}</pre>	

Example: Ensuring that the source is defined in the event pattern

The following policy allows users to only create rules with `EventPatterns` that have the source field. With this policy, an IAM user or role can't create a rule with an `EventPattern` that doesn't provide a specific source.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleIfSourceIsSpecified",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*",
            "Condition": {
                "Null": {
                    "events:source": "false"
                }
            }
        }
    ]
}
```

The following table shows some examples of event patterns that are allowed or denied by this policy.

Event Pattern	Allowed by the Policy

<pre>{
"source": ["aws.ec2"],</pre>
	Yes

Event Pattern	Allowed by the Policy

<pre>"detail-type": ["EC2 Instance State-change Notification"]
}</pre>	

<pre>{
"source": ["aws.ecs", "aws.ec2"]
}</pre>
	Yes

<pre>{
"detail-type": ["EC2 Instance State-change Notification"]
}</pre>
	No

Example: Defining a list of allowed sources in an event pattern with multiple sources

The following policy allows users to create rules with `EventPatterns` that have multiple sources in them. Each source in the event pattern must be a member of the list provided in the condition. When you use the `ForAllValues` condition, make sure that at least one of the items in the condition list is defined.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleIfSourceIsSpecifiedAndIsEitherS3OrEC2OrBoth",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringEquals": {
```

```
                    "events:source": [ "aws.ec2", "aws.s3" ]
                },
                "Null": {
                    "events:source": "false"
                }
            }
        }
    ]
}
```

The following table shows some examples of event patterns that are allowed or denied by this policy.

Event Pattern	Allowed by the Policy
`{ "source": [ "aws.ec2" ] }`	Yes
`{ "source": [ "aws.ec2", "aws.s3" ] }`	Yes
`{ "source": [ "aws.ec2", "aws.autoscaling" ] }`	No
`{ "detail-type": [ "EC2 Instance State-change Notification" ] }`	No

Example: Limiting PutRule access by detail.service

You can restrict an IAM user or role to creating rules only for events that have a certain value in the events:details.service field. The value of events:details.service isn't necessarily the name of an AWS service.

This policy condition is helpful when you work with events from AWS Health that relate to security or abuse. By using this policy condition, you can limit access to these sensitive alerts to only those users who need to see them.

For example, the following policy allows the creation of rules only for events where the value of events:details.service is ABUSE.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleEventsWithDetailServiceEC2",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "events:detail.service": "ABUSE"
                }
            }
        }
    ]
}
```

Example: Limiting PutRule access by detail.eventTypeCode

You can restrict an IAM user or role to creating rules only for events that have a certain value in the events:details.eventTypeCode field. This policy condition is helpful when you work with events from AWS Health that relate to security or abuse. By using this policy condition, you can limit access to these sensitive alerts to only those users who need to see them.

For example, the following policy allows the creation of rules only for events where the value of `events:details.eventTypeCode` is `AWS_ABUSE_DOS_REPORT`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutRuleEventsWithDetailServiceEC2",
            "Effect": "Allow",
            "Action": "events:PutRule",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "events:detail.eventTypeCode": "AWS_ABUSE_DOS_REPORT"
                }
            }
        }
    ]
}
```

Example: Ensuring that only AWS CloudTrail events for API calls from a certain `PrincipalId` are allowed

All AWS CloudTrail events have the PrincipalId of the user who made the API call in the `detail.userIdentity.principalId` path of an event. Using the `events:detail.userIdentity.principalId` condition key, you can limit the access of IAM users or roles to the CloudTrail events for only those coming from a specific account.

```
"Version": "2012-10-17",
"Statement": [
    {
        "Sid": "AllowPutRuleOnlyForCloudTrailEventsWhereUserIsASpecificIAMUser",
        "Effect": "Allow",
        "Action": "events:PutRule",
        "Resource": "*",
        "Condition": {
            "StringEquals": {
```

```
                    "events:detail-type": [ "AWS API Call via CloudTrail" ],
                    "events:detail.userIdentity.principalId":
 [ "AIDAJ45Q7YFFAREXAMPLE" ]
                }
            }
        }
    ]
}
```

The following table shows some examples of event patterns that are allowed or denied by this policy.

Event pattern	Allowed by the policy
`{ "detail-type": [ "AWS API Call via CloudTrail" ] }`	No
`{ "detail-type": [ "AWS API Call via CloudTrail" ], "detail.userIdentity.principalId": [ "AIDAJ45Q7YFFAREXAMPLE" ] }`	Yes
`{ "detail-type": [ "AWS API Call via CloudTrail" ], "detail.userIdentity.principalId": [ "AROAIDPPEZS35WEXAMPLE:AssumedRoleSessionName" ] }`	No

Example: Limiting access to targets

If an IAM user or role has `events:PutTargets` permission, they can add any target under the same account to the rules that they are allowed to access. The following policy limits users to adding targets to only a specific rule: `MyRule` under account `123456789012`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutTargetsOnASpecificRule",
            "Effect": "Allow",
            "Action": "events:PutTargets",
            "Resource": "arn:aws-eusc:events:eusc-de-east-1:123456789012:rule/
MyRule"
        }
    ]
}
```

To limit what target can be added to the rule, use the `events:TargetArn` condition key. You can limit targets to only Lambda functions, as in the following example.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowPutTargetsOnASpecificRuleAndOnlyLambdaFunctions",
            "Effect": "Allow",
            "Action": "events:PutTargets",
            "Resource": "arn:aws-eusc:events:eusc-de-
east-1:123456789012:rule/rule-name",
            "Condition": {
            "ForAnyValue:ArnLike": {
                    "events:TargetArn": "arn:aws-eusc:lambda:*:*:function:*"
                }
            }
```

```
        }
    ]
}
```

Using service-linked roles for EventBridge

Amazon EventBridge uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to EventBridge. Service-linked roles are predefined by EventBridge and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up EventBridge easier because you don't have to manually add the necessary permissions. EventBridge defines the permissions of its service-linked roles, and unless defined otherwise, only EventBridge can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your EventBridge resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Using roles to create secrets for API destinations in EventBridge

The following topic details usage of the **AWSServiceRoleForAmazonEventBridgeApiDestinations** service-linked role.

Service-linked role permissions for EventBridge

EventBridge uses the service-linked role named **AWSServiceRoleForAmazonEventBridgeApiDestinations** – Enables access to the Secrets Manager Secrets created by EventBridge.

The **AWSServiceRoleForAmazonEventBridgeApiDestinations** service-linked role trusts the following services to assume the role:

- `apidestinations.events.amazonaws.com`

The role permissions policy named **AmazonEventBridgeApiDestinationsServiceRolePolicy** allows EventBridge to complete the following actions on the specified resources:

- Action: `create, describe, update and delete secrets; get and put secret values` on `secrets created for all connections by EventBridge`

You must configure permissions to allow your users, groups, or roles to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for EventBridge

You don't need to manually create a service-linked role. When you create a connection in the AWS Management Console, the AWS CLI, or the AWS API, EventBridge creates the service-linked role for you.

