aws

Developer Guide

Amazon Kinesis Data Streams



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

Amazon Kinesis Data Streams: Developer 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 Kinesis Data Streams? 1
What can I do with Kinesis Data Streams? 1
Benefits of using Kinesis Data Streams 2
Related services 3
Terminology and concepts 4
Review the high-level architecture of Kinesis Data Streams 4
Become familiar with the terminology of Kinesis Data Streams 5
Kinesis Data Stream 5
Data Record 5
Capacity Mode 5
Retention Period 5
Producer 6
Consumer 6
Amazon Kinesis Data Streams Application 6
Shard 6
Partition Key 7
Sequence Number 7
Kinesis Client Library 7
Application Name 7
Server-Side Encryption 8
Data delivery 8
Quotas and limits 10
API Limits 12
KDS Control Plane API Limits 13
KDS Data Plane API Limits 17
Increasing Quotas 19
Complete prerequisites to set up Amazon Kinesis Data Streams 21
Sign up for an AWS account 21
Download libraries and tools 21
Configure your development environment 22
Use the AWS CLI to perform Amazon Kinesis Data Streams operations 23
Tutorial: Install and configure the AWS CLI for Kinesis Data Streams 23
Install the AWS CLI 23
Configure the AWS CLI 25

Tutorial: Perform basic Kinesis Data Streams operations using the AWS CLI ... 25
Step 1: Create a stream ... 25
Step 2: Put a record ... 27
Step 3: Get the record ... 28
Step 4: Clean up ... 31
Getting started tutorials ... 32
Tutorial: Process real-time stock data using KPL and KCL 2.x ... 32
Complete prerequisites ... 33
Create a data stream ... 34
Create an IAM policy and user ... 35
Download and build the code ... 40
Implement the producer ... 41
Implement the consumer ... 45
(Optional) Extend the consumer ... 50
Clean up resources ... 51
Tutorial: Process real-time stock data using KPL and KCL 1.x ... 53
Complete prerequisites ... 54
Create a data stream ... 55
Create an IAM policy and user ... 56
Download and build the implementation code ... 61
Implement the producer ... 62
Implement the consumer ... 67
(Optional) Extend the consumer ... 71
Clean up resources ... 72
Tutorial: Analyze real-time stock data using Amazon Managed Service for Apache Flink ... 73
Prerequisites ... 74
Step 1: Set up the AWS CLI ... 75
Step 2: Create an application ... 76
Tutorial: Use AWS Lambda with Amazon Kinesis Data Streams ... 93
Use the AWS Streaming Data Solution for Amazon Kinesis ... 94
Create and manage Kinesis data streams ... 95
Choose the right mode to stream in ... 95
What are the different modes in Kinesis Data Streams? ... 96
On-demand Standard mode features and use cases ... 97
On-demand Advantage mode features and use cases ... 98
Provisioned mode features and use cases ... 99

Switch between modes ... 100
Create a stream using the AWS Management Console ... 101
Create a stream using the APIs ... 102
Build the Kinesis Data Streams client ... 102
Create the stream ... 102
Update a stream ... 104
Use the console ... 104
Use the API ... 106
Use the AWS CLI ... 106
List streams ... 107
List shards ... 108
Delete a stream ... 111
Reshard a stream ... 112
Decide on a strategy for resharding ... 113
Split a shard ... 114
Merge two shards ... 115
Complete the resharding action ... 116
Change the data retention period ... 119
Tag your resources ... 120
Review tag basics ... 120
Track costs using tagging ... 122
Understand tag restrictions ... 122
Tag streams using the Kinesis Data Streams console ... 122
Tag streams using the AWS CLI ... 124
Tag streams using the Kinesis Data Streams APIs ... 125
Tag consumers using the AWS CLI ... 125
Tag consumers using the Kinesis Data Streams APIs ... 126
Handle large records ... 126
How large records work ... 127
Update your stream to use large records ... 127
Optimize your stream performance with large records ... 128
Mitigate throttling with large records ... 128
Handle large records using the Kinesis Data Streams APIs ... 128
AWS components compatible with large records ... 129
Regions where large records are supported ... 133
Perform resilience testing with AWS Fault Injection Service ... 135

Provisioned throughput exception errors 136
Expired iterator exception errors 139
Service-managed record distribution 141
How service-managed record distribution works 141
When to use service-managed record distribution 142
Comparison of record distribution strategies 142
Transition behavior 143
Important considerations 144
Configure service-managed distribution 145
Prerequisites 145
Create a stream with service-managed record distribution 146
Enable service-managed record distribution on an existing stream 146
Verify the current record distribution strategy 147
Configure the record distribution strategy in the console 148
Switch back to user-managed partition keys 149
Safe migration patterns 150
Error handling 152
IAM permissions 152
Write records to service-managed streams 153
Producer API behavior with service-managed distribution 153
Examples 154
Error handling 155
Migrate existing producers 156
Best practices 156
Read records from service-managed streams 157
GetRecords API behavior 157
SubscribeToShard API behavior 158
Consumer considerations 158
Monitor consumption from service-managed streams 159
KPL and KCL compatibility 159
Upgrade KPL to send null partition keys 160
Grant DescribeStreamSummary permission to the KPL 160
No changes required for existing producers and consumers 160
Aggregation is not supported on service-managed streams 161
KCL considerations 162
Summary 162

Integration case studies ... 163
AWS Lambda triggers with service-managed streams ... 163
Amazon Data Firehose consuming from service-managed streams ... 163
Amazon Managed Service for Apache Flink consuming from service-managed streams ... 164
CloudWatch metric streams with service-managed streams ... 164
CloudFront real-time logs to service-managed streams ... 165
Amazon EMR with service-managed streams ... 165
Amazon EventBridge Pipes with service-managed streams ... 166
Amazon DynamoDB with Kinesis data streams ... 166
Amazon Redshift streaming ingestion from service-managed streams ... 166
Amazon Kinesis Agent with service-managed streams ... 167
Amazon EventBridge event bus with service-managed streams ... 167
Streaming tables and Amazon S3 delivery with service-managed streams ... 167
Write data to Kinesis Data Streams ... 168
Develop producers using the Amazon Kinesis Producer Library (KPL) ... 169
Review the role of the KPL ... 170
Realize the advantages of using the KPL ... 170
Understand when not to use the KPL ... 171
Install the KPL ... 172
Migrate to KPL 1.x ... 172
Transition to Amazon Trust Services (ATS) certificates for the KPL ... 177
KPL supported platforms ... 177
KPL key concepts ... 178
Integrate the KPL with producer code ... 180
Write to your Kinesis data stream using the KPL ... 182
Configure the KPL ... 184
Implement consumer de-aggregation ... 185
Use the KPL with Amazon Data Firehose ... 188
Use the KPL with the AWS Glue Schema Registry ... 189
Configure the KPL proxy configuration ... 189
KPL version lifecycle policy ... 190
Develop producers using the Kinesis Data Streams API with the AWS SDK for Java ... 191
Add data to a stream ... 192
Interact with data using the AWS Glue Schema Registry ... 198
Write to Amazon Kinesis Data Streams using Kinesis Agent ... 198
Complete the prerequisites for Kinesis Agent ... 199

Download and install the agent 200
Configure and start the agent 201
Specify the agent configuration settings 202
Monitor multiple file directories and write to multiple streams 205
Use the agent to pre-process data 206
Use agent CLI commands 210
FAQ 211
Write to Kinesis Data Streams using other AWS services 212
Write to Kinesis Data Streams using AWS Amplify 213
Write to Kinesis Data Streams using Amazon Aurora 213
Write to Kinesis Data Streams using Amazon CloudFront 213
Write to Kinesis Data Streams using Amazon CloudWatch Logs 214
Write to Kinesis Data Streams using Connect Customer 214
Write to Kinesis Data Streams using AWS Database Migration Service 214
Write to Kinesis Data Streams using Amazon DynamoDB 215
Write to Kinesis Data Streams using Amazon EventBridge 215
Write to Kinesis Data Streams using AWS IoT Core 215
Write to Kinesis Data Streams using Amazon Relational Database Service 215
Write to Kinesis Data Streams usingAmazon Pinpoint 216
Write to Kinesis Data Streams using Amazon Quantum Ledger Database (Amazon QLDB) 216
Write to Kinesis Data Streams using third-party integrations 216
Apache Flink 217
Fluentd 217
Debezium 217
Oracle GoldenGate 217
Kafka Connect 217
Adobe Experience 217
Striim 218
Troubleshoot Kinesis Data Streams producers 218
My producer application is writing at a slower rate than expected 218
I receive an unauthorized KMS master key permission error 220
Troubleshoot other common issues for producers 220
Optimize Kinesis Data Streams producers 221
Customize KPL retries and rate limit behavior 221
Apply best practices to KPL aggregation 222

Read data from Kinesis Data Streams 224
Develop enhanced fan-out consumers with dedicated throughput 225
Differences between shared throughput consumer and enhanced fan-out consumer 226
Regions supported for up to 50 enhanced fan-out consumers (On-demand Advantage only) 133
Manage enhanced fan-out consumers 229
Use the Data Viewer in the Kinesis console 231
Query your data streams in the Kinesis console 232
Use Kinesis Client Library 233
What is Kinesis Client Library? 233
KCL key features and benefits 233
KCL concepts 234
DynamoDB metadata tables and load balancing in KCL 235
Develop consumers with KCL 239
Multi-stream processing with KCL 251
Use the AWS Glue Schema registry with KCL 253
IAM permissions required for KCL consumer applications 254
KCL configurations 261
KCL version lifecycle policy 276
Single table format for KCL 277
Migrate from previous KCL versions 283
Previous KCL version documentation 304
Develop consumers with the AWS SDK for Java 385
Develop shared-throughput consumers with the AWS SDK for Java 386
Develop enhanced fan-out consumers with the AWS SDK for Java 392
Interact with data using the AWS Glue Schema Registry 394
Develop consumers using AWS Lambda 395
Develop consumers using Managed Service for Apache Flink 395
Develop consumers using Amazon Data Firehose 395
Use other AWS services to read data from Kinesis Data Streams 396
Read data from Kinesis Data Streams using Amazon EMR 396
Read data from Kinesis Data Streams using Amazon EventBridge Pipes 396
Read data from Kinesis Data Streams using AWS Glue 396
Read data from Kinesis Data Streams using Amazon Redshift 397
Read from Kinesis Data Streams using third-party integrations 397
Apache Flink 397

Adobe Experience Platform ... 398
Apache Druid ... 398
Apache Spark ... 398
Databricks ... 398
Kafka Confluent Platform ... 398
Kinesumer ... 399
Talend ... 399
Troubleshoot Kinesis Data Streams consumers ... 399
Compilation error with the LeaseManagementConfig constructor ... 399
Some Kinesis Data Streams records are skipped when using the Kinesis Client Library ... 401
Records belonging to the same shard are processed by different record processors at the same time ... 401
The consumer application is reading at a slower rate than expected ... 402
GetRecords returns an empty records array even when there is data in the stream ... 402
The shard iterator expires unexpectedly ... 403
Consumer record processing is falling behind ... 404
Unauthorized KMS key permission error ... 405
DynamoDbException: The document path provided in the update expression is invalid for update ... 405
Troubleshoot other common issues for consumers ... 405
Optimize Kinesis Data Streams consumers ... 406
Improve low-latency processing ... 406
Process serialized data using AWS Lambda with the Amazon Kinesis Producer Library ... 407
Use resharding, scaling, and parallel processing to change the number of shards ... 407
Handle duplicate records ... 409
Handle startup, shutdown, and throttling ... 411
Streaming tables and S3 delivery ... 414
How data delivery works ... 415
Delivery destinations ... 415
Data flow ... 416
Key capabilities ... 416
Requirements ... 417
Data delivery concepts ... 418
Data freshness ... 418
Input formats for streaming tables ... 418
Input formats for general purpose Amazon S3 buckets ... 418

Schema enforcement ... 419
No backfill ... 419
New table per delivery ... 419
Dead-letter queue ... 419
Inline compaction ... 420
Delivery states ... 420
Append-only delivery ... 420
Streaming tables ... 420
How it works ... 421
Getting started ... 423
Manage streaming tables ... 424
Iceberg behaviors for streaming table ... 431
S3 general purpose delivery ... 435
How it works ... 435
Getting started ... 438
Manage S3 delivery ... 440
S3 output key template for Amazon S3 delivery ... 446
IAM permissions for data delivery ... 449
Lifecycle management permissions ... 449
Service execution role ... 450
Security for data delivery ... 458
Encryption in transit ... 459
Encryption at rest ... 459
Source stream encryption ... 460
IAM access control ... 460
AWS CloudTrail logging ... 461
Monitoring data delivery ... 461
CloudWatch metrics ... 461
CloudWatch Logs ... 464
AWS CloudTrail logging ... 466
Best practices for data delivery ... 467
Throughput and data freshness ... 467
Schema management (streaming tables on Apache Iceberg) ... 467
S3 object layout (general purpose Amazon S3 buckets) ... 468
Security ... 468
Dead-letter queue ... 468

Monitoring ... 469
Troubleshooting data delivery ... 469
Delivery stuck in CREATING state ... 469
Delivery in FAILED state ... 470
High data freshness ... 470
Failed records greater than zero ... 470
No data appearing at the destination ... 471
Delivery suspended ... 471
Permission denied errors in CloudWatch Logs ... 472
Delivery not available for my stream ... 472
Delivery stops after a schema change ... 472
Cannot delete a stream ... 472
Delivery quotas and limits ... 473
Resource quotas ... 473
API limits ... 473
Delivery constraints ... 474
Update constraints ... 474
Monitor Kinesis Data Streams ... 476
Monitor the Kinesis Data Streams service with CloudWatch ... 476
Amazon Kinesis Data Streams dimensions and metrics ... 477
Access Amazon CloudWatch metrics for Kinesis Data Streams ... 492
Monitor Kinesis Data Streams Agent health with CloudWatch ... 493
Monitor with CloudWatch ... 493
Log Amazon Kinesis Data Streams API calls with AWS CloudTrail ... 494
Kinesis Data Streams information in CloudTrail ... 494
Example: Kinesis Data Streams log file entries ... 496
Monitor the KCL with CloudWatch ... 500
Metrics and namespace ... 500
Metric levels and dimensions ... 500
Metric configuration ... 501
List of metrics ... 502
Monitor the KPL with CloudWatch ... 519
Metrics, dimensions, and namespaces ... 519
Metric level and granularity ... 519
Local access and Amazon CloudWatch upload ... 521
List of metrics ... 521

Security 525
Data protection in Kinesis Data Streams 526
What is server-side encryption for Kinesis Data Streams? 526
Costs, Regions, and performance considerations 528
How do I get started with server-side encryption? 529
Create and use user-generated KMS keys 530
Permissions to use user-generated KMS keys 530
Verify and Troubleshoot KMS key permissions 533
Use Kinesis Data Streams with interface VPC endpoints 533
Controlling access to Kinesis Data Streams resources using IAM 536
Policy syntax 538
Actions for Kinesis Data Streams 539
Amazon Resource Names (ARNs) for Kinesis Data Streams 539
Example policies for Kinesis Data Streams 539
Share your data stream with another account 542
Configure an AWS Lambda function to read from Kinesis Data Streams in another account 548
Share access using resource-based policies 549
Access control for streaming tables and Amazon S3 delivery 551
Test with dry run 551
Supported APIs 552
AWS CloudTrail logging 553
Using dry run with the AWS CLI 553
Using dry run with AWS SDKs 553
Compliance validation 554
Resilience in Kinesis Data Streams 555
Disaster recovery in Kinesis Data Streams 555
Infrastructure Security 556
Security best practices for Kinesis Data Streams 557
Implement least privilege access 557
Use IAM roles 557
Implement server-side encryption in dependent resources 557
Use CloudTrail to monitor API calls 558
Working with AWS SDKs 559
Code examples 560
Basics 561

Learn the basics 561
Actions 565
Serverless examples 622
Invoke a Lambda function from a Kinesis trigger 622
Reporting batch item failures for Lambda functions with a Kinesis trigger 633
Document history 647

What is Amazon Kinesis Data Streams?

You can use Amazon Kinesis Data Streams to collect and process large streams of data records in real time. You can create data-processing applications, known as *Kinesis Data Streams applications*. A typical Kinesis Data Streams application reads data from a *data stream* as data records. These applications can use the Kinesis Client Library, and they can run on Amazon EC2 instances. You can send the processed records to dashboards, use them to generate alerts, dynamically change pricing and advertising strategies, or send data to a variety of other AWS services. For information about Kinesis Data Streams features and pricing, see Amazon Kinesis Data Streams.

Kinesis Data Streams is part of the Kinesis streaming data platform, along with Firehose, Kinesis Video Streams, and Managed Service for Apache Flink.

For more information about AWS big data solutions, see Big Data on AWS. For more information about AWS streaming data solutions, see What is Streaming Data?.

Topics

- What can I do with Kinesis Data Streams?
- Benefits of using Kinesis Data Streams
- Related services

What can I do with Kinesis Data Streams?

You can use Kinesis Data Streams for rapid and continuous data intake and aggregation. The type of data used can include IT infrastructure log data, application logs, social media, market data feeds, and web clickstream data. Because the response time for the data intake and processing is in real time, the processing is typically lightweight.

The following are typical scenarios for using Kinesis Data Streams:

Accelerated log and data feed intake and processing

> You can have producers push data directly into a stream. For example, push system and application logs and they are available for processing in seconds. This prevents the log data from being lost if the front end or application server fails. Kinesis Data Streams provides accelerated data feed intake because you don't batch the data on the servers before you submit it for intake.

Real-time metrics and reporting

You can use data collected into Kinesis Data Streams for simple data analysis and reporting in real time. For example, your data-processing application can work on metrics and reporting for system and application logs as the data is streaming in, rather than wait to receive batches of data.

Real-time data analytics

This combines the power of parallel processing with the value of real-time data. For example, process website clickstreams in real time, and then analyze site usability engagement using multiple different Kinesis Data Streams applications running in parallel.

Complex stream processing

You can create Directed Acyclic Graphs (DAGs) of Kinesis Data Streams applications and data streams. This typically involves putting data from multiple Kinesis Data Streams applications into another stream for downstream processing by a different Kinesis Data Streams application.

Benefits of using Kinesis Data Streams

Although you can use Kinesis Data Streams to solve a variety of streaming data problems, a common use is the real-time aggregation of data followed by loading the aggregate data into a data warehouse or map-reduce cluster.