> **Important**
>
> This service-linked role can appear in your account if you completed an action in another service that uses the features supported by this role. If you were using the EventBridge service before February 11, 2021, when it began supporting service-linked roles, then EventBridge created the **AWSServiceRoleForAmazonEventBridgeApiDestinations** role in your account. To learn more, see A new role appeared in my AWS account.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create a connection, EventBridge creates the service-linked role for you again.

Editing a service-linked role for EventBridge

EventBridge does not allow you to edit the **AWSServiceRoleForAmazonEventBridgeApiDestinations** service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for EventBridge

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don’t have an unused entity that is not actively monitored

or maintained. However, you must clean up your service-linked role before you can manually delete it.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first delete any resources used by the role.

> **Note**
>
> If the EventBridge service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To delete EventBridge resources used by the AWSServiceRoleForAmazonEventBridgeApiDestinations (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Under **Integrations** choose **API destinations**, then choose the **Connections** tab.
3. Choose the connection, then choose **Delete**.

To delete EventBridge resources used by the AWSServiceRoleForAmazonEventBridgeApiDestinations (AWS CLI)

- Use the following command: `delete-connection`.

To delete EventBridge resources used by the AWSServiceRoleForAmazonEventBridgeApiDestinations (API)

- Use the following command: `DeleteConnection`.

Manually delete the service-linked role

Use the IAM console, the AWS CLI, or the AWS API to delete the **AWSServiceRoleForAmazonEventBridgeApiDestinations** service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Supported Regions for EventBridge service-linked roles

EventBridge supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Using roles for schema discovery in Amazon EventBridge

The following topic details usage of the **AWSServiceRoleForSchemas** service-linked role.

Service-linked role permissions for EventBridge

EventBridge uses the service-linked role named **AWSServiceRoleForSchemas** – Grants permissions to Managed Rules created by Amazon EventBridge schemas..

The **AWSServiceRoleForSchemas** service-linked role trusts the following services to assume the role:

- `schemas.amazonaws.com`

The role permissions policy named **AmazonEventBridgeSchemasServiceRolePolicy** allows EventBridge to complete the following actions on the specified resources:

- Action: `put, enable, disable, and delete rules; put and remove targets; list targets per rule` on `all managed rules created by EventBridge`

You must configure permissions to allow your users, groups, or roles to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for EventBridge

You don't need to manually create a service-linked role. When you conduct a Schema Discovery in the AWS Management Console, the AWS CLI, or the AWS API, EventBridge creates the service-linked role for you.

> **Important**
>
> This service-linked role can appear in your account if you completed an action in another service that uses the features supported by this role. If you were using the EventBridge service before November 27, 2019, when it began supporting service-linked roles, then

EventBridge created the **AWSServiceRoleForSchemas** role in your account. To learn more, see A new role appeared in my AWS account.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you conduct a Schema Discovery, EventBridge creates the service-linked role for you again.

Editing a service-linked role for EventBridge

EventBridge does not allow you to edit the **AWSServiceRoleForSchemas** service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for EventBridge

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up your service-linked role before you can manually delete it.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first delete any resources used by the role.

Note

If the EventBridge service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To delete EventBridge resources used by the AWSServiceRoleForSchemas (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. Under **Buses** choose **Event buses**, then choose an event bus.
3. Choose **Stop discovery**.

To delete EventBridge resources used by the AWSServiceRoleForSchemas (AWS CLI)

- Use the following command: `delete-discoverer`.

To delete EventBridge resources used by the AWSServiceRoleForSchemas (API)

- Use the following command: `DeleteDiscoverer`.

Manually delete the service-linked role

Use the IAM console, the AWS CLI, or the AWS API to delete the **AWSServiceRoleForSchemas** service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Supported Regions for EventBridge service-linked roles

EventBridge supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Logging Amazon EventBridge API calls using AWS CloudTrail

Amazon EventBridge is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for EventBridge as events. The calls captured include calls from the EventBridge console and code calls to the EventBridge API operations. Using the information collected by CloudTrail, you can determine the request that was made to EventBridge, the IP address from which the request was made, when it was made, and additional details.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For

more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

EventBridge management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

All Amazon EventBridge API operations except PutEvents and PutPartnerEvents are control plane operations. For a complete list, see Actions in the *Amazon EventBridge API Reference*. For example management events, see the section called “Management event examples”.

CloudTrail records Custom Event Bus API calls with the event source `eventsv2.amazonaws.com`. `PutEvents` and `PutRawEvents` publish events to a bus. Every other Custom Event Bus API operation is a management event.

For a complete list of Custom Event Bus API operations, see Actions in the *Custom Event Bus API Reference*.

EventBridge data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, publishing events to an event bus). These are also known as data plane operations.

Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the EventBridge resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

The following table lists the EventBridge resource types for which you can log data events. The **Resource type (console)** column shows the value to choose from the **Resource type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Resource type (console)	resources.type value	Data APIs logged to CloudTrail
EventBridge event bus	`AWS::Events::EventBus`	• PutEvents
EventBridge partner event source	`AWS::Events::EventSource`	• PutPartnerEvents
EventBridge endpoint	`AWS::Events::Endpoint`	• PutEvents (when using a global endpoint; successful calls only, not populated on `AccessDenied`)

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.`ARN fields to log only those events that are important to you. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*.

Note

The `detail` field in event entries is redacted in CloudTrail data event logs to protect sensitive data.

Data event delivery

Note

CloudTrail sends data events only to the AWS account that called the API.

`PutEvents`

Cross-account `PutEvents` calls

When an account that you granted permission to via a resource policy calls `PutEvents` to your bus, only the caller's account receives the CloudTrail data event. Your account (the bus owner) does not.

Bus-to-bus forwarding

Up to four accounts can be involved in a bus-to-bus forwarding scenario. The following table shows which accounts receive a CloudTrail data event.

Account role	Receives CloudTrail data event
The account that originally called `PutEvents` (the sender)	Yes, always.
The account that owns the source bus (the source bus owner)	No.
The account that owns the rule and the IAM role on the rule target (the rule owner)	Yes, but only for cross-Region forwarding. For cross-Region forwarding, EventBridge uses the IAM role on the rule target to call

Account role	Receives CloudTrail data event
	`PutEvents` on the destination Region. This makes the rule owner's account the effective API caller. Same-Region forwarding does not generate an additional data event.
The account that owns the destination bus (the destination bus owner)	No.

> **Note**
>
> For cross-Region forwarding, the original `PutEvents` call and the forwarding `PutEvents` call are recorded as separate CloudTrail data events in their respective Regions. To capture both, ensure that CloudTrail is configured to log data events in both Regions.