Data is put into Kinesis data streams, which ensures durability and elasticity. The delay between the time a record is put into the stream and the time it can be retrieved (put-to-get delay) is typically less than 1 second. In other words, a Kinesis Data Streams application can start consuming the data from the stream almost immediately after the data is added. The managed service aspect of Kinesis Data Streams relieves you of the operational burden of creating and running a data intake pipeline. You can create streaming map-reduce–type applications. The elasticity of Kinesis Data Streams enables you to scale the stream up or down, so that you never lose data records before they expire.

Multiple Kinesis Data Streams applications can consume data from a stream, so that multiple actions, like archiving and processing, can take place concurrently and independently. For example, two applications can read data from the same stream. The first application calculates running aggregates and updates an Amazon DynamoDB table, and the second application compresses and archives data to a data store like Amazon Simple Storage Service (Amazon S3). The DynamoDB table with running aggregates is then read by a dashboard for up-to-the-minute reports.

The Kinesis Client Library enables fault-tolerant consumption of data from streams and provides scaling support for Kinesis Data Streams applications.

Related services

For information about using Amazon EMR clusters to read and process Kinesis data streams directly, see Kinesis Connector.

Amazon Kinesis Data Streams Terminology and concepts

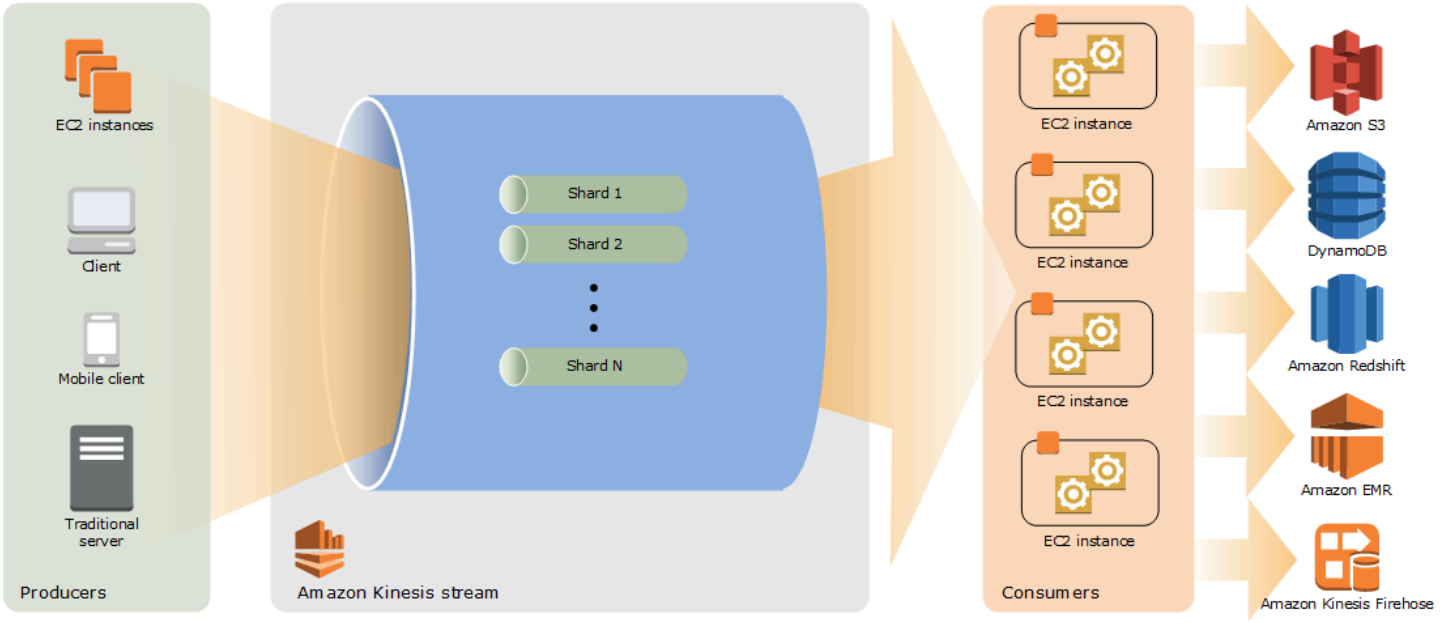
Before you get started with Amazon Kinesis Data Streams, learn about its architecture and terminology.

Topics

- Review the high-level architecture of Kinesis Data Streams
- Become familiar with the terminology of Kinesis Data Streams

Review the high-level architecture of Kinesis Data Streams

The following diagram illustrates the high-level architecture of Kinesis Data Streams. The *producers* continually push data to Kinesis Data Streams, and the *consumers* process the data in real time. Consumers (such as a custom application running on Amazon EC2 or an Amazon Data Firehose delivery stream) can store their results using an AWS service such as Amazon DynamoDB, Amazon Redshift, or Amazon S3.



Become familiar with the terminology of Kinesis Data Streams

Kinesis Data Stream

A *Kinesis data stream* is a set of shards. Each shard has a sequence of data records. Each data record has a sequence number that is assigned by Kinesis Data Streams.

Data Record

A *data record* is the unit of data stored in a Kinesis data stream. Data records are composed of a sequence number, a partition key, and a data blob, which is an immutable sequence of bytes. Kinesis Data Streams does not inspect, interpret, or change the data in the blob in any way. A data blob can be up to 1 MB.

Capacity Mode

A data stream *capacity mode* determines how capacity is managed and how you are charged for the usage of your data stream. Currently, in Kinesis Data Streams, you can choose between an **on-demand** mode and a **provisioned** mode for your data streams. For more information, see Choose the right mode to stream in.

With the **on-demand** mode, Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput. You are charged only for the actual throughput that you use and Kinesis Data Streams automatically accommodates your workloads' throughput needs as they ramp up or down. For more information, see On-demand Standard mode features and use cases.

With the **provisioned** mode, you must specify the number of shards for the data stream. The total capacity of a data stream is the sum of the capacities of its shards. You can increase or decrease the number of shards in a data stream as needed and you are charged for the number of shards at an hourly rate. For more information, see Provisioned mode features and use cases.

Retention Period

The *retention period* is the length of time that data records are accessible after they are added to the stream. A stream's retention period is set to a default of 24 hours after creation. You can increase the retention period up to 8760 hours (365 days) using the IncreaseStreamRetentionPeriod operation, and decrease the retention period down to a minimum of 24 hours using the DecreaseStreamRetentionPeriod operation. Additional charges apply for

streams with a retention period set to more than 24 hours. For more information, see Amazon Kinesis Data Streams Pricing.

Producer

Producers put records into Amazon Kinesis Data Streams. For example, a web server sending log data to a stream is a producer.

Consumer

Consumers get records from Amazon Kinesis Data Streams and process them. These consumers are known as Amazon Kinesis Data Streams Application.

Amazon Kinesis Data Streams Application

An *Amazon Kinesis Data Streams application* is a consumer of a stream that commonly runs on a fleet of EC2 instances.

There are two types of consumers that you can develop: shared fan-out consumers and enhanced fan-out consumers. To learn about the differences between them, and to see how you can create each type of consumer, see Read data from Amazon Kinesis Data Streams.

The output of a Kinesis Data Streams application can be input for another stream, enabling you to create complex topologies that process data in real time. An application can also send data to a variety of other AWS services. There can be multiple applications for one stream, and each application can consume data from the stream independently and concurrently.

Shard

A *shard* is a uniquely identified sequence of data records in a stream. A stream is composed of one or more shards, each of which provides a fixed unit of capacity. Each shard can support up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second and up to 1,000 records per second for writes, up to a maximum total data write rate of 1 MB per second (including partition keys). The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards.

If your data rate increases, you can increase or decrease the number of shards allocated to your stream. For more information, see Reshard a stream.

Partition Key

A *partition key* is used to group data by shard within a stream. Kinesis Data Streams segregates the data records belonging to a stream into multiple shards. It uses the partition key that is associated with each data record to determine which shard a given data record belongs to. Partition keys are Unicode strings, with a maximum length limit of 256 characters for each key. An MD5 hash function is used to map partition keys to 128-bit integer values and to map associated data records to shards using the hash key ranges of the shards. When an application puts data into a stream, it must specify a partition key.

Sequence Number

Each data record has a *sequence number* that is unique per partition-key within its shard. Kinesis Data Streams assigns the sequence number after you write to the stream with `client.putRecords` or `client.putRecord`. Sequence numbers for the same partition key generally increase over time. The longer the time period between write requests, the larger the sequence numbers become.

> **Note**
>
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each dataset.

Kinesis Client Library

Kinesis Client Library is compiled into your application to enable fault-tolerant consumption of data from the stream. Kinesis Client Library makes sure that for every shard there is a record processor running and processing that shard. The library also simplifies reading data from the stream. Kinesis Client Library uses Amazon DynamoDB tables to store metadata related to data consumption. It creates three tables per application that is processing data. For more information, see Use Kinesis Client Library.

Application Name

The name of an Amazon Kinesis Data Streams application identifies the application. Each of your applications must have a unique name that is scoped to the AWS account and Region used by

the application. This name is used as a name for the control table in Amazon DynamoDB and the namespace for Amazon CloudWatch metrics.

Server-Side Encryption

Amazon Kinesis Data Streams can automatically encrypt sensitive data as a producer enters it into a stream. Kinesis Data Streams uses AWS KMS master keys for encryption. For more information, see Data protection in Amazon Kinesis Data Streams.

> **Note**
>
> To read from or write to an encrypted stream, producer and consumer applications must have permission to access the master key. For information about granting permissions to producer and consumer applications, see the section called "Permissions to use user-generated KMS keys".

> **Note**
>
> Using server-side encryption incurs AWS Key Management Service (AWS KMS) costs. For more information, see AWS Key Management Service Pricing.

Data delivery

Data delivery is a fully managed capability that delivers streaming data from a Kinesis data stream to an analytics or storage destination without any infrastructure to manage. There are no connectors, consumer applications, or compute resources to provision. Kinesis Data Streams reads from your stream, buffers and aggregates records, and delivers them to your configured destination within minutes. Data delivery supports two destination types: streaming tables on Apache Iceberg, and general purpose Amazon S3 buckets. For more information, see Streaming tables and S3 delivery for Amazon Kinesis Data Streams.

Delivery to streaming tables in Iceberg format

Continuously materialize streaming data as queryable Apache Iceberg tables on Amazon S3 Tables with no ETL pipeline required. Delivery to streaming tables continuously materializes stream records as Apache Iceberg tables on Amazon S3 Tables. Built-in intelligent inline compaction

eliminates small file proliferation and keeps query performance predictable without sacrificing data freshness. Query or transform data with any engine, including Apache Spark, Trino, Apache Flink, or Amazon Athena, for real-time analytics directly on your streaming data.

Delivery to Amazon S3

Deliver streaming data in source format to Amazon S3 buckets for backup, archival, and downstream data processing. Delivery to Amazon S3 delivers streaming records directly to general purpose Amazon S3 buckets in the original source format. Use this capability for long-term data retention, compliance archival, replay and reprocessing workflows, or as a staging layer for downstream batch processing pipelines. Combined with delivery to streaming tables, you can build a complete lakehouse architecture with real-time queryable tables for analytics alongside full-fidelity raw backup in Amazon S3.

Quotas and limits

The following table describes stream and shard quotas and limits for Amazon Kinesis Data Streams.

Quota	On-demand mode	Provisioned mode
Number of data streams	There's no upper quota on the number of streams within your AWS account. By default, you can create up to 50 data streams with the on-demand capacity mode. If you require an increase of this quota, raise a support ticket.	There's no upper quota on the number of streams with the provisioned mode within an account.
Number of shards	There's no upper limit. Number of shards depends on the amount of data ingested and the level of throughpu t you require. Kinesis Data Streams automatically scales the number of shards in response to changes in data volume and traffic.	There's no upper limit. The default shard quota is 20,000 shards per AWS account for the following AWS Regions: • *US East (N. Virginia)* • *US West (Oregon)* • *Europe (Ireland)* For all other Regions, the default shard quota is 1,000 or 6,000 shards per AWS account. You can view your account's shard quota and utilization through the Service Quotas console at https://eusc-de-east-1.con

Quota	On-demand mode	Provisioned mode
		sole.amazonaws-eusc.eu/servicequotas/. To request an increase to the shard quota, use the Service Quotas console or AWS CLI. For more information, see Requesting a quota increase.
Data stream throughput	By default, new data streams created with the on-demand capacity mode have 4 MB/s of write and 8 MB/s of read throughput. In *US East (N. Virginia)*, *US West (Oregon)*, and *Europe (Ireland)* AWS Regions, data streams with the on-demand capacity mode scale up to 10 GB/s of write and 20 GB/s read throughput. For other Regions, data streams with the on-demand capacity mode scale up to 200 MB/s of write and 400 MB/s read throughput. If you require an increase up to 10 GB/s write and 20 GB/s read capacity for these Regions, submit a support ticket.	There is no upper limit. Maximum throughput depends on the number of shards provisioned for the stream. Each shard can support up to 1 MB/sec or 1,000 records/sec write throughput or up to 2 MB/sec or 2,000 records/sec read throughput. If you need more ingest capacity, you can easily scale up the number of shards in the stream using the AWS Management Console or the UpdateShardCount API.
Data payload size	The maximum size of the data payload of a record before `base64-encoding` is up to 10 MiB. Kinesis is designed to handle intermittent large records (1-10MiB in size) using burst capacity.	

Quota	On-demand mode	Provisioned mode
`GetRecords` transaction size	GetRecords can retrieve up to 10 MB of data per call from a single shard, and up to 10,000 records per call. Each call to `GetRecords` is counted as one read transaction. Each shard can support up to five read transactions per second. Each read transaction can provide up to 10,000 records with an upper quota of 10 MB per transaction.	
Data read rate per shard	Each shard can support up to a maximum total data read rate of 2 MB per second via GetRecords. If a call to `GetRecords` returns 10 MB, subsequent calls made within the next 5 seconds throw an exception.	
Number of registered consumers per data stream	With Kinesis On-demand Advantage mode, you can create up to 50 registered consumers (Enhanced Fan-out). With Kinesis On-Demand Standard and Kinesis Provisioned modes, you can create up to 20 registered consumers (Enhanced Fan-out Limit) for each data stream.	
Switching between provisioned and on-demand modes	For each data stream in your AWS account, you can switch between the on-demand and provisioned capacity modes twice within 24 hours.	
Streaming tables and Amazon S3 delivery	For quotas and limits that apply to streaming tables and Amazon S3 delivery, see the section called “Delivery quotas and limits”.	Not supported. Streaming tables and Amazon S3 delivery require on-demand capacity mode.

API Limits

Like most AWS APIs, Kinesis Data Streams API operations are rate-limited. The following limits apply per AWS account per Region. For more information on Kinesis Data Streams APIs, see the Amazon Kinesis API Reference.

KDS Control Plane API Limits

The following section describes limits for the KDS control plane APIs. KDS control plane APIs let you create and manage your data streams. These limits apply per AWS account per Region.

Control Plane API Limits

API	API call limit	Per Account/Stream	Description
AddTagsToStream	5 transactions per second (TPS)	Per Account	50 tags per data stream
CreateStream	5 TPS	Per Account	There is no upper quota on the number of streams you can have in an account. You receive a `LimitExceededException` when making a `CreateStream` request when you try to do one of the following: • Have more than five streams in the `CREATING` state at any point in time. • Create more shards than are authorized for your account.
DecreaseStreamRetentionPeriod	5 TPS	Per Stream	The minimum value of a data stream's retention period is 24 hours.

API	API call limit	Per Account/Stream	Description
DeleteResourcePolicy	5 TPS	Per Account	If you require an increase of this limit, raise a Support ticket.
DeleteStream	5 TPS	Per Account	
DeregisterStreamConsumer	5 TPS	Per Stream	
DescribeAccountSettings	5 TPS	Per account	
DescribeLimits	1 TPS	Per Account	
DescribeStream	10 TPS	Per Account	
DescribeStreamConsumer	20 TPS	Per Stream	
DescribeStreamSummary	20 TPS	Per Account	
DisableEnhancedMonitoring	5 TPS	Per Stream	
EnableEnhancedMonitoring	5 TPS	Per Stream	
GetResourcePolicy	5 TPS	Per Account	If you require an increase of this limit, raise a Support ticket.
IncreaseStreamRetentionPeriod	5 TPS	Per Stream	The maximum value of a stream's retention period is 8760 hours (365 days).

API	API call limit	Per Account/Stream	Description
ListShards	1000 TPS	Per Stream	
ListStreamConsumers	5 TPS	Per Stream	
ListStreams	5 TPS	Per Account	
ListTagsForStream	5 TPS	Per Stream	
MergeShards	5 TPS	Per Stream	Only applicable for provisioned.
PutResourcePolicy	5 TPS	Per Account	If you require an increase of this limit, raise a Support ticket.
RegisterStreamConsumer	5 TPS	Per Stream	You can register up to 20 consumers per data stream. A given consumer can only be registered with one data stream at a time. Only 5 consumers can be created simultaneously. In other words, you cannot have more than 5 consumers in a `CREATING` status at the same time.
RemoveTagsFromStream	5 TPS	Per Stream	
SplitShard	5 TPS	Per Stream	Only applicable for provisioned

API	API call limit	Per Account/Stream	Description
StartStreamEncryption		Per Stream	You can successfully apply a new AWS KMS key for server-side encryption 25 times in a rolling 24-hour period.
StopStreamEncryption		Per Stream	You can successfully disable server-side encryption 25 times in a rolling 24-hour period.
UpdateShardCount		Per Stream	Only applicable for provisioned. The default limit on number of shards is 10,000. There are additional limits on this API. For more information, see UpdateShardCount.
UpdateStreamMode		Per stream	For each data stream in your AWS account, you can switch between the on-demand and provisioned capacity modes twice within 24 hours.

API	API call limit	Per Account/Stream	Description
UpdateStreamWarmThroughput	5 TPS	Per account	The maximum warm throughput that can be configured is the on-demand mode's data stream throughput limit for the account and Region.
UpdateAccountSettings	5 TPS	Per account	Enable or disable account settings, such as On-demand Advantage mode.

KDS Data Plane API Limits

The following section describes the limits for the KDS data plane APIs. KDS data plane APIs enable you to use your data streams for collecting and processing data records in real time. These limits apply per shard within your data streams.