Global endpoints with event replication

When you use a global endpoint with event replication enabled, EventBridge creates a managed rule that forwards events cross-Region to the secondary bus. This is a cross-Region bus-to-bus forwarding scenario, so the same rules from the table above apply:

- Primary Region: EventBridge generates a data event for the original `PutEvents` call. The event includes both the event bus ARN and the endpoint resource ARN.
- Secondary Region: Because you own the managed rule and the IAM role used to replicate events, your account receives a data event for the replication `PutEvents` call. The event includes only the event bus ARN. The endpoint resource ARN is not included.

`PutPartnerEvents`

Events received from a SaaS partner

When a SaaS partner sends events to your bus, your account does not receive a `PutPartnerEvents` CloudTrail data event. Only the SaaS partner's account receives the data event.

Events sent to a customer bus as a SaaS partner

When you send events to a customer's event bus as a SaaS partner, your account receives the `PutPartnerEvents` CloudTrail data event. The event includes only the event source ARN as a resource. The customer's event bus ARN is not included.

For examples of data events that EventBridge logs to CloudTrail, see the section called “Data event examples”.

EventBridge management event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows a CloudTrail event that demonstrates the `PutRule` operation.


```
{
  "eventVersion":"1.03",
  "userIdentity":{
    "type":"Root",
    "principalId":"123456789012",
    "arn":"arn:aws:iam::123456789012:root",
    "accountId":"123456789012",
    "accessKeyId":"AKIAIOSFODNN7EXAMPLE",
    "sessionContext":{
      "attributes":{
      "mfaAuthenticated":"false",
      "creationDate":"2015-11-17T23:56:15Z"
      }
    }
  },
  "eventTime":"2015-11-18T00:11:28Z",
  "eventSource":"events.amazonaws.com",
  "eventName":"PutRule",
  "awsRegion":"us-east-1",
  "sourceIPAddress":"AWS Internal",
  "userAgent":"AWS CloudWatch Console",
  "requestParameters":{
    "description":"",
    "name":"cttest2",
    "state":"ENABLED",
```

```
    "eventPattern":"{\"source\":[\"aws.ec2\"],\"detail-type\":[\"EC2 Instance State-
change Notification\"]}",
    "scheduleExpression":""
  },
  "responseElements":{
    "ruleArn":"arn:aws:events:us-east-1:123456789012:rule/cttest2"
  },
  "requestID":"e9caf887-8d88-11e5-a331-3332aa445952",
  "eventID":"49d14f36-6450-44a5-a501-b0fdcdfaeb98",
  "eventType":"AwsApiCall",
  "apiVersion":"2015-10-07",
  "recipientAccountId":"123456789012"
}
```

In a successful Custom Event Bus `PutResourcePolicy` call, the `requestParameters` field holds the policy document as the caller submitted it. The following example shows a CloudTrail event for such a call:

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROA3XFRBF23EXAMPLE:johndoe-session",
    "arn": "arn:aws:sts::111122223333:assumed-role/MyRole/johndoe-session",
    "accountId": "111122223333",
    "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROA3XFRBF23EXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/MyRole",
        "accountId": "111122223333",
        "userName": "MyRole"
      },
      "attributes": {
        "creationDate": "2026-09-28T16:13:38Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2026-09-28T16:13:43Z",
  "eventSource": "eventsv2.amazonaws.com",
  "eventName": "PutResourcePolicy",
```

```
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.0.2.1",
  "userAgent": "aws-cli/2.37.3",
  "requestParameters": {
    "resourceArn": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
    "policyDocument": "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Sid\":
\"AllowPartnerAccountToSubscribe\",\"Effect\":\"Allow\",\"Principal\":{\"AWS
\":\"arn:aws:iam::444455556666:root\"},\"Action\":[\"events:CreateSubscriber
\",\"events:DescribeEventBus\"],\"Resource\":\"arn:aws:events:us-
east-1:111122223333:event-busv2/orders/EXAMPLE1234567890abcdef\"}]}",
    "policyName": "default"
  },
  "responseElements": {
    "resourceArn": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef",
    "policyName": "default",
    "revisionId": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa"
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEccccc",
  "readOnly": false,
  "resources": [
    {
      "accountId": "111122223333",
      "type": "AWS::EventsV2::EventBus",
      "ARN": "arn:aws:events:us-east-1:111122223333:event-busv2/orders/
EXAMPLE1234567890abcdef"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "111122223333",
  "eventCategory": "Management",
  "tlsDetails": {
    "tlsVersion": "TLSv1.3",
    "cipherSuite": "TLS_AES_128_GCM_SHA256",
    "clientProvidedHostHeader": "111122223333.eventsv2.us-east-1.amazonaws.com"
  }
}
```


For information about CloudTrail record contents, see CloudTrail record contents in the *AWS CloudTrail User Guide*.

EventBridge data event examples

The following example shows a CloudTrail data event for a successful `PutEvents` call.


```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROA3XFRBF23EXAMPLE:johndoe-session",
    "arn": "arn:aws:sts::111122223333:assumed-role/MyRole/johndoe-session",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROA3XFRBF23EXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/MyRole",
        "accountId": "111122223333",
        "userName": "MyRole"
      },
      "attributes": {
        "creationDate": "2026-03-07T00:49:09Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2026-03-07T00:51:07Z",
  "eventSource": "events.amazonaws.com",
  "eventName": "PutEvents",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.0.2.1",
  "userAgent": "aws-cli/2.33.13",
  "requestParameters": {
    "entries": [
      {
        "source": "my-application",
        "detailType": "MyDetailType",
        "detail": "HIDDEN_DUE_TO_SECURITY_REASONS",
        "eventBusName": "default"
      }
    ]
  },
  "responseElements": {
```