Data Plane API limits

API	API call limit	Payload limit	Additional details
GetRecords	5 TPS	The maximum number of records that can be returned per call is 10,000. The maximum size of data that `GetRecords` can return is 10 MB.	If a call returns this amount of data, subsequent calls made within the next 5 seconds th row `ProvisionedThroughputExceededException`. If there is insufficient provisioned

API	API call limit	Payload limit	Additional details
			throughput on the stream, subsequent calls made within the next 1 second throw `ProvisionedThroughputExceededException`.
GetShardIterator	5 TPS		A shard iterator expires 5 minutes after it is returned to the requester. If a GetShardIterator request is made too often, you receive a ProvisionedThroughputExceededException.
PutRecord	1000 TPS	Each shard can support writes up to 1,000 records per second, up to a maximum data write total of 10MiB per second.	Kinesis is designed to handle intermittent large records (1-10MiB in size) using burst capacity.

API	API call limit	Payload limit	Additional details
PutRecords		Each PutRecords request can support up to 500 records. Each record in the request can be as large as 10 MiB, up to a limit of 10 MiB for the entire request, including partition keys. Each shard can support writes up to 1,000 records per second, up to a maximum data write total of 1 MB per second.	Kinesis is designed to handle intermitt ent large records (1-10MiB in size) using burst capacity.
SubscribeToShard	You can make one call to Subscribe ToShard per second per registered consumer per shard.		If you call Subscribe ToShard again with the same ConsumerA RN and ShardId within 5 seconds of a successful call, you'll get a Resource InUseException.

Increasing Quotas

You can use Service Quotas to request an increase for a quota, if the quota is adjustable. Some requests are automatically resolved, while others are submitted to AWS Support. You can track the status of a quota increase request that is submitted to AWS Support. Requests to increase Service Quotas do not receive priority support. If you have an urgent request, contact AWS Support. For more information, see What Is Service Quotas?

To request a service quota increase, follow the procedure outlined in Requesting a Quota Increase.

Complete prerequisites to set up Amazon Kinesis Data Streams

Before you use Amazon Kinesis Data Streams for the first time, complete the following tasks to set up your environment.

Tasks

- Sign up for an AWS account
- Download libraries and tools
- Configure your development environment

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*.

Download libraries and tools

The following libraries and tools will help you work with Kinesis Data Streams:

- The Amazon Kinesis API Reference is the basic set of operations that Kinesis Data Streams supports. For more information about performing basic operations using Java code, see the following:
  - Develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java
  - Develop consumers with the AWS SDK for Java
  - Create and manage Kinesis data streams
- The AWS SDKs for Go, Java, JavaScript, .NET, PHP, Python, and Ruby include Kinesis Data Streams support and samples. If your version of the AWS SDK for Java does not include samples for Kinesis Data Streams, you can also download them from GitHub.
- The Kinesis Client Library (KCL) provides an easy-to-use programming model for processing data. The KCL can help you get started quickly with Kinesis Data Streams in Java, Node.js, .NET, Python, and Ruby. For more information see Reading Data from Streams.

- The AWS Command Line Interface supports Kinesis Data Streams. The AWS CLI enables you to control multiple AWS services from the command line and automate them through scripts.

Configure your development environment

To use the KCL, ensure that your Java development environment meets the following requirements:

- Java 1.7 (Java SE 7 JDK) or later. You can download the latest Java software from Java SE Downloads on the Oracle website.
- Apache Commons package (Code, HTTP Client, and Logging)
- Jackson JSON processor

Note that the AWS SDK for Java includes Apache Commons and Jackson in the third-party folder. However, the SDK for Java works with Java 1.6, while the Kinesis Client Library requires Java 1.7.

Use the AWS CLI to perform Amazon Kinesis Data Streams operations

This section shows you how to perform basic Amazon Kinesis Data Streams operations using the AWS Command Line Interface. You will learn fundamental Kinesis Data Streams data flow principles and the steps necessary to put and get data from an Kinesis data stream.

If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in Amazon Kinesis Data Streams Terminology and concepts.

Topics

- Tutorial: Install and configure the AWS CLI for Kinesis Data Streams
- Tutorial: Perform basic Kinesis Data Streams operations using the AWS CLI

For CLI access, you need an access key ID and a secret access key. Use temporary credentials instead of long-term access keys when possible. Temporary credentials include an access key ID, a secret access key, and a security token that indicates when the credentials expire. For more information, see Using temporary credentials with AWS resources in the *IAM User Guide*.

You can find detailed step-by-step IAM and security key set up instructions at Create an IAM User.

In this section, the specific commands discussed are given verbatim, except where specific values are necessarily different for each run. Also, the examples are using the US West (Oregon) region, but the steps in this section work in any of the regions where Kinesis Data Streams is supported.

Tutorial: Install and configure the AWS CLI for Kinesis Data Streams

Install the AWS CLI

For detailed steps on how to install the AWS CLI for Windows and for Linux, OS X, and Unix operating systems, see Installing the AWS CLI.

Use the following command to list available options and services:

```
aws help
```

You will be using the Kinesis Data Streams service, so you can review the AWS CLI subcommands related to Kinesis Data Streams using the following command:

```
aws kinesis help
```

This command results in output that includes the available Kinesis Data Streams commands:

```
AVAILABLE COMMANDS

       o add-tags-to-stream

       o create-stream

       o delete-stream

       o describe-stream

       o get-records

       o get-shard-iterator

       o help

       o list-streams

       o list-tags-for-stream

       o merge-shards

       o put-record

       o put-records

       o remove-tags-from-stream

       o split-shard

       o wait
```

This command list corresponds to the Kinesis Data Streams API documented in the Amazon Kinesis Service API Reference. For example, the `create-stream` command corresponds to the `CreateStream` API action.

The AWS CLI is now successfully installed, but not configured. This is shown in the next section.

Configure the AWS CLI

For general use, the `aws configure` command is the fastest way to set up your AWS CLI installation. For more information, see Configuring the AWS CLI.

Tutorial: Perform basic Kinesis Data Streams operations using the AWS CLI

This section describes basic use of a Kinesis data stream from the command line using the AWS CLI. Be sure you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and concepts.

> **Note**
>
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS Free Tier. When you are finished with this tutorial, delete your AWS resources to stop incurring charges. For more information, see Step 4: Clean up.

Topics

- Step 1: Create a stream
- Step 2: Put a record
- Step 3: Get the record
- Step 4: Clean up

Step 1: Create a stream

Your first step is to create a stream and verify that it was successfully created. Use the following command to create a stream named "Foo":

```
aws kinesis create-stream --stream-name Foo
```

Next, issue the following command to check on the stream's creation progress:

```
aws kinesis describe-stream-summary --stream-name Foo
```

You should get output that is similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "CREATING",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

In this example, the stream has a status CREATING, which means it is not yet ready to use. Check again in a few moments, and you should see output similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
```

```
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

There is information in this output that you don't need for this tutorial. The important information for now is `"StreamStatus": "ACTIVE"`, which tells you that the stream is ready to be used, and the information on the single shard that you requested. You can also verify the existence of your new stream by using the `list-streams` command, as shown here:

```
aws kinesis list-streams
```

Output:

```
{
    "StreamNames": [
        "Foo"
    ]
}
```

Step 2: Put a record

Now that you have an active stream, you are ready to put some data. For this tutorial, you will use the simplest possible command, `put-record`, which puts a single data record containing the text "testdata" into the stream:

```
aws kinesis put-record --stream-name Foo --partition-key 123 --data testdata
```

This command, if successful, will result in output similar to the following example:

```
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber": "49546986683135544286507457936321625675700192471156785154"
}
```

Congratulations, you just added data to a stream! Next you will see how to get data out of the stream.

Step 3: Get the record

GetShardIterator

Before you can get data from the stream, you must obtain the shard iterator for the shard you are interested in. A shard iterator represents the position of the stream and shard from which the consumer (`get-record` command in this case) will read. You'll use the `get-shard-iterator` command as follows:

```
aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-iterator-type
 TRIM_HORIZON --stream-name Foo
```

Recall that the `aws kinesis` commands have a Kinesis Data Streams API behind them, so if you are curious about any of the parameters shown, you can read about them in the `GetShardIterator` API reference topic. Successful execution will result in output similar to the following example:

```
{
    "ShardIterator": "AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd
+efGN2aHFdkH1rJl4BL9Wyrk+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="
}
```

The long string of seemingly random characters is the shard iterator (yours will be different). You must copy/paste the shard iterator into the get command, shown next. Shard iterators have a valid lifetime of 300 seconds, which should be enough time for you to copy/paste the shard iterator into the next command. You must remove any newlines from your shard iterator before pasting to the next command. If you get an error message that the shard iterator is no longer valid, run the `get-shard-iterator` command again.

GetRecords

The `get-records` command gets data from the stream, and it resolves to a call to `GetRecords` in the Kinesis Data Streams API. The shard iterator specifies the position in the shard from which you want to start reading data records sequentially. If there are no records available in the portion of the shard that the iterator points to, `GetRecords` returns an empty list. It might take multiple calls to get to a portion of the shard that contains records.

In the following example of the `get-records` command:

```
aws kinesis get-records --shard-iterator
 AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG4lNR1EMi
+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

If you are running this tutorial from a Unix-type command processor such as bash, you can automate the acquisition of the shard iterator using a nested command, like this:

```
SHARD_ITERATOR=$(aws kinesis get-shard-iterator --shard-id shardId-000000000000 --
shard-iterator-type TRIM_HORIZON --stream-name Foo --query 'ShardIterator')

aws kinesis get-records --shard-iterator $SHARD_ITERATOR
```

If you are running this tutorial from a system that supports PowerShell, you can automate acquisition of the shard iterator using a command such as this:

```
aws kinesis get-records --shard-iterator ((aws kinesis get-shard-iterator --shard-id
 shardId-000000000000 --shard-iterator-type TRIM_HORIZON --stream-name Foo).split('"')
[4])
```

The successful result of the `get-records` command will request records from your stream for the shard that you specified when you obtained the shard iterator, as in the following example:

```
{
  "Records":[ {
    "Data":"dGVzdGRhdGE=",
    "PartitionKey":"123”,
    "ApproximateArrivalTimestamp": 1.441215410867E9,
    "SequenceNumber":"49544985256907370027570885864065577703022652638596431874"
  } ],
  "MillisBehindLatest":24000,

 "NextShardIterator":"AAAAAAAAAAEDOW3ugseWPE4503kqN1yN1UaodY8unE0sYslMUmC6lX9hlig5+t4RtZM0/
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvPOZvrUIudb8UkH3V
}
```

Note that `get-records` is described above as a *request*, which means you may receive zero or more records even if there are records in your stream. Any records returned may not represent all the records currently in your stream. This is normal, and production code will poll the stream for

records at appropriate intervals. This polling speed will vary depending on your specific application design requirements.

In your record in this part of the tutorial, you will notice that the data appears to be garbage – and it's not the clear text `testdata` we sent. This is due to the way `put-record` uses Base64 encoding to allow you to send binary data. However, the Kinesis Data Streams support in the AWS CLI does not provide Base64 *decoding* because Base64 decoding to raw binary content printed to stdout can lead to undesired behavior and potential security issues on certain platforms and terminals. If you use a Base64 decoder (for example, https://www.base64decode.org/) to manually decode `dGVzdGRhdGE=` you will see that it is, in fact, `testdata`. This is sufficient for the sake of this tutorial because, in practice, the AWS CLI is rarely used to consume data. More often, it is used to monitor the state of the stream and obtain information, as shown previously (`describe-stream` and `list-streams`). For more information about the KCL, see Developing Custom Consumers with Shared Throughput Using KCL.

`get-records` doesn't always return all records in the stream/shard specified. When that happens, use the `NextShardIterator` from the last result to get the next set of records. If more data were being put into the stream, which is the normal situation in production applications, you could keep polling for data using `get-records` each time. However, if you do not call `get-records` using the next shard iterator within the 300 second shard iterator lifetime, you will get an error message, and you must use the `get-shard-iterator` command to get a fresh shard iterator.

Also provided in this output is `MillisBehindLatest`, which is the number of milliseconds the GetRecords operation's response is from the tip of the stream, indicating how far behind current time the consumer is. A value of zero indicates record processing is caught up, and there are no new records to process at this moment. In the case of this tutorial, you may see a number that's quite large if you've been taking time to read along as you go. By default, data records stay in a stream for 24 hours waiting for you to retrieve them. This time frame is called the retention period and it is configurable up to 365 days.

A successful `get-records` result will always have a `NextShardIterator` even if there are no more records currently in the stream. This is a polling model that assumes a producer is potentially putting more records into the stream at any given time. Although you can write your own polling routines, if you use the previously mentioned KCL for developing consumer applications, this polling is taken care of for you.

If you call `get-records` until there are no more records in the stream and shard you are pulling from, you will see output with empty records similar to the following example:

```
{
    "Records": [],
    "NextShardIterator": "AAAAAAAAAAGCJ5jzQNjmdhO6B/YDIDE56jmZmrmMA/r1WjoHXC/
kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wul/
EhyNeSs5DYXLSSC5XCapmCAYGFjYER69QSdQjxMmBPE/hiybFDi5qtkT6/PsZNz6kFoqtDk="
}
```

Step 4: Clean up

Delete your stream to free up resources and avoid unintended charges to your account. Do this any time you have created a stream and will not be using it, because charges accrue per stream whether you are putting and getting data with it or not. The clean-up command is as follows:

```
aws kinesis delete-stream --stream-name Foo
```

Success results in no output. Use `describe-stream` to check on the deletion progress:

```
aws kinesis describe-stream-summary --stream-name Foo
```

If you execute this command immediately after the delete command, you will see output similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "samplestream",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/samplestream",
        "StreamStatus": "ACTIVE",
```

After the stream is fully deleted, `describe-stream` will result in a "not found" error:

```
A client error (ResourceNotFoundException) occurred when calling the
 DescribeStreamSummary operation:
Stream Foo under account 123456789012 not found.
```

Getting started tutorials for Amazon Kinesis Data Streams

Amazon Kinesis Data Streams provides a number of different solutions to ingesting and consuming data from Kinesis data streams. The tutorials in this section are designed to further assist you in understanding Amazon Kinesis Data Streams concepts and functionality and identify the solution that meets your needs.

Topics

- Tutorial: Process real-time stock data using KPL and KCL 2.x
- Tutorial: Process real-time stock data using KPL and KCL 1.x
- Tutorial: Analyze real-time stock data using Amazon Managed Service for Apache Flink
- Tutorial: Use AWS Lambda with Amazon Kinesis Data Streams
- Use the AWS Streaming Data Solution for Amazon Kinesis

Tutorial: Process real-time stock data using KPL and KCL 2.x

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a basic Amazon Kinesis Data Streams application that performs calculations on the stream. You will learn how to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near real time.

> **Important**
>
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS Free Tier. After the consumer application starts, it also incurs nominal charges for Amazon DynamoDB usage. The consumer application uses DynamoDB to track processing state. When you are finished with this application, delete your AWS resources to stop incurring charges. For more information, see Clean up resources.

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator that has a starting point of real market data for the top 25 stocks by market capitalization as of February 2015. If you have access to

a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

You can work through the steps in this tutorial on your desktop or laptop computer and run both the producer and consumer code on the same machine or any platform that supports the defined requirements.

The examples shown use the US West (Oregon) Region, but they work on any of the AWS Regions that support Kinesis Data Streams.

Tasks

- Complete prerequisites
- Create a data stream
- Create an IAM policy and user
- Download and build the code
- Implement the producer
- Implement the consumer
- (Optional) Extend the consumer
- Clean up resources

Complete prerequisites

You must meet the following requirements to complete this tutorial:

Create and use an Amazon Web Services Account

Before you begin, make sure that you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and concepts, particularly with streams, shards, producers, and consumers. It is also helpful to have completed the steps in the following guide: Tutorial: Install and configure the AWS CLI for Kinesis Data Streams.

You must have an AWS account and a web browser to access the AWS Management Console.

For console access, use your IAM user name and password to sign in to the AWS Management Console from the IAM sign-in page. For information about AWS security credentials, including

programmatic access and alternatives to long-term credentials, see AWS security credentials in the *IAM User Guide*. For details about signing in to your AWS account, see How to sign in to AWS in the *AWS Sign-In User Guide*.

For more information about IAM and security key setup instructions, see Create an IAM User.

Fulfill system software requirements

The system that you are using to run the application must have Java 7 or higher installed. To download and install the latest Java Development Kit (JDK), go to Oracle's Java SE installation site.

You need the latest AWS SDK for Java version.

The consumer application requires the Kinesis Client Library (KCL) version 2.2.9 or higher, which you can obtain from GitHub at https://github.com/awslabs/amazon-kinesis-client/tree/master.

Next steps

Create a data stream

Create a data stream

First, you must create the data stream that you will use in subsequent steps of this tutorial.

To create a stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. In the navigation bar, expand the Region selector and choose a Region.
4. Choose **Create Kinesis stream**.
5. Enter a name for your data stream (for example, **`StockTradeStream`**).
6. Enter **1** for the number of shards, but keep **Estimate the number of shards you'll need** collapsed.
7. Choose **Create Kinesis stream**.

On the **Kinesis streams** list page, the status of your stream appears as `CREATING` while the stream is being created. When the stream is ready to use, the status changes to `ACTIVE`.

If you choose the name of your stream, in the page that appears, the **Details** tab displays a summary of your data stream configuration. The **Monitoring** section displays monitoring information for the stream.

Next steps

Create an IAM policy and user

Create an IAM policy and user

Security best practices for AWS dictate the use of fine-grained permissions to control access to different resources. AWS Identity and Access Management (IAM) lets you to manage users and user permissions in AWS. An IAM policy explicitly lists actions that are allowed and the resources on which the actions are applicable.

The following are the minimum permissions generally required for Kinesis Data Streams producers and consumers.

Producer

Actions	Resource	Purpose
`DescribeStream` , `DescribeStreamSummary` , `DescribeStreamConsumer`	Kinesis data stream	Before attempting to read records, the consumer checks if if it's active, and if the shards are contained in the data str
`SubscribeToShard` , `RegisterStreamConsumer`	Kinesis data stream	Subscribes and registers consumers to a shard.
`PutRecord` , `PutRecords`	Kinesis data stream	Writes records to Kinesis Data Streams.