```
    "failedEntryCount": 0,
    "entries": [
      {
        "eventId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
      }
    ]
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
  "readOnly": false,
  "resources": [
    {
      "type": "AWS::Events::EventBus",
      "ARN": "arn:aws:events:us-east-1:111122223333:event-bus/default"
    }
  ],
  "eventType": "AwsApiCall",
  "apiVersion": "2015-10-07",
  "managementEvent": false,
  "recipientAccountId": "111122223333",
  "eventCategory": "Data",
  "tlsDetails": {
    "tlsVersion": "TLSv1.3",
    "cipherSuite": "TLS_AES_128_GCM_SHA256",
    "clientProvidedHostHeader": "events.us-east-1.amazonaws.com"
  }
}
```

The following example shows a CloudTrail data event for a `PutEvents` call using a global endpoint. Note the additional `AWS::Events::Endpoint` resource in the `resources` array.

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROA3XFRBF23EXAMPLE:johndoe-session",
    "arn": "arn:aws:sts::111122223333:assumed-role/MyRole/johndoe-session",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROA3XFRBF23EXAMPLE",
```

```
        "arn": "arn:aws:iam::111122223333:role/MyRole",
        "accountId": "111122223333",
        "userName": "MyRole"
      },
      "attributes": {
        "creationDate": "2026-03-07T00:49:09Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2026-03-07T00:51:06Z",
  "eventSource": "events.amazonaws.com",
  "eventName": "PutEvents",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.0.2.1",
  "userAgent": "aws-cli/2.33.13",
  "requestParameters": {
    "entries": [
      {
        "source": "my-application",
        "detailType": "MyDetailType",
        "detail": "HIDDEN_DUE_TO_SECURITY_REASONS",
        "eventBusName": "default"
      }
    ],
    "endpointId": "abc1234567.us-east-1"
  },
  "responseElements": {
    "failedEntryCount": 0,
    "entries": [
      {
        "eventId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE44444"
      }
    ]
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE55555",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE66666",
  "readOnly": false,
  "resources": [
    {
      "type": "AWS::Events::EventBus",
      "ARN": "arn:aws:events:us-east-1:111122223333:event-bus/default"
    },
    {
```

```
      "type": "AWS::Events::Endpoint",
      "ARN": "arn:aws:events:us-east-1:111122223333:endpoint/MyGlobalEndpoint"
    }
  ],
  "eventType": "AwsApiCall",
  "apiVersion": "2015-10-07",
  "managementEvent": false,
  "recipientAccountId": "111122223333",
  "eventCategory": "Data",
  "tlsDetails": {
    "tlsVersion": "TLSv1.3",
    "cipherSuite": "TLS_AES_128_GCM_SHA256",
    "clientProvidedHostHeader": "abc1234567.us-east-1.endpoint.events.amazonaws.com"
  }
}
```

The following example shows a CloudTrail data event for a successful `PutPartnerEvents` call.

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROA3XFRBF23EXAMPLE:partner-session",
    "arn": "arn:aws:sts::111122223333:assumed-role/PartnerRole/partner-session",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROA3XFRBF23EXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/PartnerRole",
        "accountId": "111122223333",
        "userName": "PartnerRole"
      },
      "attributes": {
        "creationDate": "2026-03-07T00:49:09Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2026-03-07T00:51:07Z",
  "eventSource": "events.amazonaws.com",
  "eventName": "PutPartnerEvents",
```

```
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.0.2.1",
  "userAgent": "aws-cli/2.33.13",
  "requestParameters": {
    "entries": [
      {
        "time": "Mar 7, 2026, 12:51:07 AM",
        "source": "aws.partner/example.com/my-integration",
        "detailType": "MyDetailType",
        "detail": "HIDDEN_DUE_TO_SECURITY_REASONS"
      }
    ]
  },
  "responseElements": {
    "failedEntryCount": 0,
    "entries": [
      {
        "eventId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE77777"
      }
    ]
  },
  "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE88888",
  "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE99999",
  "readOnly": false,
  "resources": [
    {
      "type": "AWS::Events::EventSource",
      "ARN": "arn:aws:events:us-east-1::event-source/aws.partner/example.com/my-
integration"
    }
  ],
  "eventType": "AwsApiCall",
  "apiVersion": "2015-10-07",
  "managementEvent": false,
  "recipientAccountId": "111122223333",
  "eventCategory": "Data",
  "tlsDetails": {
    "tlsVersion": "TLSv1.3",
    "cipherSuite": "TLS_AES_128_GCM_SHA256",
    "clientProvidedHostHeader": "events.us-east-1.amazonaws.com"
  }
}
```

CloudTrail log entries for actions taken by EventBridge Pipes

EventBridge Pipes assumes the provided IAM role when reading events from sources, invoking enrichments, or invoking targets. For CloudTrail entries related to actions taken in your account on all enrichments, targets, and Amazon SQS, Kinesis, and DynamoDB sources, the `sourceIPAddress` and `invokedBy` fields will include `pipes.amazonaws.com`.

Sample CloudTrail log entry for all enrichments, targets, and Amazon SQS, Kinesis, and DynamoDB sources


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "...",
    "arn": "arn:aws:sts::111222333444:assumed-role/...",
    "accountId": "111222333444",
    "accessKeyId": "...",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "...",
        "arn": "...",
        "accountId": "111222333444",
        "userName": "userName"
      },
      "webIdFederationData": {},
      "attributes": {
        "creationDate": "2022-09-22T21:41:15Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "pipes.amazonaws.com"
  },
  "eventTime": ",,,",
  "eventName": "...",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "pipes.amazonaws.com",
  "userAgent": "pipes.amazonaws.com",
  "requestParameters": {
    ...
  },
  "responseElements": null,
```

```
  "requestID": "...",
  "eventID": "...",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "...",
  "eventCategory": "Management"
}
```

For all other sources, the `sourceIPAddress` field of the CloudTrail log entries will have a dynamic IP address and shouldn't be relied upon for any integration or event categorization. In addition, these entries won't have the `invokedBy` field.

Sample CloudTrail log entry for all other sources

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    ...
  },
  "eventTime": ",,,",
  "eventName": "...",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "127.0.0.1",
  "userAgent": "Python-httplib2/0.8 (gzip)",
}
```

Compliance validation in Amazon EventBridge

Third-party auditors such as SOC, PCI, FedRAMP, and HIPAA assess the security and compliance of AWS services as part of multiple AWS compliance programs.

For a list of AWS services in scope of specific compliance programs, see . For general information, see .

Your compliance responsibility when using EventBridge is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- – A collection of workbooks and guides.
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – Information about how AWS Config assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.

Amazon EventBridge resilience

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Infrastructure security in Amazon EventBridge

As a managed service, Amazon EventBridge is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access EventBridge through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

You can call these API operations from any network location, and you can use resource-based access policies in EventBridge, which can include restrictions based on the source IP address. You can also use EventBridge policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given EventBridge resource from only the specific VPC within the AWS network.

Configuration and vulnerability analysis in Amazon EventBridge

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model.

Dual-stack service endpoints in Amazon EventBridge

In addition to the IPv4 service endpoints, EventBridge also provides dual-stack endpoints for the following resources. You can access these endpoints using either IPv4 or IPv6 requests.

Resource	Region endpoint	FIPS endpoint	Regions
Custom Event Bus	HTTPS://eventsv2. *region-code* .api.aws	HTTPS://eventsv2-fips. *region-code* .api.aws	See the section called "Names, endpoints, and permissions"
Custom Event Bus - Classic	HTTPS://events. *region-code* .api.aws	HTTPS://events-fips. *region-code* .api.aws	EventBridge
Pipes	HTTPS://pipes. *region-code* .api.aws	HTTPS::://pipes-fips. *region-code* .api.aws	EventBridge Pipes
Scheduler	HTTPS://scheduler. *region-code* .api.aws	HTTPS://scheduler-fips. *region-code* .api.aws	EventBridge Scheduler
Schemas	HTTPS://schemas. *region-code* .api.aws	HTTPS://schemas-fips. *region-code* .api.aws	EventBridge Schemas

To make a request to a dual-stack endpoint, you must use the mechanism provided by the tool or AWS SDK to specify the endpoint. For more information, see Dual-stack and FIPS endpoints in the *AWS SDKs and Tools Reference Guide*.

For example, the AWS CLI provides the `--endpoint-url` option. The following example uses `--endpoint-url` to specify the dual-stack endpoint for event buses in the US West (Oregon) Region.