Consumer

Actions	Resource	Purpose
`DescribeStream`	Kinesis data stream	Before attempting to read records, the consumer checks if if it's active, and if the shards are contained in the data str

Actions	Resource	Purpose
`GetRecords` , `GetShardIterator`	Kinesis data stream	Reads records from a shard.
`CreateTable` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB table	If the consumer is developed using the Kinesis Client Libra 1.x or 2.x), it needs permissions to a DynamoDB table to tr state of the application.
`DeleteItem`	Amazon DynamoDB table	For when the consumer performs split/merge operations Streams shards.
`PutMetricData`	Amazon CloudWatch log	The KCL also uploads metrics to CloudWatch, which are us application.

For this tutorial, you will create a single IAM policy that grants all of the preceding permissions. In production, you might want to create two policies, one for producers and one for consumers.

To create an IAM policy

1. Locate the Amazon Resource Name (ARN) for the new data stream that you created in the previous step. You can find this ARN listed as **Stream ARN** at the top of the **Details** tab. The ARN format is as follows:

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   The AWS Region code; for example, `us-west-2`. For more information, see Region and Availability Zone Concepts.

   account

   The AWS account ID, as shown in Account Settings.

name

The name of the data stream that you created in the preceding step, which is `StockTradeStream`.

2. Determine the ARN for the DynamoDB table to be used by the consumer (and to be created by the first consumer instance). It must be in the following format:

```
arn:aws:dynamodb:region:account:table/name
```

The Region and account ID are identical to the values in the ARN of the data stream that you're using for this tutorial, but the *name* is the name of the DynamoDB table created and used by the consumer application. KCL uses the application name as the table name. In this step, use `StockTradesProcessor` for the DynamoDB table name, because that is the application name used in later steps in this tutorial.

3. In the IAM console, in **Policies** (https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/home#policies), choose **Create policy**. If this is the first time that you have worked with IAM policies, choose **Get Started**, **Create Policy**.
4. Choose **Select** next to **Policy Generator**.
5. Choose **Amazon Kinesis** as the AWS service.
6. Select `DescribeStream`, `GetShardIterator`, `GetRecords`, `PutRecord`, and `PutRecords` as the allowed actions.
7. Enter the ARN of the data stream that you're using in this tutorial.
8. Use **Add Statement** for each of the following:

AWS Service	Actions	ARN
Amazon DynamoDB	`CreateTable` , `DeleteItem` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	The ARN of the DynamoDB table that you created in Step 2 of this procedure.
Amazon CloudWatch	`PutMetricData`	*

The asterisk (*) that is used when specifying an ARN is not required. In this case, it's because there is no specific resource in CloudWatch on which the `PutMetricData` action is invoked.

9. Choose **Next Step**.
10. Change **Policy Name** to `StockTradeStreamPolicy`, review the code, and choose **Create Policy**.

The resulting policy document should look like this:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt123",
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream",
                "kinesis:PutRecord",
                "kinesis:PutRecords",
                "kinesis:GetShardIterator",
                "kinesis:GetRecords",
                "kinesis:ListShards",
                "kinesis:DescribeStreamSummary",
                "kinesis:RegisterStreamConsumer"
            ],
            "Resource": [
                "arn:aws-eusc:kinesis:us-west-2:111122223333:stream/
StockTradeStream"
            ]
        },
        {
            "Sid": "Stmt234",
            "Effect": "Allow",
            "Action": [
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": [
```

```
                "arn:aws-eusc:kinesis:us-west-2:111122223333:stream/
StockTradeStream/*"
            ]
        },
        {
            "Sid": "Stmt456",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws-eusc:dynamodb:us-west-2:111122223333:table/
StockTradesProcessor"
            ]
        },
        {
            "Sid": "Stmt789",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

To create an IAM user

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. On the **Users** page, choose **Add user**.
3. For **User name**, type `StockTradeStreamUser`.
4. For **Access type**, choose **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. Search by name for the policy that you created in the preceding procedure (`StockTradeStreamPolicy`. Select the box to the left of the policy name, and then choose **Next: Review**.
7. Review the details and summary, and then choose **Create user**.

8. Copy the **Access key ID**, and save it privately. Under **Secret access key**, choose **Show**, and save that key privately also.
9. Paste the access and secret keys to a local file in a safe place that only you can access. For this application, create a file named `~/.aws/credentials` (with strict permissions). The file should be in the following format:

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

To attach an IAM policy to a user

1. In the IAM console, open Policies and choose **Policy Actions**.
2. Choose `StockTradeStreamPolicy` and **Attach**.
3. Choose `StockTradeStreamUser` and **Attach Policy**.

Next steps

Download and build the code

Download and build the code

This topic provides sample implementation code for the sample stock trades ingestion into the data stream (*producer*) and the processing of this data (*consumer*).

To download and build the code

1. Download the source code from the https://github.com/aws-samples/amazon-kinesis-learning GitHub repo to your computer.
2. Create a project in your IDE with the source code, adhering to the provided directory structure.
3. Add the following libraries to the project:
   - Amazon Kinesis Client Library (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient

- Apache Commons Lang
- Apache Commons Logging
- Guava (Google Core Libraries For Java)
- Jackson Annotations
- Jackson Core
- Jackson Databind
- Jackson Dataformat: CBOR
- Joda Time

4. Depending on your IDE, the project might be built automatically. If not, build the project using the appropriate steps for your IDE.

If you complete these steps successfully, you are now ready to move to the next section, the section called “Implement the producer”.

Next steps

Implement the producer

This tutorial uses the real-world scenario of stock market trade monitoring. The following principles briefly explain how this scenario maps to the producer and its supporting code structure.

Refer to the source code and review the following information.

StockTrade class

An individual stock trade is represented by an instance of the StockTrade class. This instance contains attributes such as the ticker symbol, price, number of shares, the type of the trade (buy or sell), and an ID uniquely identifying the trade. This class is implemented for you.

Stream record

A stream is a sequence of records. A record is a serialization of a `StockTrade` instance in JSON format. For example:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
```

```
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator class

StockTradeGenerator has a method called `getRandomTrade()` that returns a new randomly generated stock trade every time it is invoked. This class is implemented for you.

StockTradesWriter class

The `main` method of the producer, StockTradesWriter continuously retrieves a random trade and then sends it to Kinesis Data Streams by performing the following tasks:

1. Reads the data stream name and Region name as input.
2. Uses the `KinesisAsyncClientBuilder` to set the Region, credentials, and client configuration.
3. Checks that the stream exists and is active (if not, it exits with an error).
4. In a continuous loop, calls the `StockTradeGenerator.getRandomTrade()` method and then the `sendStockTrade` method to send the trade to the stream every 100 milliseconds.

The `sendStockTrade` method of the `StockTradesWriter` class has the following code:

```
private static void sendStockTrade(StockTrade trade, KinesisAsyncClient
 kinesisClient,
            String streamName) {
       byte[] bytes = trade.toJsonAsBytes();
       // The bytes could be null if there is an issue with the JSON serialization
 by the Jackson JSON library.
       if (bytes == null) {
           LOG.warn("Could not get JSON bytes for stock trade");
           return;
       }

       LOG.info("Putting trade: " + trade.toString());
       PutRecordRequest request = PutRecordRequest.builder()
               .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol
 as the partition key, explained in the Supplemental Information section below.
               .streamName(streamName)
               .data(SdkBytes.fromByteArray(bytes))
```

```
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }
```

Refer to the following code breakdown:

- The `PutRecord` API expects a byte array, and you must convert trade to JSON format. This single line of code performs that operation:

  ```
  byte[] bytes = trade.toJsonAsBytes();
  ```

- Before you can send the trade, you create a new `PutRecordRequest` instance (called request in this case). Each `request` requires the stream name, partition key, and a data blob.

  ```
  PutPutRecordRequest request = PutRecordRequest.builder()
      .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol as the
   partition key, explained in the Supplemental Information section below.
      .streamName(streamName)
      .data(SdkBytes.fromByteArray(bytes))
      .build();
  ```

  The example uses a stock ticker as a partition key, which maps the record to a specific shard. In practice, you should have hundreds or thousands of partition keys per shard such that records are evenly dispersed across your stream. For more information about how to add data to a stream, see Write data to Amazon Kinesis Data Streams.

  Now `request` is ready to send to the client (the put operation):

  ```
    kinesisClient.putRecord(request).get();
  ```

- Error checking and logging are always useful additions. This code logs error conditions:

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

Add the try/catch block around the `put` operation:

```
try {
  kinesisClient.putRecord(request).get();
} catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
} catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again
 next cycle.", e);
}
```

This is because a Kinesis Data Streams put operation can fail because of a network error, or due to the data stream reaching its throughput limits and getting throttled. It is recommended that you carefully consider your retry policy for `put` operations to avoid data loss, such as using a retry.

- Status logging is helpful but optional:

```
LOG.info("Putting trade: " + trade.toString());
```

The producer shown here uses the Kinesis Data Streams API single record functionality, `PutRecord`. In practice, if an individual producer generates many records, it is often more efficient to use the multiple records functionality of `PutRecords` and send batches of records at a time. For more information, see Write data to Amazon Kinesis Data Streams.

To run the producer

1. Verify that the access key and secret key pair retrieved in Create an IAM policy and user are saved in the file `~/.aws/credentials`.
2. Run the `StockTradeWriter` class with the following arguments:

   ```
   StockTradeStream us-west-2
   ```

   If you created your stream in a region other than `us-west-2`, you have to specify that region here instead.

You should see output similar to the following:

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Your stock trades are now being ingested by Kinesis Data Streams.

Next steps

Implement the consumer

Implement the consumer

The consumer application in this tutorial continuously processes the stock trades in your data stream. It then outputs the most popular stocks being bought and sold every minute. The

application is built on top of the Kinesis Client Library (KCL), which does much of the heavy lifting common to consumer apps. For more information, see KCL 1.x and 2.x information.

Refer to the source code and review the following information.

StockTradesProcessor class

The main class of the consumer, provided for you, which performs the following tasks:

- Reads the application, data stream, and Region names, passed in as arguments.
- Creates a `KinesisAsyncClient` instance with the Region name.
- Creates a `StockTradeRecordProcessorFactory` instance that serves instances of `ShardRecordProcessor`, implemented by a `StockTradeRecordProcessor` instance.
- Creates a `ConfigsBuilder` instance with the `KinesisAsyncClient`, `StreamName`, `ApplicationName`, and `StockTradeRecordProcessorFactory` instance. This is useful for creating all configurations with default values.
- Creates a KCL scheduler (previously, in KCL versions 1.x it was known as the KCL worker) with the `ConfigsBuilder` instance.
- The scheduler creates a new thread for each shard (assigned to this consumer instance), which continuously loops to read records from the data stream. It then invokes the `StockTradeRecordProcessor` instance to process each batch of records received.

StockTradeRecordProcessor class

Implementation of the `StockTradeRecordProcessor` instance, which in turn implements five required methods: `initialize`, `processRecords`, `leaseLost`, `shardEnded`, and `shutdownRequested`.

The `initialize` and `shutdownRequested` methods are used by the KCL to let the record processor know when it should be ready to start receiving records and when it should expect to stop receiving records, respectively, so it can perform any application-specific setup and termination tasks. `leaseLost` and `shardEnded` are used to implement any logic for what to do when a lease is lost or a processing has reached the end of a shard. In this example, we simply log messages indicating these events.

The code for these methods is provided for you. The main processing happens in the `processRecords` method, which in turn uses `processRecord` for each record. This latter method is provided as the mostly empty skeleton code for you to implement in the next step, where it is explained in greater detail.

Also of note is the implementation of the support methods for `processRecord`: `reportStats`, and `resetStats`, which are empty in the original source code.

The `processRecords` method is implemented for you, and performs the following steps:

- For each record passed in, it calls `processRecord` on it.
- If at least 1 minute has elapsed since the last report, calls `reportStats()`, which prints out the latest stats, and then `resetStats()` which clears the stats so that the next interval includes only new records.
- Sets the next reporting time.
- If at least 1 minute has elapsed since the last checkpoint, calls `checkpoint()`.
- Sets the next checkpointing time.

This method uses 60-second intervals for the reporting and checkpointing rate. For more information about checkpointing, see Using the Kinesis Client Library.

StockStats class

This class provides data retention and statistics tracking for the most popular stocks over time. This code is provided for you and contains the following methods:

- `addStockTrade(StockTrade)`: injects the given `StockTrade` into the running statistics.
- `toString()`: returns the statistics in a formatted string.

This class tracks the most popular stock by keeping a running count of the total number of trades for each stock and the maximum count. It updates these counts whenever a stock trade arrives.

Add code to the methods of the `StockTradeRecordProcessor` class, as shown in the following steps.

To implement the consumer

1. Implement the `processRecord` method by instantiating a correctly sized `StockTrade` object and adding the record data to it, logging a warning if there's a problem.

```
byte[] arr = new byte[record.data().remaining()];
record.data().get(arr);
StockTrade trade = StockTrade.fromJsonAsBytes(arr);
```

```
    if (trade == null) {
        log.warn("Skipping record. Unable to parse record into StockTrade.
 Partition Key: " + record.partitionKey());
        return;
        }
stockStats.addStockTrade(trade);
```

2. Implement a `reportStats` method. Modify the output format to suit your preferences.

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute
 ******\n" +
stockStats + "\n" +
"*****************************************************************\n");
```

3. Implement the `resetStats` method, which creates a new `stockStats` instance.

```
stockStats = new StockStats();
```

4. Implement the following methods required by `ShardRecordProcessor` interface:

```
@Override
public void leaseLost(LeaseLostInput leaseLostInput) {
    log.info("Lost lease, so terminating.");
}

@Override
public void shardEnded(ShardEndedInput shardEndedInput) {
    try {
        log.info("Reached shard end checkpointing.");
        shardEndedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        log.error("Exception while checkpointing at shard end. Giving up.", e);
    }
}

@Override
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
```

```
    log.info("Scheduler is shutting down, checkpointing.");
    checkpoint(shutdownRequestedInput.checkpointer());
}

private void checkpoint(RecordProcessorCheckpointer checkpointer) {
    log.info("Checkpointing shard " + kinesisShardId);
    try {
        checkpointer.checkpoint();
    } catch (ShutdownException se) {
        // Ignore checkpoint if the processor instance has been shutdown (fail
 over).
        log.info("Caught shutdown exception, skipping checkpoint.", se);
    } catch (ThrottlingException e) {
        // Skip checkpoint when throttled. In practice, consider a backoff and
 retry policy.
        log.error("Caught throttling exception, skipping checkpoint.", e);
    } catch (InvalidStateException e) {
        // This indicates an issue with the DynamoDB table (check for table,
 provisioned IOPS).
        log.error("Cannot save checkpoint to the DynamoDB table used by the Amazon
 Kinesis Client Library.", e);
    }
}
```

To run the consumer

1. Run the producer that you wrote in to inject simulated stock trade records into your stream.
2. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials`.
3. Run the `StockTradesProcessor` class with the following arguments:

   ```
   StockTradesProcessor StockTradeStream us-west-2
   ```

   Note that if you created your stream in a Region other than `us-west-2`, you have to specify that Region here instead.

After a minute, you should see output like the following, refreshed every minute thereafter:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
****************************************************************
```

Next steps

(Optional) Extend the consumer

(Optional) Extend the consumer

This optional section shows how you can extend the consumer code for a slightly more elaborate scenario.

If you want to know about the biggest sell orders each minute, you can modify the `StockStats` class in three places to accommodate this new priority.

To extend the consumer

1. Add new instance variables:

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. Add the following code to `addStockTrade`:

```
if (type == TradeType.SELL) {
     if (largestSellOrderStock == null || trade.getQuantity() >
 largestSellOrderQuantity) {
         largestSellOrderStock = trade.getTickerSymbol();
         largestSellOrderQuantity = trade.getQuantity();
     }
 }
```

3. Modify the `toString` method to print the additional information:

```
public String toString() {
    return String.format(
        "Most popular stock being bought: %s, %d buys.%n" +
        "Most popular stock being sold: %s, %d sells.%n" +
        "Largest sell order: %d shares of %s.",
        getMostPopularStock(TradeType.BUY),
 getMostPopularStockCount(TradeType.BUY),
        getMostPopularStock(TradeType.SELL),
 getMostPopularStockCount(TradeType.SELL),
        largestSellOrderQuantity, largestSellOrderStock);
}
```