```
aws events describe-event-bus --region us-west-2 --endpoint-url https://events.us-west-2.api.aws
```

For a list of Region codes to use when specifying regional endpoints, see Regional endpoints in the *AWS General Reference*.

Considerations when using dual-stack endpoints

Keep the following considerations in mind when using dual-stack endpoints to access EventBridge resources.

For event buses:

- Dual-stack support for event buses includes archives and replays.
- EventBridge does not currently support dual-stack endpoints for global endpoints.

  If you configure the AWS SDK to use dual-stack endpoints, calls to `PutEvents` will result in an error. This is because the SDK uses the dual-stack domain (`events.`*`region-code`*`.api.aws`) rather than `events.`*`region-code`*`.amazonaws.com` when constructing the endpoint, resulting in the request being sent to an endpoint that does not exist.

For EventBridge Pipes:

- EventBridge Pipes supports dual-stack requests with Interface VPC endpoints.

Tagging resources in Amazon EventBridge

A *tag* is a custom attribute label that you or AWS assigns to an AWS resource. In EventBridge, you can assign tags to rule and event buses. Each resource can have a maximum of 50 tags.

You use tags to identify and organize your AWS resources. Many AWS services support tagging, so you can assign the same tag to resources from different services to indicate that the resources are related. For example, you could assign the same tag to an EventBridge rule that you assign to an EC2 instance.

A tag has two parts:

- A *tag key*, for example, `CostCenter`, `Environment`, or `Project`.
  - Tag keys are case sensitive.
  - The maximum tag key length is 128 Unicode characters in UTF-8.
  - For each resource, each tag key must be unique.
  - Allowed characters are letters, numbers, spaces representable in UTF-8, and the following characters: ***. : + = @ _ / - (hyphen)***.
  - The `aws:` prefix is prohibited for tags because it's reserved for AWS use. You can't edit or delete tag keys or values with this prefix. Tags with this prefix don't count against your tags per resource limit.
- An optional *tag value* field, for example, `111122223333` or `Production`.
  - Each tag key can have only one value.
  - Tag values are case sensitive.
  - Omitting the tag value is the same as using an empty string.
  - The maximum tag value length is 256 Unicode characters in UTF-8.
  - Allowed characters are letters, numbers, spaces representable in UTF-8, and the following characters: ***. : + = @ _ / - (hyphen)***.

> **Tip**
>
> As a best practice, decide on a strategy for capitalizing tags and consistently implement that strategy across all resource types. For example, decide whether to use `Costcenter`, `costcenter`, or `CostCenter` and then use the same convention for all tags.

For more information on managing tags, see Working with Tag Editor in the *Resource Groups User Guide*.

Adding or removing tags on event buses

You can add or remove tags on event buses.

To add or remove tags on an event bus (console)

1. Open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, choose **Event buses**.
3. Choose the event bus you want to update.
4. On the events bus details page, choose the **Tags** tab, and then choose **Manage tags**.
5. Do one of the following:
   - To add a tag:
     a. Choose **Add new tag**.
     b. Specify the key and value for the tag
     c. Choose **Update**.
   - To remove a tag:
     a. For the tag you want to remove, choose **Remove**.
     b. Choose **Update**.

To add or remove tags from an event bus (AWS CLI)

- To add tags, use tag-resource.

  To remove tags, use untag-resource.

  You can also determine the tags on an event bus by using list-tags-for-resource.

Amazon EventBridge quotas

The following quotas apply to the Custom Event Bus, to Custom Event Bus - Classic, to the API destinations and connections that both buses deliver to, to the schema registry, and to EventBridge Pipes.

> **Note**
>
> If you use Custom Event Bus - Classic, the event bus that routes with rules and targets, see the section called "Custom Event Bus - Classic quotas".

Topics

- Custom Event Bus quotas
- Custom Event Bus - Classic quotas
- API destinations and connections
- EventBridge Schema Registry quotas
- EventBridge Pipes quotas

> **Note**
>
> For a list of the quotas for EventBridge Scheduler, see Quotas for EventBridge Scheduler in the *EventBridge Scheduler User Guide.*

Custom Event Bus quotas

The Service Quotas console lists 6 quotas for the Custom Event Bus under Amazon EventBridge, each with an `[EventsV2]` prefix; 4 of them are adjustable. To raise the bus limit, for example, request an increase to `[EventsV2] Event Buses`. The other limits in this section are fixed values that the API enforces and are not adjustable. These quotas are separate from the Custom Event Bus - Classic quotas in the section called "Custom Event Bus - Classic quotas".

Service Quotas

The following table lists the quotas in the Service Quotas console, per account per Region.

Quota	Default	Adjustable
Event buses owned by the account. Buses shared with the account do not count	5	Yes
Event sources owned by the account	200	Yes
Subscribers per bus the account owns, counting every account the bus is shared with	10,000	Yes
Size, in bytes, of the `default` resource policy of a bus the account owns. The `AWS_RAM` policy that AWS RAM writes for a shared bus does not count	20,480	Yes
Events per second per bus, counting every event source and every account the bus is shared with	500,000	No
Ingestion per event group per second, counted as the number of events plus their total size in KB in any one-second window	1,500	No

Your account's bus count is also published as the `ResourceCount` metric in the `AWS/Usage` namespace, so you can alarm before you reach the limit; see the section called “Observability”. Publishing and subscriber management are throttled separately for each account: `PutEvents` and `PutRawEvents` draw from one budget, and `CreateSubscriber` and `DeleteSubscriber` from another. Handle `ThrottlingException` with backoff and retry.

If the `default` policy in a `PutResourcePolicy` call exceeds the resource policy size quota, the call fails with `PolicyLengthExceededException`. See the section called “Write a custom resource policy”.

Fixed limits

The following values are set by the API and cannot be raised. Where a setting takes a range, you choose any value in the range; the maximum is the fixed part.

Setting	You choose	Fixed maximum or value
`StorageConfiguration.RetentionPeriodInDays`	1 to 365 days	365 days
`RetryPolicy.MaxRetryAttempts`	0 to 185, default 5	185
`RetryPolicy.MaxEventAgeInSeconds`	60 to 86,400, default 300	86,400 seconds
`BatchConfiguration.MaxBatchSize`	1 to 500	500
`BatchConfiguration.MaxBatchWindowInSeconds`	0 to 300, default 0	300 seconds
`InvocationTimeoutSeconds` on Lambda, Step Functions, and HTTP targets	1 to 30, default 30	30 seconds
Entries per `PutEvents` or `PutRawEvents` request	1 to 100	100
Deduplication window	Not configurable	300 seconds

Setting	You choose	Fixed maximum or value
JSONata expression in a `Transformer` or a target parameter	Not configurable	8,192 characters