If you run the consumer now (remember to run the producer also), you should see output similar to this:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
Largest sell order: 996 shares of BUD.
****************************************************************
```

Next steps

Clean up resources

Clean up resources

Because you are paying to use the Kinesis data stream, make sure that you delete it and the corresponding Amazon DynamoDB table when you are done with it. Nominal charges occur on an active stream even when you aren't sending and getting records. This is because an active stream is using resources by continuously "listening" for incoming records and requests to get records.

To delete the stream and table

1. Shut down any producers and consumers that you might still have running.

2. Open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
3. Choose the stream that you created for this application (`StockTradeStream`).
4. Choose **Delete Stream**.
5. Open the DynamoDB console at https://eusc-de-east-1.console.amazonaws-eusc.eu/dynamodb/.
6. Delete the `StockTradesProcessor` table.

Summary

Processing a large amount of data in near real time doesn't require writing complicated code or developing a huge infrastructure. It is as basic as writing logic to process a small amount of data (like writing `processRecord(Record)`) but using Kinesis Data Streams to scale so that it works for a large amount of streaming data. You don't have to worry about how your processing would scale because Kinesis Data Streams handles it for you. All you have to do is send your streaming records to Kinesis Data Streams and write the logic to process each new record received.

Here are some potential enhancements for this application.

Aggregate across all shards

Currently, you get stats resulting from aggregation of the data records received by a single worker from a single shard. (A shard cannot be processed by more than one worker in a single application at the same time.) Of course, when you scale and have more than one shard, you might want to aggregate across all shards. You can do this by having a pipeline architecture where the output of each worker is fed into another stream with a single shard, which is processed by a worker that aggregates the outputs of the first stage. Because the data from the first stage is limited (one sample per minute per shard), it would easily be handled by one shard.

Scale processing

When the stream scales up to have many shards (because many producers are sending data), the way to scale the processing is to add more workers. You can run the workers in Amazon EC2 instances and use Auto Scaling groups.

Use connectors to Amazon S3/DynamoDB/Amazon Redshift/Storm

As a stream is continuously processed, its output can be sent to other destinations. AWS provides connectors to integrate Kinesis Data Streams with other AWS services and third-party tools.

Tutorial: Process real-time stock data using KPL and KCL 1.x

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a simple Amazon Kinesis Data Streams application that performs calculations on the stream. You will learn how to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near real time.

> **Important**
>
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS Free Tier. After the consumer application starts, it also incurs nominal charges for Amazon DynamoDB usage. The consumer application uses DynamoDB to track processing state. When you are finished with this application, delete your AWS resources to stop incurring charges. For more information, see Clean up resources.

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator that has a starting point of real market data for the top 25 stocks by market capitalization as of February 2015. If you have access to a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

You can work through the steps in this tutorial on your desktop or laptop computer and run both the producer and consumer code on the same machine or any platform that supports the defined requirements, such as Amazon Elastic Compute Cloud (Amazon EC2).

The examples shown use the US West (Oregon) Region, but they work on any of the AWS Regions that support Kinesis Data Streams.

Tasks

- Complete prerequisites
- Create a data stream
- Create an IAM policy and user

- Download and build the implementation code
- Implement the producer
- Implement the consumer
- (Optional) Extend the consumer
- Clean up resources

Complete prerequisites

The following are requirements for completing the Tutorial: Process real-time stock data using KPL and KCL 1.x.

Create and use an Amazon Web Services Account

Before you begin, ensure that you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and concepts, particularly streams, shards, producers, and consumers. It is also helpful to have completed Tutorial: Install and configure the AWS CLI for Kinesis Data Streams.

You need an AWS account and a web browser to access the AWS Management Console.

For console access, use your IAM user name and password to sign in to the AWS Management Console from the IAM sign-in page. For information about AWS security credentials, including programmatic access and alternatives to long-term credentials, see AWS security credentials in the *IAM User Guide*. For details about signing in to your AWS account, see How to sign in to AWS in the *AWS Sign-In User Guide*.

For more information about IAM and security key setup instructions, see Create an IAM User.

Fulfill system software requirements

The system used to run the application must have Java 7 or higher installed. To download and install the latest Java Development Kit (JDK), go to Oracle's Java SE installation site.

If you have a Java IDE, such as Eclipse, you can open the source code, edit, build, and run it.

You need the latest AWS SDK for Java version. If you are using Eclipse as your IDE, you can install the AWS Toolkit for Eclipse instead.

The consumer application requires the Kinesis Client Library (KCL) version 1.2.1 or higher, which you can obtain from GitHub at Kinesis Client Library (Java).

Next Steps

Create a data stream

Create a data stream

In the first step of the Tutorial: Process real-time stock data using KPL and KCL 1.x, you create the stream that you will use in subsequent steps.

To create a stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. In the navigation bar, expand the Region selector and choose a Region.
4. Choose **Create Kinesis stream**.
5. Enter a name for your stream (for example, **`StockTradeStream`**).
6. Enter **1** for the number of shards, but keep **Estimate the number of shards you'll need** collapsed.
7. Choose **Create Kinesis stream**.

On the **Kinesis streams** list page, the status of your stream is `CREATING` while the stream is being created. When the stream is ready to use, the status changes to `ACTIVE`. Choose the name of your stream. In the page that appears, the **Details** tab displays a summary of your stream configuration. The **Monitoring** section displays monitoring information for the stream.

Additional information about shards

When you begin to use Kinesis Data Streams outside of this tutorial, you might need to plan the stream creation process more carefully. You should plan for expected maximum demand when you provision shards. Using this scenario as an example, US stock market trading traffic peaks during the day (Eastern time) and demand estimates should be sampled from that time of day. You then have a choice to provision for the maximum expected demand, or scale your stream up and down in response to demand fluctuations.

A *shard* is a unit of throughput capacity. On the **Create Kinesis stream** page, expand **Estimate the number of shards you'll need**. Enter the average record size, the maximum records written per second, and the number of consuming applications, using the following guidelines:

Average record size

An estimate of the calculated average size of your records. If you don't know this value, use the estimated maximum record size for this value.

Max records written

Consider the number of entities providing data and the approximate number of records per second produced by each. For example, if you are getting stock trade data from 20 trading servers and each generates 250 trades per second, the total number of trades (records) per second is 5000/second.

Number of consuming applications

The number of applications that independently read from the stream to process the stream in a different way and produce different output. Each application can have multiple instances running on different machines (that is, run in a cluster) so that it can keep up with a high volume stream.

If the estimated number of shards shown exceeds your current shard limit, you might need to submit a request to increase that limit before you can create a stream with that number of shards. To request an increase to your shard limit, use the Kinesis Data Streams Limits form. For more information about streams and shards, see Create and manage Kinesis data streams.

Next steps

Create an IAM policy and user

Create an IAM policy and user

Security best practices for AWS dictate the use of fine-grained permissions to control access to different resources. AWS Identity and Access Management (IAM) allows you to manage users and user permissions in AWS. An IAM policy explicitly lists actions that are allowed and the resources on which the actions are applicable.

The following are the minimum permissions generally required for a Kinesis Data Streams producer and consumer.

Producer

Actions	Resource	Purpose
`DescribeStream` , `DescribeStreamSummary` , `DescribeStreamConsumer`	Kinesis data stream	Before attempting to write records, the producer checks if is active, and if the shards are contained in the stream, and consumer.
`SubscribeToShard` , `RegisterStreamConsumer`	Kinesis data stream	Subscribes and register a consumers to a Kinesis Data Stre
`PutRecord` , `PutRecords`	Kinesis data stream	Write records to Kinesis Data Streams.

Consumer

Actions	Resource	Purpose
`DescribeStream`	Kinesis data stream	Before attempting to read records, the consumer checks if is active, and if the shards are contained in the stream.
`GetRecords` , `GetShardIterator`	Kinesis data stream	Read records from a Kinesis Data Streams shard.
`CreateTable` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB table	If the consumer is developed using the Kinesis Client Libra permissions to a DynamoDB table to track the processing on. The first consumer started creates the table.
`DeleteItem`	Amazon DynamoDB table	For when the consumer performs split/merge operations Streams shards.
`PutMetricData`	Amazon CloudWatch log	The KCL also uploads metrics to CloudWatch, which are us application.

For this application, you create a single IAM policy that grants all of the preceding permissions. In practice, you might want to consider creating two policies, one for producers and one for consumers.

To create an IAM policy

1. Locate the Amazon Resource Name (ARN) for the new stream. You can find this ARN listed as **Stream ARN** at the top of the **Details** tab. The ARN format is as follows:

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   The Region code; for example, `us-west-2`. For more information, see Region and Availability Zone Concepts.

   account

   The AWS account ID, as shown in Account Settings.

   name

   The name of the stream from Create a data stream, which is `StockTradeStream`.
2. Determine the ARN for the DynamoDB table to be used by the consumer (and created by the first consumer instance). It must be in the following format:

   ```
   arn:aws:dynamodb:region:account:table/name
   ```

   The Region and account are from the same place as the previous step, but this time *name* is the name of the table created and used by the consumer application. The KCL used by the consumer uses the application name as the table name. Use `StockTradesProcessor`, which is the application name used later.
3. In the IAM console, in **Policies** (https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/home#policies), choose **Create policy**. If this is the first time that you have worked with IAM policies, choose **Get Started**, **Create Policy**.
4. Choose **Select** next to **Policy Generator**.
5. Choose **Amazon Kinesis** as the AWS service.
6. Select `DescribeStream`, `GetShardIterator`, `GetRecords`, `PutRecord`, and `PutRecords` as the allowed actions.

7. Enter the ARN that you created in Step 1.
8. Use **Add Statement** for each of the following:

AWS Service	Actions	ARN
Amazon DynamoDB	`CreateTable` , `DeleteItem` , `DescribeTable` , `GetItem`, `PutItem`, `Scan`, `UpdateItem`	The ARN you created in Step 2
Amazon CloudWatch	`PutMetricData`	*

   The asterisk (*) that is used when specifying an ARN is not required. In this case, it's because there is no specific resource in CloudWatch on which the `PutMetricData` action is invoked.
9. Choose **Next Step**.
10. Change **Policy Name** to `StockTradeStreamPolicy`, review the code, and choose **Create Policy**.

The resulting policy document should look something like the following:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt123",
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream",
                "kinesis:PutRecord",
                "kinesis:PutRecords",
                "kinesis:GetShardIterator",
                "kinesis:GetRecords",
                "kinesis:ListShards",
                "kinesis:DescribeStreamSummary",
```

```
                "kinesis:RegisterStreamConsumer"
            ],
            "Resource": [
                "arn:aws-eusc:kinesis:us-west-2:111122223333:stream/
StockTradeStream"
            ]
        },
        {
            "Sid": "Stmt234",
            "Effect": "Allow",
            "Action": [
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": [
                "arn:aws-eusc:kinesis:us-west-2:111122223333:stream/
StockTradeStream/*"
            ]
        },
        {
            "Sid": "Stmt456",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws-eusc:dynamodb:us-west-2:111122223333:table/
StockTradesProcessor"
            ]
        },
        {
            "Sid": "Stmt789",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

To create an IAM user

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. On the **Users** page, choose **Add user**.
3. For **User name**, type `StockTradeStreamUser`.
4. For **Access type**, choose **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. Search by name for the policy that you created. Select the box to the left of the policy name, and then choose **Next: Review**.
7. Review the details and summary, and then choose **Create user**.
8. Copy the **Access key ID**, and save it privately. Under **Secret access key**, choose **Show**, and save that key privately also.
9. Paste the access and secret keys to a local file in a safe place that only you can access. For this application, create a file named `~/.aws/credentials` (with strict permissions). The file should be in the following format:

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

To attach an IAM policy to a user

1. In the IAM console, open Policies and choose **Policy Actions**.
2. Choose `StockTradeStreamPolicy` and **Attach**.
3. Choose `StockTradeStreamUser` and **Attach Policy**.

Next Steps

Download and build the implementation code

Download and build the implementation code

Skeleton code is provided for the the section called “Tutorial: Process real-time stock data using KPL and KCL 1.x”. It contains a stub implementation for both the stock trade stream ingestion

(*producer*) and the processing of the data (*consumer*). The following procedure shows how to complete the implementation.

To download and build the implementation code

1. Download the source code to your computer.
2. Create a project in your favorite IDE with the source code, adhering to the provided directory structure.
3. Add the following libraries to the project:
   - Amazon Kinesis Client Library (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient
   - Apache Commons Lang
   - Apache Commons Logging
   - Guava (Google Core Libraries For Java)
   - Jackson Annotations
   - Jackson Core
   - Jackson Databind
   - Jackson Dataformat: CBOR
   - Joda Time
4. Depending on your IDE, the project might be built automatically. If not, build the project using the appropriate steps for your IDE.

If you complete these steps successfully, you are now ready to move to the next section, the section called “Implement the producer”. If your build generates errors at any stage, investigate and fix them before proceeding.

Next steps

Implement the producer

The application in the Tutorial: Process real-time stock data using KPL and KCL 1.x uses the real-world scenario of stock market trade monitoring. The following principles briefly explain how this scenario maps to the producer and supporting code structure.

Refer to the source code and review the following information.

StockTrade class

An individual stock trade is represented by an instance of the `StockTrade` class. This instance contains attributes such as the ticker symbol, price, number of shares, the type of the trade (buy or sell), and an ID uniquely identifying the trade. This class is implemented for you.

Stream record

A stream is a sequence of records. A record is a serialization of a `StockTrade` instance in JSON format. For example:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator class

`StockTradeGenerator` has a method called `getRandomTrade()` that returns a new randomly generated stock trade every time it is invoked. This class is implemented for you.

StockTradesWriter class

The `main` method of the producer, `StockTradesWriter` continuously retrieves a random trade and then sends it to Kinesis Data Streams by performing the following tasks:

1. Reads the stream name and Region name as input.
2. Creates an `AmazonKinesisClientBuilder`.
3. Uses the client builder to set the Region, credentials, and client configuration.
4. Builds an `AmazonKinesis` client using the client builder.
5. Checks that the stream exists and is active (if not, it exits with an error).

6. In a continuous loop, calls the `StockTradeGenerator.getRandomTrade()` method and then the `sendStockTrade` method to send the trade to the stream every 100 milliseconds.

The `sendStockTrade` method of the `StockTradesWriter` class has the following code:

```
private static void sendStockTrade(StockTrade trade, AmazonKinesis kinesisClient,
 String streamName) {
    byte[] bytes = trade.toJsonAsBytes();
    // The bytes could be null if there is an issue with the JSON serialization by
 the Jackson JSON library.
    if (bytes == null) {
        LOG.warn("Could not get JSON bytes for stock trade");
        return;
    }

    LOG.info("Putting trade: " + trade.toString());
    PutRecordRequest putRecord = new PutRecordRequest();
    putRecord.setStreamName(streamName);
    // We use the ticker symbol as the partition key, explained in the Supplemental
 Information section below.
    putRecord.setPartitionKey(trade.getTickerSymbol());
    putRecord.setData(ByteBuffer.wrap(bytes));

    try {
        kinesisClient.putRecord(putRecord);
    } catch (AmazonClientException ex) {
        LOG.warn("Error sending record to Amazon Kinesis.", ex);
    }
}
```

Refer to the following code breakdown:

- The `PutRecord` API expects a byte array, and you must convert `trade` to JSON format. This single line of code performs that operation:

  ```
  byte[] bytes = trade.toJsonAsBytes();
  ```

- Before you can send the trade, you create a new `PutRecordRequest` instance (called `putRecord` in this case):

  ```
  PutRecordRequest putRecord = new PutRecordRequest();
  ```

Each `PutRecord` call requires the stream name, partition key, and data blob. The following code populates these fields in the `putRecord` object using its `setXxxx()` methods:

```
putRecord.setStreamName(streamName);
putRecord.setPartitionKey(trade.getTickerSymbol());
putRecord.setData(ByteBuffer.wrap(bytes));
```

The example uses a stock ticket as a partition key, which maps the record to a specific shard. In practice, you should have hundreds or thousands of partition keys per shard such that records are evenly dispersed across your stream. For more information about how to add data to a stream, see Add data to a stream.

Now `putRecord` is ready to send to the client (the `put` operation):

```
kinesisClient.putRecord(putRecord);
```

- Error checking and logging are always useful additions. This code logs error conditions:

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

Add the try/catch block around the `put` operation:

```
try {
       kinesisClient.putRecord(putRecord);
} catch (AmazonClientException ex) {
       LOG.warn("Error sending record to Amazon Kinesis.", ex);
}
```

This is because a Kinesis Data Streams `put` operation can fail because of a network error, or due to the stream reaching its throughput limits and getting throttled. We recommend carefully considering your retry policy for `put` operations to avoid data loss, such as using a retry.

- Status logging is helpful but optional:

```
LOG.info("Putting trade: " + trade.toString());
```

The producer shown here uses the Kinesis Data Streams API single record functionality, `PutRecord`. In practice, if an individual producer generates many records, it is often more efficient to use the multiple records functionality of `PutRecords` and send batches of records at a time. For more information, see Add data to a stream.

To run the producer

1. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials`.
2. Run the `StockTradeWriter` class with the following arguments:

   ```
   StockTradeStream us-west-2
   ```

   If you created your stream in a Region other than `us-west-2`, you have to specify that Region here instead.

You should see output similar to the following:

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter
 sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Your stock trade stream is now being ingested by Kinesis Data Streams.

Next steps

Implement the consumer

Implement the consumer

The consumer application in the Tutorial: Process real-time stock data using KPL and KCL 1.x continuously processes the stock trades stream that you created in . It then outputs the most popular stocks being bought and sold every minute. The application is built on top of the Kinesis Client Library (KCL), which does much of the heavy lifting common to consumer apps. For more information, see Develop KCL 1.x consumers.

Refer to the source code and review the following information.

StockTradesProcessor class

Main class of the consumer, provided for you, which performs the following tasks:

- Reads the application, stream, and Region names, passed in as arguments.
- Reads credentials from `~/.aws/credentials`.
- Creates a `RecordProcessorFactory` instance that serves instances of `RecordProcessor`, implemented by a `StockTradeRecordProcessor` instance.
- Creates a KCL worker with the `RecordProcessorFactory` instance and a standard configuration including the stream name, credentials, and application name.
- The worker creates a new thread for each shard (assigned to this consumer instance), which continuously loops to read records from Kinesis Data Streams. It then invokes the `RecordProcessor` instance to process each batch of records received.

StockTradeRecordProcessor class

Implementation of the `RecordProcessor` instance, which in turn implements three required methods: `initialize`, `processRecords`, and `shutdown`.

As the names suggest, `initialize` and `shutdown` are used by the Kinesis Client Library to let the record processor know when it should be ready to start receiving records and when it should expect to stop receiving records, respectively, so it can do any application-specific setup and termination tasks. The code for these is provided for you. The main processing happens in the `processRecords` method, which in turn uses `processRecord` for each record. This latter method is provided as mostly empty skeleton code for you to implement in the next step, where it is explained further.

Also of note is the implementation of support methods for `processRecord`: `reportStats`, and `resetStats`, which are empty in the original source code.

The `processRecords` method is implemented for you, and performs the following steps:

- For each record passed in, calls `processRecord` on it.
- If at least 1 minute has elapsed since the last report, calls `reportStats()`, which prints out the latest stats, and then `resetStats()` which clears the stats so that the next interval includes only new records.
- Sets the next reporting time.
- If at least 1 minute has elapsed since the last checkpoint, calls `checkpoint()`.
- Sets the next checkpointing time.

This method uses 60-second intervals for the reporting and checkpointing rate. For more information about checkpointing, see Additional information about the consumer.

StockStats class

This class provides data retention and statistics tracking for the most popular stocks over time. This code is provided for you and contains the following methods:

- `addStockTrade(StockTrade)`: Injects the given `StockTrade` into the running statistics.
- `toString()`: Returns the statistics in a formatted string.

This class tracks the most popular stock by keeping a running count of the total number of trades for each stock and the maximum count. It updates these counts whenever a stock trade arrives.

Add code to the methods of the `StockTradeRecordProcessor` class, as shown in the following steps.

To implement the consumer

1. Implement the `processRecord` method by instantiating a correctly sized `StockTrade` object and adding the record data to it, logging a warning if there's a problem.