JSONata expression in a universal target's `Input`	Not configurable	262,144 characters
Filters in one `FilterConfiguration`	Up to 3, one per scope	4,096 bytes across all filters; 1,000 `$or` combinations
Filter changes per subscriber	Not configurable	24 in any rolling 24 hours; see the section called "Updating a subscriber"
Subscriber creates or deletes in progress on one bus	Not configurable	1 across all accounts; a second call fails with `ConcurrentModificationException`
Subscribers replaying from a past position at the same time	Not configurable	Limited per bus; `CreateSubscriber` fails with `LimitExceededException` when reached
`errorMessage` in a dead-letter record	Not configurable	1,024 characters

Custom Event Bus - Classic quotas

Custom Event Bus - Classic has the following quotas. The Service Quotas table also lists the API destination, connection, and global endpoint quotas, which are account-level resources.

The Service Quotas console provides information about EventBridge quotas. Along with viewing the default quotas, you can use the Service Quotas console to request quota increases for adjustable quotas.

Name	Default	Adjustable	Description
Api destinations	Each supported Region: 3,000	Yes	The maximum number of API destinations per account per Region.
Connections	Each supported Region: 3,000	Yes	The maximum number of connections per account per Region.
CreateEndpoint throttle limit in transactions per second	Each supported Region: 5 per second	No	The maximum number of requests per second for CreateEndpoint API. Additional requests are throttled.
DeleteEndpoint throttle limit in transactions per second	Each supported Region: 5 per second	No	The maximum number of requests per second for DeleteEndpoint API. Additional requests are throttled.
Endpoints	Each supported Region: 100	Yes	The maximum number of endpoints per account per Region.
Event bus policy size	Each supported Region: 10,240	Yes	Maximum policy size, in characters. This policy size increases each time you grant access to another account. You can see your current policy and its size by using the DescribeEventBus API.
Event buses	Each supported Region: 100	Yes	Maximum event buses per account.

Name	Default	Adjustable	Description
Event pattern size	Each supported Region: 2,048	Yes	Maximum size of an event pattern, in characters.

Name	Default	Adjustable	Description
Invocations throttle limit in transactions per second	us-east-1: 18,750 per second us-east-2: 4,500 per second us-west-1: 2,250 per second us-west-2: 18,750 per second ap-east-1: 1,100 per second ap-northeast-1: 2,250 per second ap-northeast-2: 1,100 per second ap-south-1: 1,100 per second ap-southeast-1: 2,250 per second ap-southeast-2: 2,250 per second ca-central-1: 1,100 per second eu-central-1: 4,500 per second	Yes	An invocation is an event matching a rule and being sent on to the rules targets. After the limit is reached, the invocations are throttled; that is, they still happen but they are delayed.

Name	Default	Adjustable	Description
	eu-north-1: 1,100 per second eu-south-2: 4,500 per second eu-west-1: 18,750 per second eu-west-2: 2,250 per second eu-west-3: 1,100 per second sa-east-1: 1,100 per second Each of the other supported Regions: 750 per second		
Number of rules	af-south-1: 100 eu-south-1: 100 Each of the other supported Regions: 300	Yes	Maximum number of rules an account can have per event bus

Name	Default	Adjustable	Description
PutEvents throttle limit in transactions per second	us-east-1: 10,000 per second us-east-2: 2,400 per second us-west-1: 1,200 per second us-west-2: 10,000 per second ap-east-1: 600 per second ap-northeast-1: 1,200 per second ap-northeast-2: 600 per second ap-south-1: 600 per second ap-southeast-1: 1,200 per second ap-southeast-2: 1,200 per second ca-central-1: 600 per second eu-central-1: 2,400 per second	Yes	Maximum number of requests per second for PutEvents API. Additional requests are throttled.

Name	Default	Adjustable	Description
	eu-north-1: 600 per second eu-south-2: 2,400 per second eu-west-1: 10,000 per second eu-west-2: 1,200 per second eu-west-3: 600 per second sa-east-1: 600 per second Each of the other supported Regions: 400 per second		
Rate of invocations per API destination	Each supported Region: 300 per second	Yes	The maximum number of invocations per second to send to each API destination endpoint per account per Region. Once the quota is met, future invocations to that API endpoint are throttled. The invocations will still occur, but are delayed.

Name	Default	Adjustable	Description
Targets per rule	Each supported Region: 5	No	Maximum number of targets that can be associated with a rule
Throttle limit in transactions per second for control plane APIs	Each supported Region: 50 per second	No	Maximum number of requests per second for EventBridge control plane API operations. Additional requests are throttled.
UpdateEndpoint throttle limit in transactions per second	Each supported Region: 5 per second	No	The maximum number of requests per second for UpdateEndpoint API. Additional requests are throttled.
[EventsV2] Combined ingestion throughput and count per Event Group per second	Each supported Region: 1,500 per second	No	The maximum event throughput per Event Group per Second. Counted as the sum of (number of events) + (total size of those events in KB) delivered to a given event group within any one-second window.
[EventsV2] Event Buses	Each supported Region: 5	Yes	The maximum number of V2 Event Buses owned by this account per Region. Does not include buses shared with this account.

Name	Default	Adjustable	Description
[EventsV2] Event Sources	Each supported Region: 200	Yes	The maximum number of Event Sources owned by this account per Region.
[EventsV2] Events per second per custom event bus	Each supported Region: 500,000 per second	No	The maximum number of events per second ingested per custom event bus, counted across all event sources and all accounts the bus is shared with.
[EventsV2] Resource policy size per event bus	Each supported Region: 20,480 Bytes	Yes	The maximum size, in bytes, of the customer-managed (default) resource policy of a V2 event bus. Policies that AWS Resource Access Manager writes for a shared bus do not count against this quota.
[EventsV2] Subscribers per V2 Event Bus	Each supported Region: 10,000	Yes	The maximum number of Subscribers that can be associated with each V2 Event Bus owned by this account, across all accounts it is shared with.

In addition, EventBridge has the following quotas that are not managed through the Service Quotas console.

Name	Default	Description
Rules containing wildcards	Each supported Region: 30 rules per event bus	Maximum number of rules, per event bus per account, that can contain event filters that include wildcards. This quota cannot be adjusted. For more information on using wildcards in event patterns, see ???.
Schema discovery levels	Each supported Region: 255 levels	Maximum number of levels schema discovery will infer events that are nested. Any events past 255 levels are ignored.
Connections (Private)	Each supported Region: 20	The maximum number of private connections per account per Region.

API destinations and connections