```
StockTrade trade = StockTrade.fromJsonAsBytes(record.getData().array());
if (trade == null) {
    LOG.warn("Skipping record. Unable to parse record into StockTrade. Partition
 Key: " + record.getPartitionKey());
    return;
}
stockStats.addStockTrade(trade);
```

2. Implement a simple `reportStats` method. Feel free to modify the output format to your preferences.

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute
 ******\n" +
                   stockStats + "\n" +

 "**************************************************************\n");
```

3. Finally, implement the `resetStats` method, which creates a new `stockStats` instance.

```
stockStats = new StockStats();
```

To run the consumer

1. Run the producer that you wrote in to inject simulated stock trade records into your stream.
2. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials` .
3. Run the `StockTradesProcessor` class with the following arguments:

```
StockTradesProcessor StockTradeStream us-west-2
```

   Note that if you created your stream in a Region other than `us-west-2`, you have to specify that Region here instead.

After a minute, you should see output like the following, refreshed every minute thereafter:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
**************************************************************
```

Additional information about the consumer

If you are familiar with the advantages of the Kinesis Client Library, discussed in Develop KCL 1.x consumers and elsewhere, you might wonder why you should use it here. Although you use only a single shard stream and a single consumer instance to process it, it is still easier to implement the

consumer using the KCL. Compare the code implementation steps in the producer section to the consumer, and you can see the comparative ease of implementing a consumer. This is largely due to the services that the KCL provides.

In this application, you focus on implementing a record processor class that can process individual records. You don't have to worry about how the records are fetched from Kinesis Data Streams; The KCL fetches the records and invoke the record processor whenever there are new records available. Also, you don't have to worry about how many shards and consumer instances there are. If the stream is scaled up, you don't have to rewrite your application to handle more than one shard or one consumer instance.

The term *checkpointing* means to record the point in the stream up to the data records that have been consumed and processed thus far. If the application crashes, the stream is read from that point and not from the beginning of the stream. The subject of checkpointing and the various design patterns and best practices for it are outside the scope of this chapter. However, it is something you may encounter in production environments.

As you learned in , the `put` operations in the Kinesis Data Streams API take a *partition key* as input. Kinesis Data Streams uses a partition key as a mechanism to split records across multiple shards (when there is more than one shard in the stream). The same partition key always routes to the same shard. This allows the consumer that processes a particular shard to be designed with the assumption that records with the same partition key are only sent to that consumer, and no records with the same partition key end up at any other consumer. Therefore, a consumer's worker can aggregate all records with the same partition key without worrying that it might be missing needed data.

In this application, the consumer's processing of records is not intensive, so you can use one shard and do the processing in the same thread as the KCL thread. However, in practice, consider first scaling up the number of shards. In some cases you may want to switch processing to a different thread, or use a thread pool if your record processing is expected to be intensive. In this way, the KCL can fetch new records more quickly while the other threads can process the records in parallel. Multithreaded design is not trivial and should be approached with advanced techniques, so increasing your shard count is usually the most effective way to scale up.

Next steps

(Optional) Extend the consumer

(Optional) Extend the consumer

The application in the Tutorial: Process real-time stock data using KPL and KCL 1.x might already be sufficient for your purposes. This optional section shows how you can extend the consumer code for a slightly more elaborate scenario.

If you want to know about the biggest sell orders each minute, you can modify the `StockStats` class in three places to accommodate this new priority.

To extend the consumer

1. Add new instance variables:

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. Add the following code to `addStockTrade`:

```
if (type == TradeType.SELL) {
    if (largestSellOrderStock == null || trade.getQuantity() >
largestSellOrderQuantity) {
        largestSellOrderStock = trade.getTickerSymbol();
        largestSellOrderQuantity = trade.getQuantity();
    }
}
```

3. Modify the `toString` method to print the additional information:

```
public String toString() {
    return String.format(
            "Most popular stock being bought: %s, %d buys.%n" +
            "Most popular stock being sold: %s, %d sells.%n" +
            "Largest sell order: %d shares of %s.",
            getMostPopularStock(TradeType.BUY),
getMostPopularStockCount(TradeType.BUY),
            getMostPopularStock(TradeType.SELL),
getMostPopularStockCount(TradeType.SELL),
            largestSellOrderQuantity, largestSellOrderStock);
}
```

If you run the consumer now (remember to run the producer also), you should see output similar to this:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
 Most popular stock being bought: WMT, 27 buys.
 Most popular stock being sold: PTR, 14 sells.
 Largest sell order: 996 shares of BUD.
 ****************************************************************
```

Next steps

Clean up resources

Clean up resources

Because you are paying to use the Kinesis data stream, make sure that you delete it and the corresponding Amazon DynamoDB table when you are done with it. Nominal charges occur on an active stream even when you aren't sending and getting records. This is because an active stream is using resources by continuously "listening" for incoming records and requests to get records.

To delete the stream and table

1. Shut down any producers and consumers that you may still have running.
2. Open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
3. Choose the stream that you created for this application (`StockTradeStream`).
4. Choose **Delete Stream**.
5. Open the DynamoDB console at https://eusc-de-east-1.console.amazonaws-eusc.eu/dynamodb/.
6. Delete the `StockTradesProcessor` table.

Summary

Processing a large amount of data in near real time doesn't require writing complicated code or developing a huge infrastructure. It is as basic as writing logic to process a small amount of data (like writing `processRecord(Record)`) but using Kinesis Data Streams to scale so that it works for a large amount of streaming data. You don't have to worry about how your processing would scale because Kinesis Data Streams handles it for you. All you have to do is send your streaming records to Kinesis Data Streams and write the logic to process each new record received.

Here are some potential enhancements for this application.

Aggregate across all shards

Currently, you get stats resulting from aggregation of the data records received by a single worker from a single shard. (A shard cannot be processed by more than one worker in a single application at the same time.) Of course, when you scale and have more than one shard, you might want to aggregate across all shards. You can do this by having a pipeline architecture where the output of each worker is fed into another stream with a single shard, which is processed by a worker that aggregates the outputs of the first stage. Because the data from the first stage is limited (one sample per minute per shard), it would easily be handled by one shard.

Scale processing

When the stream scales up to have many shards (because many producers are sending data), the way to scale the processing is to add more workers. You can run the workers in Amazon EC2 instances and use Auto Scaling groups.

Use connectors to Amazon S3/DynamoDB/Amazon Redshift/Storm

As a stream is continuously processed, its output can be sent to other destinations. AWS provides connectors to integrate Kinesis Data Streams with other AWS services and third-party tools.

Next steps

- For more information about using Kinesis Data Streams API operations, see Develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java, Develop shared-throughput consumers with the AWS SDK for Java, and Create and manage Kinesis data streams.
- For more information about the Kinesis Client Library, see Develop KCL 1.x consumers.
- For more information about how to optimize your application, see Optimize Amazon Kinesis Data Streams consumers.

Tutorial: Analyze real-time stock data using Amazon Managed Service for Apache Flink

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a simple Amazon Managed Service for Apache Flink application that performs calculations on the

stream. You will learn how to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near real time.

With Amazon Managed Service for Apache Flink, you can use Java or Scala to process and analyze streaming data. The service lets you author and run Java or Scala code against streaming sources to perform time-series analytics, feed real-time dashboards, and create real-time metrics.

You can build Flink applications in Managed Service for Apache Flink using open-source libraries based on Apache Flink. Apache Flink is a popular framework and engine for processing data streams.

> **Important**
>
> After you create two data streams and an application, your account incurs nominal charges for Kinesis Data Streams and Managed Service for Apache Flink usage because they are not eligible for the AWS Free Tier. When you are finished with this application, delete your AWS resources to stop incurring charges.

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator. If you have access to a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

The examples shown use the US West (Oregon) Region, but they work on any of the AWS Regions that support Managed Service for Apache Flink.

Tasks

- Prerequisites for completing the exercises
- Set up the AWS Command Line Interface (AWS CLI)
- Create and run a Managed Service for Apache Flink application

Prerequisites for completing the exercises

To complete the steps in this guide, you must have the following:

- Java Development Kit (JDK) version 8. Set the `JAVA_HOME` environment variable to point to your JDK install location.
- We recommend that you use a development environment (such as Eclipse Java Neon or IntelliJ Idea) to develop and compile your application.
- Git Client. Install the Git client if you haven't already.
- Apache Maven Compiler Plugin. Maven must be in your working path. To test your Apache Maven installation, enter the following:

```
$ mvn -version
```

To get started, go to Sign up for an AWS account.

Set up the AWS Command Line Interface (AWS CLI)

In this step, you download and configure the AWS CLI to use with Amazon Managed Service for Apache Flink.

Note

The getting started exercises in this guide assume that you are using administrator credentials (`adminuser`) in your account to perform the operations.

Note

If you already have the AWS CLI installed, you might need to upgrade to get the latest functionality. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*. To check the version of the AWS CLI, run the following command:

```
aws --version
```

The exercises in this tutorial require the following AWS CLI version or later:

```
aws-cli/1.16.63
```

To set up the AWS CLI

1. Download and configure the AWS CLI. For instructions, see the following topics in the *AWS Command Line Interface User Guide*:
   - Installing the AWS Command Line Interface
   - Configuring the AWS CLI
2. Add a named profile for the administrator user in the AWS CLI config file. You use this profile when executing the AWS CLI commands. For more information about named profiles, see Named Profiles in the *AWS Command Line Interface User Guide*.

   ```
   [profile adminuser]
   aws_access_key_id = adminuser access key ID
   aws_secret_access_key = adminuser secret access key
   region = aws-region
   ```

   For a list of available AWS Regions, see AWS Regions and Endpoints in the *Amazon Web Services General Reference*.
3. Verify the setup by entering the following help command at the command prompt:

   ```
   aws help
   ```

After you set up an AWS account and the AWS CLI, you can try the next exercise, in which you configure a sample application and test the end-to-end setup.

Next step

Create and run a Managed Service for Apache Flink application

Create and run a Managed Service for Apache Flink application

In this exercise, you create a Managed Service for Apache Flink application with data streams as a source and a sink.

This section contains the following steps:

- Create two Amazon Kinesis data streams
- Write Sample Records to the Input Stream

- Download and examine the Apache Flink streaming Java code
- Compile the application code
- Upload the Apache Flink streaming Java code
- Create and run the Managed Service for Apache Flink application

Create two Amazon Kinesis data streams

Before you create a Amazon Managed Service for Apache Flink for this exercise, create two Kinesis data streams (`ExampleInputStream` and `ExampleOutputStream`). Your application uses these streams for the application source and destination streams.

You can create these streams using either the Amazon Kinesis console or the following AWS CLI command. For console instructions, see Creating and Updating Data Streams.

To create the data streams (AWS CLI)

1. To create the first stream (`ExampleInputStream`), use the following Amazon Kinesis `create-stream` AWS CLI command.

```
$ aws kinesis create-stream \
--stream-name ExampleInputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

2. To create the second stream that the application uses to write output, run the same command, changing the stream name to `ExampleOutputStream`.

```
$ aws kinesis create-stream \
--stream-name ExampleOutputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

Write Sample Records to the Input Stream

In this section, you use a Python script to write sample records to the stream for the application to process.

> **Note**
>
> This section requires the AWS SDK for Python (Boto).

1. Create a file named `stock.py` with the following contents:

```
import datetime
import json
import random
import boto3

STREAM_NAME = "ExampleInputStream"


def get_data():
    return {
        "EVENT_TIME": datetime.datetime.now().isoformat(),
        "TICKER": random.choice(["AAPL", "AMZN", "MSFT", "INTC", "TBV"]),
        "PRICE": round(random.random() * 100, 2),
    }


def generate(stream_name, kinesis_client):
    while True:
        data = get_data()
        print(data)
        kinesis_client.put_record(
            StreamName=stream_name, Data=json.dumps(data),
 PartitionKey="partitionkey"
        )


if __name__ == "__main__":
    generate(STREAM_NAME, boto3.client("kinesis"))
```

2. Later in the tutorial, you run the `stock.py` script to send data to the application.

```
$ python stock.py
```

Download and examine the Apache Flink streaming Java code

The Java application code for this examples is available from GitHub. To download the application code, do the following:

1. Clone the remote repository with the following command:

```
git clone https://github.com/aws-samples/amazon-kinesis-data-analytics-java-examples.git
```

2. Navigate to the `GettingStarted` directory.

The application code is located in the `CustomSinkStreamingJob.java` and `CloudWatchLogSink.java` files. Note the following about the application code:

- The application uses a Kinesis source to read from the source stream. The following snippet creates the Kinesis sink:

```
return env.addSource(new FlinkKinesisConsumer<>(inputStreamName,
                new SimpleStringSchema(), inputProperties));
```

Compile the application code

In this section, you use the Apache Maven compiler to create the Java code for the application. For information about installing Apache Maven and the Java Development Kit (JDK), see Prerequisites for completing the exercises.

Your Java application requires the following components:

- A Project Object Model (pom.xml) file. This file contains information about the application's configuration and dependencies, including the Amazon Managed Service for Apache Flink libraries.
- A `main` method that contains the application's logic.

> **Note**
>
> **To use the Kinesis connector for the following application, you must download the source code for the connector and build it as described in the Apache Flink documentation.**

To create and compile the application code

1. Create a Java/Maven application in your development environment. For information about creating an application, see the documentation for your development environment:
   - Creating your first Java project (Eclipse Java Neon)
   - Creating, Running and Packaging Your First Java Application (IntelliJ Idea)
2. Use the following code for a file named `StreamingJob.java`.

```
package com.amazonaws.services.kinesisanalytics;

import com.amazonaws.services.kinesisanalytics.runtime.KinesisAnalyticsRuntime;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisConsumer;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisProducer;
import
 org.apache.flink.streaming.connectors.kinesis.config.ConsumerConfigConstants;

import java.io.IOException;
import java.util.Map;
import java.util.Properties;

public class StreamingJob {

    private static final String region = "us-east-1";
    private static final String inputStreamName = "ExampleInputStream";
    private static final String outputStreamName = "ExampleOutputStream";

    private static DataStream<String>
 createSourceFromStaticConfig(StreamExecutionEnvironment env) {
        Properties inputProperties = new Properties();
```

```
        inputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);

inputProperties.setProperty(ConsumerConfigConstants.STREAM_INITIAL_POSITION,
"LATEST");

        return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(), inputProperties));
    }

    private static DataStream<String>
createSourceFromApplicationProperties(StreamExecutionEnvironment env)
            throws IOException {
        Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
        return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(),
                applicationProperties.get("ConsumerConfigProperties")));
    }

    private static FlinkKinesisProducer<String> createSinkFromStaticConfig() {
        Properties outputProperties = new Properties();
        outputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
        outputProperties.setProperty("AggregationEnabled", "false");

        FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(), outputProperties);
        sink.setDefaultStream(outputStreamName);
        sink.setDefaultPartition("0");
        return sink;
    }

    private static FlinkKinesisProducer<String>
createSinkFromApplicationProperties() throws IOException {
        Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
        FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(),
                applicationProperties.get("ProducerConfigProperties"));

        sink.setDefaultStream(outputStreamName);
        sink.setDefaultPartition("0");
        return sink;
    }
```

```
    public static void main(String[] args) throws Exception {
        // set up the streaming execution environment
        final StreamExecutionEnvironment env =
 StreamExecutionEnvironment.getExecutionEnvironment();

        /*
         * if you would like to use runtime configuration properties, uncomment the
         * lines below
         * DataStream<String> input = createSourceFromApplicationProperties(env);
         */

        DataStream<String> input = createSourceFromStaticConfig(env);

        /*
         * if you would like to use runtime configuration properties, uncomment the
         * lines below
         * input.addSink(createSinkFromApplicationProperties())
         */

        input.addSink(createSinkFromStaticConfig());

        env.execute("Flink Streaming Java API Skeleton");
    }
}
```

Note the following about the preceding code example:

- This file contains the `main` method that defines the application's functionality.
- Your application creates source and sink connectors to access external resources using a `StreamExecutionEnvironment` object.
- The application creates source and sink connectors using static properties. To use dynamic application properties, use the `createSourceFromApplicationProperties` and `createSinkFromApplicationProperties` methods to create the connectors. These methods read the application's properties to configure the connectors.

3. To use your application code, you compile and package it into a JAR file. You can compile and package your code in one of two ways:

   - Use the command line Maven tool. Create your JAR file by running the following command in the directory that contains the `pom.xml` file:

```
mvn package
```

- Use your development environment. See your development environment documentation for details.

You can either upload your package as a JAR file, or you can compress your package and upload it as a ZIP file. If you create your application using the AWS CLI, you specify your code content type (JAR or ZIP).

4. If there are errors while compiling, verify that your `JAVA_HOME` environment variable is correctly set.

If the application compiles successfully, the following file is created:

`target/java-getting-started-1.0.jar`

Upload the Apache Flink streaming Java code

In this section, you create an Amazon Simple Storage Service (Amazon S3) bucket and upload your application code.

To upload the application code

1. Open the Amazon S3 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/s3/.
2. Choose **Create bucket**.
3. Enter **ka-app-code-*<username>*** in the **Bucket name** field. Add a suffix to the bucket name, such as your user name, to make it globally unique. Choose **Next**.
4. In the **Configure options** step, keep the settings as they are, and choose **Next**.
5. In the **Set permissions** step, keep the settings as they are, and choose **Next**.
6. Choose **Create bucket**.
7. In the Amazon S3 console, choose the **ka-app-code-*<username>*** bucket, and choose **Upload**.
8. In the **Select files** step, choose **Add files**. Navigate to the `java-getting-started-1.0.jar` file that you created in the previous step. Choose **Next**.
9. In the **Set permissions** step, keep the settings as they are. Choose **Next**.
10. In the **Set properties** step, keep the settings as they are. Choose **Upload**.

Your application code is now stored in an Amazon S3 bucket where your application can access it.

Create and run the Managed Service for Apache Flink application

You can create and run a Managed Service for Apache Flink application using either the console or the AWS CLI.

> **Note**
>
> When you create the application using the console, your AWS Identity and Access Management (IAM) and Amazon CloudWatch Logs resources are created for you. When you create the application using the AWS CLI, you create these resources separately.

Topics

- Create and run the application (Console)
- Create and run the application (AWS CLI)

Create and run the application (Console)

Follow these steps to create, configure, update, and run the application using the console.

Create the application

1. Open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. On the Amazon Kinesis dashboard, choose **Create analytics application**.
3. On the **Kinesis Analytics - Create application** page, provide the application details as follows:
   - For **Application name**, enter **`MyApplication`**.
   - For **Description**, enter **`My java test app`**.
   - For **Runtime**, choose **Apache Flink 1.6**.
4. For **Access permissions**, choose **Create / update IAM role `kinesis-analytics-MyApplication-us-west-2`**.
5. Choose **Create application**.

Note

When you create an Amazon Managed Service for Apache Flink application using the console, you have the option of having an IAM role and policy created for your application. Your application uses this role and policy to access its dependent resources. These IAM resources are named using your application name and Region as follows:

- Policy: `kinesis-analytics-service-MyApplication-us-west-2`
- Role: `kinesis-analytics-MyApplication-us-west-2`

Edit the IAM policy

Edit the IAM policy to add permissions to access the Kinesis data streams.

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. Choose **Policies**. Choose the **kinesis-analytics-service-MyApplication-us-west-2** policy that the console created for you in the previous section.
3. On the **Summary** page, choose **Edit policy**. Choose the **JSON** tab.
4. Add the highlighted section of the following policy example to the policy. Replace the sample account IDs (*012345678901*) with your account ID.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadCode",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": [
                "arn:aws-eusc:s3:::ka-app-code-username/java-getting-
started-1.0.jar"
            ]
        },
```

```
        {
            "Sid": "ListCloudwatchLogGroups",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups"
            ],
            "Resource": [
                "arn:aws-eusc:logs:us-west-2:012345678901:log-group:*"
            ]
        },
        {
            "Sid": "ListCloudwatchLogStreams",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws-eusc:logs:us-west-2:012345678901:log-group:/aws/
kinesis-analytics/MyApplication:log-stream:*"
            ]
        },
        {
            "Sid": "PutCloudwatchLogs",
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws-eusc:logs:us-west-2:012345678901:log-group:/aws/
kinesis-analytics/MyApplication:log-stream:kinesis-analytics-log-stream"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws-eusc:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
```

```
            "Resource": "arn:aws-eusc:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
        }
    ]
}
```

Configure the application

1. On the **MyApplication** page, choose **Configure**.
2. On the **Configure application** page, provide the **Code location**:
   - For **Amazon S3 bucket**, enter **`ka-app-code-<username>`**.
   - For **Path to Amazon S3 object**, enter **`java-getting-started-1.0.jar`**.
3. Under **Access to application resources**, for **Access permissions**, choose **Create / update IAM role `kinesis-analytics-MyApplication-us-west-2`**.
4. Under **Properties**, for **Group ID**, enter **`ProducerConfigProperties`**.
5. Enter the following application properties and values:

Key	Value
`flink.inputstream.initpos`	`LATEST`
`aws:region`	`us-west-2`
`AggregationEnabled`	`false`

6. Under **Monitoring**, ensure that the **Monitoring metrics level** is set to **Application**.
7. For **CloudWatch logging**, select the **Enable** check box.
8. Choose **Update**.

> **Note**
>
> When you choose to enable CloudWatch logging, Managed Service for Apache Flink creates a log group and log stream for you. The names of these resources are as follows:
>
> - Log group: `/aws/kinesis-analytics/MyApplication`

- Log stream: `kinesis-analytics-log-stream`

Run the application

1. On the **MyApplication** page, choose **Run**. Confirm the action.
2. When the application is running, refresh the page. The console shows the **Application graph**.

Stop the application

On the **MyApplication** page, choose **Stop**. Confirm the action.

Update the application

Using the console, you can update application settings such as application properties, monitoring settings, and the location or file name of the application JAR. You can also reload the application JAR from the Amazon S3 bucket if you need to update the application code.

On the **MyApplication** page, choose **Configure**. Update the application settings and choose **Update**.

Create and run the application (AWS CLI)

In this section, you use the AWS CLI to create and run the Managed Service for Apache Flink application. Managed Service for Apache Flink uses the `kinesisanalyticsv2` AWS CLI command to create and interact with Managed Service for Apache Flink applications.

Create a Permissions Policy

First, you create a permissions policy with two statements: one that grants permissions for the `read` action on the source stream, and another that grants permissions for `write` actions on the sink stream. You then attach the policy to an IAM role (which you create in the next section). Thus, when Managed Service for Apache Flink assumes the role, the service has the necessary permissions to read from the source stream and write to the sink stream.

Use the following code to create the `KAReadSourceStreamWriteSinkStream` permissions policy. Replace *`username`* with the user name that you used to create the Amazon S3 bucket to store the application code. Replace the account ID in the Amazon Resource Names (ARNs) (*`012345678901`*) with your account ID.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "S3",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": ["arn:aws-eusc:s3:::ka-app-code-username",
                "arn:aws-eusc:s3:::ka-app-code-username/*"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws-eusc:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws-eusc:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
        }
    ]
}
```

For step-by-step instructions to create a permissions policy, see Tutorial: Create and Attach Your First Customer Managed Policy in the *IAM User Guide*.

> **Note**
>
> To access other AWS services, you can use the AWS SDK for Java. Managed Service for Apache Flink automatically sets the credentials required by the SDK to those of the service execution IAM role that is associated with your application. No additional steps are needed.

Create an IAM Role

In this section, you create an IAM role that Managed Service for Apache Flink can assume to read a source stream and write to the sink stream.

Managed Service for Apache Flink cannot access your stream without permissions. You grant these permissions via an IAM role. Each IAM role has two policies attached. The trust policy grants Managed Service for Apache Flink permission to assume the role, and the permissions policy determines what Managed Service for Apache Flink can do after assuming the role.

You attach the permissions policy that you created in the preceding section to this role.

To create an IAM role

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. In the navigation pane, choose **Roles**, **Create Role**.
3. Under **Select type of trusted identity**, choose **AWS Service**. Under **Choose the service that will use this role**, choose **Kinesis**. Under **Select your use case**, choose **Kinesis Analytics**.

   Choose **Next: Permissions**.
4. On the **Attach permissions policies** page, choose **Next: Review**. You attach permissions policies after you create the role.
5. On the **Create role** page, enter **`KA-stream-rw-role`** for the **Role name**. Choose **Create role**.

   Now you have created a new IAM role called `KA-stream-rw-role`. Next, you update the trust and permissions policies for the role.
6. Attach the permissions policy to the role.

   > **Note**
   >
   > For this exercise, Managed Service for Apache Flink assumes this role for both reading data from a Kinesis data stream (source) and writing output to another Kinesis data

stream. So you attach the policy that you created in the previous step, the section called "Create a Permissions Policy".

a. On the **Summary** page, choose the **Permissions** tab.

b. Choose **Attach Policies**.

c. In the search box, enter **KAReadSourceStreamWriteSinkStream** (the policy that you created in the previous section).

d. Choose the **KAReadInputStreamWriteOutputStream** policy, and choose **Attach policy**.

You now have created the service execution role that your application uses to access resources. Make a note of the ARN of the new role.

For step-by-step instructions for creating a role, see Creating an IAM Role (Console) in the *IAM User Guide*.

Create the Managed Service for Apache Flink Application

1. Save the following JSON code to a file named `create_request.json`. Replace the sample role ARN with the ARN for the role that you created previously. Replace the bucket ARN suffix (*username*) with the suffix that you chose in the previous section. Replace the sample account ID (*012345678901*) in the service execution role with your account ID.

```
{
    "ApplicationName": "test",
    "ApplicationDescription": "my java test app",
    "RuntimeEnvironment": "FLINK-1_6",
    "ServiceExecutionRole": "arn:aws:iam::012345678901:role/KA-stream-rw-role",
    "ApplicationConfiguration": {
        "ApplicationCodeConfiguration": {
            "CodeContent": {
                "S3ContentLocation": {
                    "BucketARN": "arn:aws:s3:::ka-app-code-username",
                    "FileKey": "java-getting-started-1.0.jar"
                }
            },
            "CodeContentType": "ZIPFILE"
        },
        "EnvironmentProperties":  {
         "PropertyGroups": [
```

```
            {
               "PropertyGroupId": "ProducerConfigProperties",
               "PropertyMap" : {
                    "flink.stream.initpos" : "LATEST",
                    "aws.region" : "us-west-2",
                    "AggregationEnabled" : "false"
               }
            },
            {
               "PropertyGroupId": "ConsumerConfigProperties",
               "PropertyMap" : {
                    "aws.region" : "us-west-2"
               }
            }
         ]
      }
   }
}
```

2. Execute the `CreateApplication` action with the preceding request to create the application:

```
aws kinesisanalyticsv2 create-application --cli-input-json file://
create_request.json
```

The application is now created. You start the application in the next step.

Start the Application

In this section, you use the `StartApplication` action to start the application.

To start the application

1. Save the following JSON code to a file named `start_request.json`.

```
{
    "ApplicationName": "test",
    "RunConfiguration": {
        "ApplicationRestoreConfiguration": {
         "ApplicationRestoreType": "RESTORE_FROM_LATEST_SNAPSHOT"
         }
    }
```

```
}
```

2. Execute the StartApplication action with the preceding request to start the application:

```
aws kinesisanalyticsv2 start-application --cli-input-json file://start_request.json
```

The application is now running. You can check the Managed Service for Apache Flink metrics on the Amazon CloudWatch console to verify that the application is working.

Stop the Application

In this section, you use the StopApplication action to stop the application.

To stop the application

1. Save the following JSON code to a file named `stop_request.json`.

```
{"ApplicationName": "test"
}
```

2. Execute the StopApplication action with the following request to stop the application:

```
aws kinesisanalyticsv2 stop-application --cli-input-json file://stop_request.json
```

The application is now stopped.

Tutorial: Use AWS Lambda with Amazon Kinesis Data Streams

In this tutorial, you create a Lambda function to consume events from a Kinesis data stream. In this example scenario, a custom application writes records to a Kinesis data stream. AWS Lambda then polls this data stream and, when it detects new data records, invokes your Lambda function. AWS Lambda then executes the Lambda function by assuming the execution role that you specified when you created the Lambda function.

For the detailed step by step instructions, see Tutorial: Using AWS Lambda with Amazon Kinesis.

> **Note**
>
> This tutorial assumes that you have some knowledge of basic Lambda operations and the AWS Lambda console. If you haven't already, follow the instructions in Getting Started with AWS Lambda to create your first Lambda function.

Use the AWS Streaming Data Solution for Amazon Kinesis

The AWS Streaming Data Solution for Amazon Kinesis automatically configures the AWS services necessary to easily capture, store, process, and deliver streaming data. The solution provides multiple options for solving streaming data use cases that use multiple AWS services including Kinesis Data Streams, AWS Lambda, Amazon API Gateway, and Amazon Managed Service for Apache Flink.

Each solution includes the following components:

- A CloudFormation package to deploy the complete example.
- A CloudWatch dashboard for displaying application metrics.
- CloudWatch alarms on the most relevant application metrics.
- All necessary IAM roles and policies.

The solution can be found here: Streaming Data Solution for Amazon Kinesis

Create and manage Kinesis data streams

Amazon Kinesis Data Streams ingests a large amount of data in real time, durably stores the data, and makes the data available for consumption. The unit of data stored by Kinesis Data Streams is a *data record*. A *data stream* represents a group of data records. The data records in a data stream are distributed into shards.

A *shard* has a sequence of data records in a stream. It serves as a base throughput unit of a Kinesis data stream. A shard supports 1 MB/s and 1000 records per second for *writes* and 2 MB/s for *reads* in both on-demand and provisioned capacity modes. The shard limits ensure predictable performance, making it easier to design and operate a highly reliable data streaming workflow.

In this section, you learn how to set the capacity mode for the stream and how to create a stream using either the AWS Management Console or APIs. Then you can take additional actions on the stream.

Topics

- Choose the right mode to stream in
- Create a stream using the AWS Management Console
- Create a stream using the APIs
- Update a stream
- List streams
- List shards
- Delete a stream
- Reshard a stream
- Change the data retention period
- Tag your Amazon Kinesis Data Streams resources
- Handle large records
- Perform resilience testing with AWS Fault Injection Service

Choose the right mode to stream in

The following topics explain how to choose the best mode for your application and how to switch between modes, if needed.

Topics

- What are the different modes in Kinesis Data Streams?
- On-demand Standard mode features and use cases
- On-demand Advantage mode features and use cases
- Provisioned mode features and use cases
- Switch between modes

What are the different modes in Kinesis Data Streams?

A mode determines how the capacity of a data stream is managed and how you're charged for the usage of your data stream. In Amazon Kinesis Data Streams, you can choose **On-demand Standard**, **On-demand Advantage**, and **Provisioned** as the mode for your data streams.

- **On-demand Standard** – Data streams with an on-demand mode require no capacity planning and automatically scale to handle gigabytes of write and read throughput per minute. With the on-demand mode, Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput.
- **On-demand Advantage** – An account-level mode that gives you more capabilities and provides a simpler pricing structure for on-demand streams. In this mode, you can proactively warm a stream's write throughput capacity at any time. You can also trigger a scale-down if a transient data burst has caused the stream to scale significantly beyond its warm capacity. For pricing, there's no longer a fixed charge for each stream. Data ingest, data retrieval, and extended retention usage across all on-demand streams are at least 60% lower than in On-demand Standard.
- **Provisioned** - For the data streams with a provisioned mode, you must specify the number of shards for the data stream. The total capacity of a data stream is the sum of the capacities of its shards. You can increase or decrease the number of shards in a data stream, as needed.

You can use Kinesis Data Streams `PutRecord` and `PutRecords` APIs to write data into your data streams in any mode. To retrieve data, all three modes support default consumers that use the `GetRecords` API and Enhanced Fan-Out (EFO) consumers that use the `SubscribeToShard` API.

All Kinesis Data Streams capabilities, including retention mode, encryption, monitoring metrics, and others, are supported for both the on-demand and provisioned modes. Kinesis Data Streams

provides the high durability and availability in both the on-demand and provisioned capacity modes.

On-demand Standard mode features and use cases

Data streams in the on-demand mode require no capacity planning and automatically scale to handle gigabytes of write and read throughput per minute. On-demand mode simplifies ingesting and storing large data volumes at a low-latency because it eliminates provisioning and managing servers, storage, or throughput. You can ingest billions of records per day without any operational overhead.

On-demand mode is ideal for addressing the needs of highly variable and unpredictable application traffic. You no longer have to provision these workloads for peak capacity, which can result in higher costs due to low utilization. On-demand mode is suited for workloads with unpredictable and highly-variable traffic patterns.

With the on-demand capacity mode, you pay per GB of data written and read from your data streams. You do not need to specify how much read and write throughput you expect your application to perform. Kinesis Data Streams instantly accommodates your workloads as they ramp up or down. For more information, see Amazon Kinesis Data Streams pricing.

A data stream in the on-demand mode accommodates up to double the peak write throughput observed in the previous 30 days. As your data stream's write throughput reaches a new peak, Kinesis Data Streams scales the data stream's capacity automatically. For example, if your data stream has a write throughput that varies between 10 MB/s and 40 MB/s, then Kinesis Data Streams ensures that you can easily burst to double your previous peak throughput, or 80 MB/s. If the same data stream sustains a new peak throughput of 50 MB/s, Kinesis Data Streams ensures that there is enough capacity to ingest 100 MB/s of write throughput. However, write throttling can occur if your traffic increases to more than double the previous peak within a 15-minute duration. You need to retry these throttled requests.

The aggregate read capacity of a data stream with the on-demand mode increases proportionally to write throughput. This helps to ensure that consumer applications always have adequate read throughput to process incoming data in real time. You get at least twice the write throughput compared to read data using the `GetRecords` API. We recommend that you use one consumer application with the `GetRecord` API, so that it has enough room to catch up when the application needs to recover from downtime. It is recommended that you use the Enhanced Fan-Out capability of Kinesis Data Streams for scenarios that require adding more than one consumer application.

Enhanced Fan-Out supports adding up to 20 consumer applications to a data stream using the `SubscribeToShard` API, with each consumer application having dedicated throughput.

Handle read and write throughput exceptions

With the on-demand mode (same as with the provisioned capacity mode), you must specify a partition key with each record to write data into your data stream. Kinesis Data Streams uses your partition keys to distribute data across shards. Kinesis Data Streams monitors traffic for each shard. When the incoming traffic exceeds 500 KB/s per shard, it splits the shard within 15 minutes. The parent shard's hash key values are redistributed evenly across child shards.

If your incoming traffic exceeds twice your prior peak, you can experience read or write exceptions for about 15 minutes, even when your data is distributed evenly across the shards. We recommend that you retry all such requests so that all the records are properly stored in Kinesis Data Streams.

You may experience read and write exceptions if you are using a partition key that leads to uneven data distribution, and the records assigned to a particular shard exceed its limits. With on-demand mode, the data stream automatically adapts to handle uneven data distribution patterns unless a single partition key exceeds a shard's 1 MB/s throughput and 1000 records per second limits.

In the on-demand mode, Kinesis Data Streams splits the shards evenly when it detects an increase in traffic. However, it does not detect and isolate hash keys that are driving a higher portion of incoming traffic to a particular shard. If you are using highly uneven partition keys you may continue to receive write exceptions. For such use cases, we recommend that you use the provisioned capacity mode that supports granular shard splits.

On-demand Advantage mode features and use cases

On-demand Advantage is an account-level setting that unlocks more capabilities and provides a different pricing structure for all on-demand streams in the Region. In this mode, on-demand streams retain their functionality and continue to automatically scale capacity based on actual data usage. If you want to proactively warm a stream's write throughput capacity, you can configure warm throughput. For example, if your data stream has a write throughput between 10 MB/s and 40 MB/s, you can expect it to handle up to 80MB/s of instant throughput increases without throttling. However, if you forecast an upcoming event to peak around 200MB/s of traffic, you can configure the stream with a warm throughput of 200MB/s to ensure that the capacity is available when the data throughput arrives. Using warm throughput does not incur any additional cost. If a transient data burst has caused the stream to scale beyond its warm capacity, you can trigger a

scale-down. To do so, reconfigure the stream's warm throughput to the same or a lower value. The stream capacity decreases to the requested number or the amount needed to support peak data ingest usage within the last hour, whichever is higher.

Another benefit of On-demand Advantage mode is that on-demand streams transition to a simpler pricing structure. With the mode enabled, you no longer see fixed, per-stream charges, and you deal only with data ingest, data retrieval, and the optional extended retention charges. Each pricing dimension is also at a significant discount compared to the corresponding dimension in On-demand Standard. For more information, see Amazon Kinesis Data Streams pricing.

Enhanced fan-out data retrievals also do not have a price premium compared to the standard data retrievals in this mode. Additionally, with On-demand Advantage mode, you can register up to 50 consumers per stream to use enhanced fan-out. Enabling On-demand Advantage commits the account to at least 25MiB/s of data ingest and 25MiB/s of data retrieval across all on-demand streams. For accounts that meet the minimum usage requirement, the Kinesis Data Streams console has a check to see if your account's usage patterns would be a good fit to use On-demand Advantage mode.