API destinations and connections belong to your account, not to a bus, and both buses use them: a Custom Event Bus - Classic rule targets an API destination, and a Custom Event Bus subscriber delivers to one through `HttpParameters`. A Custom Event Bus producer that publishes Avro or Protobuf with a Confluent schema registry also names a connection. One set of quotas covers all of that use. In the Service Quotas table in the section called "Custom Event Bus - Classic quotas", the rows *Api destinations*, *Connections*, and *Rate of invocations per API destination* are the account-wide values; the invocation rate is shared by every rule and subscriber that targets the same API destination. The fixed table in the same section lists the private connection limit.

EventBridge Schema Registry quotas

EventBridge Schema Registry has the following quotas.

The Service Quotas console provides information about EventBridge quotas. Along with viewing the default quotas, you can use the Service Quotas console to request quota increases for adjustable quotas.

Name	Default	Adjuse	Description
DiscoveredSchemas	Each supported Region: 200	Yes	The maximum number of schemas for a discovered schema registry that you can create in the current region
Discoverers	Each supported Region: 10	Yes	The maximum number of discoverers that you can create in the current region.
Registries	Each supported Region: 10	Yes	The maximum number of registries that you can create in the current region.
SchemaVersions	Each supported Region: 100	Yes	The maximum number of versions per schema that you can create in the current region.
Schemas	Each supported Region: 100	Yes	The maximum number of schemas per registry that you can create in the current region. (Except Discovered Schema Registry)

EventBridge Pipes quotas

EventBridge Pipes has the following quotas. If you have requirements for higher maximum limits, contact support.

Resource	Regions	Default limit
Concurrent pipe executions per account	• AWS GovCloud (US-West) • AWS GovCloud (US-East) • China (Ningxia) • China (Beijing) • Asia Pacific (Osaka) • Africa (Cape Town) • Europe (Milan) • US East (Ohio) • Europe (Frankfurt) • US West (N. California) • Europe (London) • Asia Pacific (Sydney) • Asia Pacific (Tokyo) • Asia Pacific (Singapore) • Canada (Central) • Europe (Paris) • Europe (Stockholm) • South America (São Paulo) • Asia Pacific (Seoul) • Asia Pacific (Mumbai) • Asia Pacific (Hong Kong) • Middle East (Bahrain) • China (Ningxia) • China (Beijing) • Asia Pacific (Osaka) • Africa (Cape Town) • Europe (Milan)	1000

Resource	Regions	Default limit
Concurrent pipe executions per account	• US East (N. Virginia) • US West (Oregon) • Europe (Ireland)	3000
Pipes per account	All	1000

Amazon EventBridge feature availability by AWS Region

The following table shows the availability of Amazon EventBridge features by AWS Region. A check mark (✓) indicates that the feature is available in the corresponding Region; a blank cell indicates that it is not. For API destinations, the cell shows which variants are available (Private API destinations require Public API destinations). Features shown include Event buses, Scheduler, Pipes, Archive and replay, the Schema registry and Schema discovery, and Global endpoints.

Amazon EventBridge feature availability by AWS Region

Region	Event buses	Schedule	Pipes	Archive and replay	Schema registry	Schema discovery	Global endpoints	API destinations
US East (N. Virginia)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
US East (Ohio)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
US West (N. California)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
US West (Oregon)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Africa (Cape Town)	✓	✓	✓	✓	✓	✓		Public ✓
Asia Pacific (Hong Kong)	✓	✓	✓	✓	✓	✓		Public ✓ Private ✓
Asia Pacific (Taipei)	✓	✓						

Region	Event buses	Schedule	Pipes	Archive and replay	Schema registry	Schema discovery	Global endpoints	API destinations
Asia Pacific (Tokyo)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Asia Pacific (Seoul)	✓	✓	✓	✓	✓	✓	✓	Public ✓
Asia Pacific (Osaka)	✓	✓	✓	✓	✓	✓	✓	Public ✓
Asia Pacific (Mumbai)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Asia Pacific (Hyderabad)	✓	✓	✓	✓	✓			Public ✓
Asia Pacific (Singapore)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Asia Pacific (Sydney)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Asia Pacific (Jakarta)	✓	✓	✓	✓	✓	✓		Public ✓
Asia Pacific (Melbourne)	✓	✓			✓			Public ✓
Asia Pacific (Malaysia)	✓	✓						
Asia Pacific (Thailand)	✓	✓						Public ✓

Region	Event buses	Schedule	Pipes	Archive and replay	Schema registry	Schema discovery	Global endpoints	API destinations
Canada (Central)	✓	✓	✓	✓	✓	✓	✓	Public ✓
Canada West (Calgary)	✓	✓			✓			
China (Beijing)	✓	✓	✓	✓	✓	✓		Public ✓
China (Ningxia)	✓	✓	✓	✓	✓	✓		Public ✓
Europe (Frankfurt)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Europe (Zurich)	✓	✓	✓	✓	✓	✓		Public ✓
Europe (Stockholm)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Europe (Milan)	✓	✓	✓	✓	✓	✓		Public ✓
Europe (Spain)	✓	✓	✓	✓	✓	✓		Public ✓
Europe (Ireland)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Europe (London)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Europe (Paris)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
Israel (Tel Aviv)	✓	✓			✓			
Middle East (UAE)	✓	✓	✓	✓	✓	✓		Public ✓

Region	Event buses	Schedule	Pipes	Archive and replay	Schema registry	Schema discovery	Global endpoints	API destinations
Middle East (Bahrain)	✓	✓	✓	✓	✓	✓		Public ✓
Mexico (Central)	✓	✓						
South America (São Paulo)	✓	✓	✓	✓	✓	✓	✓	Public ✓ Private ✓
AWS GovCloud (US-East)	✓	✓		✓	✓	✓		
AWS GovCloud (US-West)	✓	✓		✓	✓	✓		

Document History

The following table describes important changes in each release of the *Amazon EventBridge User Guide*, beginning in July 2019. For notification about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Release Date
Updated the AmazonEventBridgeReadOnlyAccess policy	You can now view Custom Event Bus subscribers and event bus resource policies, and list the tags on EventBridge resources. • the section called “Policy updates”	September 24, 2026
Updated AmazonEventBridgeApiDestinationsServiceRolePolicy managed policy	Policy now restricts the scope of permissions for Secrets Manager operations to the same account. • the section called “Policy updates”	May 29, 2025
Updated AmazonEventBridgeApiDestinationsServiceRolePolicy managed policy	Policy now grants EventBridge AWS KMS encrypt and decrypt permissions via Secrets Manager. This enables EventBridge to update connection secret resources with new OAuth token value when access token refresh is required. • the section called “Policy updates”	March 28, 2025
Document structure update	Used page view data and depth analysis to restructure documentation sections to increase visibility of important topics. Updated guide navigation to reduce overall depth. Consolidated related topics as appropriate.	August 4, 2024

Change	Description	Release Date
Updated AWS managed policies.	AWS GovCloud (US) Regions only `AmazonEventBridgeFullAccess` and `AmazonEventBridgeSchemasFullAccess` policies do not include `iam:CreateServiceLinkedRole`, as it is not used. • the section called "Policy updates"	May 9, 2024
Generate CloudFormation templates from event buses and rules.	You can now generate CloudFormation templates from your existing Amazon EventBridge event buses and rules. • Generating an CloudFormation template from an existing EventBridge event bus	November 18, 2022
Launched EventBridge Pipes documentation.	You can now create pipes to connect sources to targets, with optional filtering and enrichment. • *Pipes*	December 1, 2022
Generate CloudFormation templates from event buses and rules.	You can now generate CloudFormation templates from your existing Amazon EventBridge event buses and rules. • Generating an CloudFormation template from an existing EventBridge event bus	November 18, 2022
Added the AmazonEventBridgePipesFullAccess policy.	Provides full access to Amazon EventBridge Pipes. • AWS managed policies: EventBridge Pipes	December 1, 2022

Change	Description	Release Date
Added the AmazonEventBridgePipesReadOnlyAccess policy.	Provides read-only access to Amazon EventBridge Pipes. • AWS managed policies: EventBridge Pipes	December 1, 2022
Added the AmazonEventBridgePipesOperatorAccess policy.	Provides read-only and operator (that is, the ability to stop and start running Pipes) access to Amazon EventBridge Pipes. • AWS managed policies: EventBridge Pipes	December 1, 2022
Updated the CloudWatchEventsFullAccess policy.	Updated to match `AmazonEventBridgeFullAccess` . • AWS managed policy: AmazonEventBridgeFullAccess	December 1, 2022
Updated the CloudWatchEventsReadOnlyAccess policy.	Updated to match `AmazonEventBridgeReadOnlyAccess` . • AWS managed policy: AmazonEventBridgeReadOnlyAccess	December 1, 2022
Updated content filtering in event patterns.	You can now use `suffix`, `equals-ignore-case` , and `$or` filtering options to create event patterns. • Using comparison operators in Amazon EventBridge event patterns	November 14, 2022
Updated the AmazonEventBridgeFullAccess policy.	Added permissions necessary for using EventBridge Schema Registry and EventBridge Scheduler. • AWS managed policy: AmazonEventBridgeFullAccess	November 10, 2022

Change	Description	Release Date
Updated the AmazonEventBridgeReadOnlyAccess policy.	You can now view EventBridge Schema Registry and EventBridge Scheduler information. • AWS managed policy: AmazonEventBridgeReadOnlyAccess	November 10, 2022
Updated content filtering in event patterns.	You can now use `suffix`, `equals-ignore-case` , and `$or` filtering options to create event patterns. • Using comparison operators in Amazon EventBridge event patterns	November 14, 2022
Updated the AmazonEventBridgeFullAccess policy.	Added permissions necessary for using EventBridge Schema Registry and EventBridge Scheduler. • AWS managed policy: AmazonEventBridgeFullAccess	November 10, 2022
Updated the AmazonEventBridgeReadOnlyAccess policy.	You can now view EventBridge Schema Registry and EventBridge Scheduler information. • AWS managed policy: AmazonEventBridgeReadOnlyAccess	November 10, 2022
Updated the AmazonEventBridgeReadOnlyAccess policy.	You can now view endpoint information. • AWS managed policy: AmazonEventBridgeReadOnlyAccess	April 07, 2022

Change	Description	Release Date
Added support for global endpoints.	Amazon EventBridge now supports using global endpoints to help make your application regional-fault tolerant at no additional cost. To learn more, see the following: • Making applications Regional-fault tolerant with global endpoints in EventBridge • `CreateEndpoint`	April 07, 2022
Added support for archives and event replays.	Amazon EventBridge now supports using archives to store events, and event replays to replay the events from an archive. To learn more, see the following: • Creating event archives in Amazon EventBridge. • `CreateArchive` • `StartReplay`	November 05, 2020
Added support for dead-letter queues and retry policy for targets.	Amazon EventBridge now supports using dead-letter queues and defining a retry policy for targets. To learn more, see the following: • Using dead-letter queues to process undelivered events in EventBridge. • `PutTargets`	October 12, 2020
Added support for JSONSchema Draft4 format schemas.	Amazon EventBridge now supports schemas in JSONSchema Draft 4 format. You can also now export schemas using the EventBridge API. To learn more, see the following. • Amazon EventBridge schemas • `Export` in the EventBridge Schema Registry API Reference.	September 28, 2020

Change	Description	Release Date
Resource-based policies for the EventBridge Schema Registry	The Amazon EventBridge Schema Registry now supports resource-based policies. For more information, see the following. • Resource-based policies for Amazon EventBridge schemas • `Policy` in the EventBridge Schema Registry API Reference • RegistryPolicy Resource Type in the AWS CloudFormation User Guide	April 30, 2020
Tags for Event Buses	This release allows you to create and manage tags for event buses. You can add tags when creating an event bus, and add or manage existing tags by calling the related API. For more information, see the following. • Tagging resources in Amazon EventBridge • Tag-based policies in Amazon EventBridge • `TagResource` • `UntagResource` • `ListTagsForResource`	February 24, 2020
Increased service quotas	Amazon EventBridge has increased quotas for invocations and for `PutEvents` . Quotas vary by region, and can be increased if necessary.	February 11, 2020

Change	Description	Release Date
Added a new topic on transforming target input, and added a link to Application Auto Scaling Events.	Improved documentation on the input transform er. • Amazon EventBridge input transformation • Use Input Transformer to extract data from an event and input that data to the target • Tutorial: Use input transformers to transform events in EventBridge Added a link to Application Auto Scaling Events. • Application Auto Scaling Events and EventBrid ge • Events from AWS services	December 20, 2019
Content-based filtering		December 19, 2019
Added links to Amazon Augmented AI event examples.	Added a link to the Amazon Augmented AI topic in the Amazon SageMaker AI Developer Guide that provides example events for Amazon Augmented AI. For more information, see the following. • Use Events in Amazon Augmented AI • Events from AWS services	December 13, 2019
Added links to Amazon Chime event examples.	Added a link to the Amazon Chime topic that provides example events for that service. For more information, see the following. • Automating Amazon Chime with EventBridge • Events from AWS services	December 12, 2019

Change	Description	Release Date
Amazon EventBridge Schemas	You can now manage schemas and generate code bindings for events in Amazon EventBridge. For more information, see the following. • Amazon EventBridge schemas • EventBridge Schemas API Reference • EventSchemas Resource Type Reference in CloudFormation	December 1, 2019
AWS CloudFormation support for Event Buses	CloudFormation now supports the EventBus resource. It also supports the EventBusName parameter in both the EventBusPolicy and Rule resources. For more information, see Amazon EventBridge Resource Type Reference .	October 7, 2019
New service	Initial release of Amazon EventBridge.	July 11, 2019