If your account's data usage is below the requirement, you'll be charged the difference as a shortfall, but it will still be at the same discounted rate. Enabling On-demand Advantage also has a minimum period of 24 hours before you can disable the mode. Overall, On-demand Advantage is the best way to stream with Kinesis Data Streams if you have consistent throughput usage close to or above the minimum commitment, need many fan-out consumers, or operate with hundreds of data streams.

Provisioned mode features and use cases

With provisioned mode, after you create the data stream, you can dynamically scale your shard capacity up or down using the AWS Management Console or the UpdateShardCount API. You can make updates while there is a Kinesis Data Streams producer or consumer application writing to or reading data from the stream.

The provisioned mode is suited for predictable traffic with capacity requirements that are easy to forecast. You can use the provisioned mode if you want fine-grained control over how data is distributed across shards.

With the provisioned mode, you must specify the number of shards for the data stream. To determine the size of a data stream with the provisioned mode, you need the following input values:

- The average size of the data record written to the stream in kilobytes (KB), rounded up to the nearest 1 KB (`average_data_size_in_KB`).
- The number of data records written to and read from the stream per second (`records_per_second`).
- The number of consumers, which are Kinesis Data Streams applications that consume data concurrently and independently from the stream (`number_of_consumers`).
- The incoming write bandwidth in KB (`incoming_write_bandwidth_in_KB`), which is equal to the `average_data_size_in_KB` multiplied by the `records_per_second`.
- The outgoing read bandwidth in KB (`outgoing_read_bandwidth_in_KB`), which is equal to the `incoming_write_bandwidth_in_KB` multiplied by the `number_of_consumers`.

You can calculate the number of shards (`number_of_shards`) that your stream needs by using the input values in the following formula.

```
number_of_shards = ceiling(max(incoming_write_bandwidth_in_KiB/1024,
 outgoing_read_bandwidth_in_KiB/2048))
```

You may still experience read and write throughput exceptions in the provisioned mode if you don't configure your data stream to handle your peak throughput. In this case, you must manually scale your data stream to accommodate your data traffic.

You may also experience read and write exceptions if you're using a partition key that leads to uneven data distribution and the records assigned to a shard exceed its limits. To resolve this issue in the provisioned mode, identify such shards and manually split them to better accommodate your traffic. For more information, see Resharding a Stream.

Switch between modes

For each data stream in your AWS account, you can switch between the on-demand and provisioned modes twice within 24 hours. Switching between modes doesn't cause any disruptions to your applications that use this data stream. You can continue writing to and reading from this data stream. As you're switching between modes, either from on-demand to provisioned or from provisioned to on-demand, the status of the stream is set to *Updating*. You must wait for the data stream status to get to *Active* before you can modify its properties again.

When you switch from provisioned to on-demand capacity mode, your data stream initially retains whatever shard count it had before the transition, and from this point on, Kinesis Data Streams

monitors your data traffic and scales the shard count of this on-demand data stream depending on your write throughput. When you switch from on-demand to provisioned mode, your data stream also initially retains whatever shard count it had before the transition, but from this point on, you're responsible for monitoring and adjusting the shard count of this data stream to properly accomodate your write throughput.

You can switch from On-demand Standard to On-demand Advantage mode by enabling an account-level setting. When enabled, the account commits to at least 25MiB/s of data ingest and 25MiB/s of data retrieval usage across all on-demand streams in the Region. After you enable it, you must wait at least 24 hours before you can disable On-demand Advantage, but you can request the change at any time. If you want to switch from On-demand Advantage to On-demand Standard, you must first remove any warm throughput configured on on-demand streams.

Create a stream using the AWS Management Console

You can create a stream using the Kinesis Data Streams console, the Kinesis Data Streams API, or the AWS Command Line Interface (AWS CLI).

To create a data stream using the console

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. In the navigation bar, expand the Region selector and choose a Region.
3. Choose **Create data stream**.
4. On the **Create Kinesis stream** page, enter a name for your data stream and then choose either the **On-demand** or **Provisioned** capacity mode. The **On-demand** mode is selected by default. For more information, see Choose the right mode to stream in.

   With the **On-demand** mode, you can then choose **Create Kinesis stream** to create your data stream. With the **Provisioned** mode, you must then specify the number of shards you need, and then choose **Create Kinesis stream**.

   On the **Kinesis streams** page, your stream's **Status** is **Creating** while the stream is being created. When the stream is ready to use, the **Status** changes to **Active**.
5. Choose the name of your stream. The **Stream Details** page displays a summary of your stream configuration, along with monitoring information.

To create a stream using the Kinesis Data Streams API

- For information about creating a stream using the Kinesis Data Streams API, see Create a stream using the APIs.

To create a stream using the AWS CLI

- For information about creating a stream using the AWS CLI, see the create-stream command.

Create a stream using the APIs

Use the following steps to create your Kinesis data stream.

Build the Kinesis Data Streams client

Before you can work with Kinesis data streams, you must build a client object. The following Java code instantiates a client builder and uses it to set the Region, credentials, and the client configuration. It then builds a client object.

```
AmazonKinesisClientBuilder clientBuilder = AmazonKinesisClientBuilder.standard();

clientBuilder.setRegion(regionName);
clientBuilder.setCredentials(credentialsProvider);
clientBuilder.setClientConfiguration(config);

AmazonKinesis client = clientBuilder.build();
```

For more information, see Kinesis Data Streams Regions and Endpoints in the *AWS General Reference*.

Create the stream

Now that you have created your Kinesis Data Streams client, you can create a stream using the console or programmatically. To create a stream programmatically, instantiate a `CreateStreamRequest` object and specify a name for the stream. If you want to use provisioned mode, specify the number of shards for the stream to use.

- **On-demand**:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
```

- **Provisioned**:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
createStreamRequest.setShardCount( myStreamSize );
```

The stream name identifies the stream. The name is scoped to the AWS account used by the application. It is also scoped by Region. That is, two streams in two different AWS accounts can have the same name, and two streams in the same AWS account but in two different Regions can have the same name, but not two streams on the same account and in the same Region.

The throughput of the stream is a function of the number of shards. For greater provisioned throughput, you require more shards. More shards also increase the cost that AWS charges for the stream. For more information about calculating an appropriate number of shards for your application, see Choose the right mode to stream in.

After you have configured the `createStreamRequest` object, create a stream by calling the `createStream` method on the client. After calling `createStream`, wait for the stream to reach the `ACTIVE` state before performing any operations on the stream. To check the state of the stream, call the `describeStream` method. However, `describeStream` throws an exception if the stream does not exist. Therefore, enclose the `describeStream` call in a `try/catch` block.

```
client.createStream( createStreamRequest );
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime ) {
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
```

```
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus =
 describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
    //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

Update a stream

You can update the details of a stream using the Kinesis Data Streams console, the Kinesis Data Streams API, or the AWS CLI.

> **Note**
>
> You can enable server-side encryption for existing streams, or for streams that you have recently created.

Use the console

To update a data stream using the console

1. Open the Amazon Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis/.
2. In the navigation bar, expand the Region selector and choose a Region.

3. Choose the name of your stream in the list. The **Stream Details** page displays a summary of your stream configuration and monitoring information.
4. To switch between on-demand and provisioned capacity modes for a data stream, choose **Edit capacity mode** in the **Configuration** tab. For more information, see Choose the right mode to stream in.

   > **Important**
   >
   > For each data stream in your AWS account, you can switch between the on-demand and provisioned modes twice within 24 hours.

5. For a data stream in on-demand mode with On-demand Advantage mode enabled, you can configure warm throughput to increase or decrease the stream's ingest capacity.

   To increase ingest capacity

   In the **Configuration** tab, choose **Edit warm throughput** and enter a target throughput value (in MB/s) that is higher than the stream's current capacity. The stream immediately provisions capacity to at least the specified warm throughput, ensuring it can handle the expected traffic without throttling. For example, if you forecast an upcoming event to peak around 200 MB/s, set the warm throughput to 200 MB/s ahead of time at no additional cost.

   To decrease ingest capacity

   If the stream has scaled significantly beyond its warm throughput value, reconfigure the warm throughput to that value or lower to trigger a scale-down. The stream scales to the requested number or the capacity needed to support peak data ingest usage observed within the last hour, whichever is higher. This ensures the stream always retains enough capacity to handle recent traffic patterns.

   Using warm throughput does not incur any additional cost. For more information, see On-demand Advantage mode features and use cases.
6. For a data stream with the provisioned mode, to edit the number of shards, choose **Edit provisioned shards** in the **Configuration** tab, and then enter a new shard count.
7. To enable server-side encryption of data records, choose **Edit** in the **Server-side encryption** section. Choose a KMS key to use as the master key for encryption, or use the default master key, **aws/kinesis**, managed by Kinesis. If you enable encryption for a stream and use your own AWS KMS master key, ensure that your producer and consumer applications have access to the

AWS KMS master key that you used. To assign permissions to an application to access a user-generated AWS KMS key, see the section called "Permissions to use user-generated KMS keys".

8. To edit the data retention period, choose **Edit** in the **Data retention period** section, and then enter a new data retention period.
9. If you have enabled custom metrics on your account, choose **Edit** in the **Shard level metrics** section, and then specify metrics for your stream. For more information, see the section called "Monitor the Kinesis Data Streams service with CloudWatch".

Use the API

To update stream details using the API, see the following methods:

- AddTagsToStream
- DecreaseStreamRetentionPeriod
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- IncreaseStreamRetentionPeriod
- RemoveTagsFromStream
- StartStreamEncryption
- StopStreamEncryption
- UpdateShardCount
- UpdateStreamMode
- UpdateStreamWarmThroughput — When On-demand Advantage is enabled for an account, use this API to configure warm throughput for an on-demand stream. When you switch a stream from Provisioned to On-demand mode using the UpdateStreamMode API, you can also set warm throughput.

Use the AWS CLI

For information about updating a stream using the AWS CLI, see the Kinesis CLI reference.

To configure warm throughput using the AWS CLI, use the `update-stream-warm-throughput` command. For example:

```
aws kinesis update-stream-warm-throughput \
```

```
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-stream \
    --warm-throughput-mi-bps=200
```

To lower capacity, reconfigure `warm-throughput-mi-bps` to the same or a lower value:

```
aws kinesis update-stream-warm-throughput \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-stream \
    --warm-throughput-mi-bps=50
```

The stream scales to the requested warm throughput or the capacity needed to support recent peak data ingest usage, whichever is higher.

List streams

Streams are scoped to the AWS account associated with the AWS credentials used to instantiate the Kinesis Data Streams client and also to the Region specified for the client. An AWS account could have many streams active at one time. You can list your streams in the Kinesis Data Streams console, or programmatically. The code in this section shows how to list all the streams for your AWS account.

```
ListStreamsRequest listStreamsRequest = new ListStreamsRequest();
listStreamsRequest.setLimit(20);
ListStreamsResult listStreamsResult = client.listStreams(listStreamsRequest);
List<String> streamNames = listStreamsResult.getStreamNames();
```

This code example first creates a new instance of `ListStreamsRequest` and calls its `setLimit` method to specify that a maximum of 20 streams should be returned for each call to `listStreams`. If you do not specify a value for `setLimit`, Kinesis Data Streams returns a number of streams less than or equal to the number in the account. The code then passes `listStreamsRequest` to the `listStreams` method of the client. The return value `listStreams` is stored in a `ListStreamsResult` object. The code calls the `getStreamNames` method on this object and stores the returned stream names in the `streamNames` list. Note that Kinesis Data Streams might return fewer streams than specified by the specified limit even if there are more streams than that in the account and Region. To ensure that you retrieve all the streams, use the `getHasMoreStreams` method as described in the next code example.

```
while (listStreamsResult.getHasMoreStreams())
{
```

```
    if (streamNames.size() > 0) {
      listStreamsRequest.setExclusiveStartStreamName(streamNames.get(streamNames.size()
 - 1));
    }
    listStreamsResult = client.listStreams(listStreamsRequest);
    streamNames.addAll(listStreamsResult.getStreamNames());
}
```

This code calls the `getHasMoreStreams` method on `listStreamsRequest` to check if there are additional streams available beyond the ones returned in the initial call to `listStreams`. If so, the code calls the `setExclusiveStartStreamName` method with the name of the last stream that was returned in the previous call to `listStreams`. The `setExclusiveStartStreamName` method causes the next call to `listStreams` to start after that stream. The group of stream names returned by that call is then added to the `streamNames` list. This process continues until all the stream names have been collected in the list.

The streams returned by `listStreams` can be in one of the following states:

- `CREATING`
- `ACTIVE`
- `UPDATING`
- `DELETING`

You can check the state of a stream using the `describeStream` method, as shown in the previous section, Create a stream using the APIs.

List shards

A data stream can have one or more shards. The recommended method for listing or retrieving the shards from a data stream is to use the ListShards API. The following example shows how you can get a list of the shards in a data stream. For a full description of the main operation used in this example and all of the parameters you can set for the operation, see ListShards.

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ListShardsRequest;
import software.amazon.awssdk.services.kinesis.model.ListShardsResponse;
```

```
import java.util.concurrent.TimeUnit;

public class ShardSample {

    public static void main(String[] args) {

        KinesisAsyncClient client = KinesisAsyncClient.builder().build();

        ListShardsRequest request = ListShardsRequest
                .builder().streamName("myFirstStream")
                .build();

        try {
            ListShardsResponse response = client.listShards(request).get(5000,
 TimeUnit.MILLISECONDS);
            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

To run the previous code example you can use a POM file like the following one.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>kinesis.data.streams.samples</groupId>
    <artifactId>shards</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
```

```
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>kinesis</artifactId>
            <version>2.0.0</version>
        </dependency>
    </dependencies>
</project>
```

With the `ListShards` API, you can use the [ShardFilter] parameter to filter out the response of the API. You can only specify one filter at a time.

If you use the `ShardFilter` parameter when invoking the ListShards API, the `Type` is the required property and must be specified. If you specify the `AT_TRIM_HORIZON`, `FROM_TRIM_HORIZON`, or `AT_LATEST` types, you do not need to specify either the `ShardId` or the `Timestamp` optional properties.

If you specify the `AFTER_SHARD_ID` type, you must also provide the value for the optional `ShardId` property. The `ShardId` property is identical in functionality to the `ExclusiveStartShardId` parameter of the ListShards API. When `ShardId` property is specified, the response includes the shards starting with the shard whose ID immediately follows the `ShardId` that you provided.

If you specify the `AT_TIMESTAMP` or `FROM_TIMESTAMP_ID` type, you must also provide the value for the optional `Timestamp` property. If you specify the `AT_TIMESTAMP` type, then all shards that were open at the provided timestamp are returned. If you specify the `FROM_TIMESTAMP` type, then all shards starting from the provided timestamp to TIP are returned.

Important

`DescribeStreamSummary` and `ListShard` APIs provide a more scalable way to retrieve information about your data streams. More specifically, the quotas for the DescribeStream API can cause throttling. For more information, see Quotas and limits. Note also that `DescribeStream` quotas are shared across all applications that interact with all data streams in your AWS account. The quotas for the ListShards API, on the other hand, are specific to a single data stream. So not only do you get higher TPS with the ListShards API, but the action scales better as you create more data streams.

We recommend that you migrate all of your producers and consumers that call the DescribeStream API to instead invoke the DescribeStreamSummary and the ListShard APIs. To identify these producers and consumers, we recommend using Athena to parse CloudTrail logs as user agents for KPL and KCL are captured in the API calls.

```
SELECT useridentity.sessioncontext.sessionissuer.username,
useridentity.arn,eventname,useragent, count(*) FROM
cloudtrail_logs WHERE Eventname IN ('DescribeStream')  AND
eventtime
    BETWEEN ''
        AND ''
GROUP BY
 useridentity.sessioncontext.sessionissuer.username,useridentity.arn,eventname,useragent
ORDER BY  count(*) DESC LIMIT 100
```

We also recommend that the AWS Lambda and Amazon Firehose integrations with Kinesis Data Streams that invoke the `DescribeStream` API are reconfigured so that the integrations instead invoke `DescribeStreamSummary` and `ListShards`. Specifically, for AWS Lambda, you must update your event source mapping. For Amazon Firehose, the corresponding IAM permissions must be updated so that they include the `ListShards` IAM permission.

Delete a stream

You can delete a stream with the Kinesis Data Streams console, or programmatically. To delete a stream programmatically, use `DeleteStreamRequest`, as shown in the following code.

```
DeleteStreamRequest deleteStreamRequest = new DeleteStreamRequest();
deleteStreamRequest.setStreamName(myStreamName);
client.deleteStream(deleteStreamRequest);
```

Shut down any applications that are operating on the stream before you delete it. If an application attempts to operate on a deleted stream, it receives `ResourceNotFound` exceptions. Also, if you subsequently create a new stream that has the same name as your previous stream, and applications that were operating on the previous stream are still running, these applications might

try to interact with the new stream as though it were the previous stream—with unpredictable results.

Reshard a stream

> **Important**
>
> You can reshard your stream using the UpdateShardCount API. Otherwise, you can continue to perform splits and merges as explained here.

Amazon Kinesis Data Streams supports *resharding*, which lets you adjust the number of shards in your stream to adapt to changes in the rate of data flow through the stream. Resharding is considered an advanced operation. If you are new to Kinesis Data Streams, return to this subject after you are familiar with all the other aspects of Kinesis Data Streams.

There are two types of resharding operations: shard split and shard merge. In a shard split, you divide a single shard into two shards. In a shard merge, you combine two shards into a single shard. Resharding is always *pairwise* in the sense that you cannot split into more than two shards in a single operation, and you cannot merge more than two shards in a single operation. The shard or pair of shards that the resharding operation acts on are referred to as *parent* shards. The shard or pair of shards that result from the resharding operation are referred to as *child* shards.

Splitting increases the number of shards in your stream and therefore increases the data capacity of the stream. Because you are charged on a per-shard basis, splitting increases the cost of your stream. Similarly, merging reduces the number of shards in your stream and therefore decreases the data capacity—and cost—of the stream.

Resharding is typically performed by an administrative application that is distinct from the producer (put) applications and the consumer (get) applications. Such an administrative application monitors the overall performance of the stream based on metrics provided by Amazon CloudWatch or based on metrics collected from the producers and consumers. The administrative application also needs a broader set of IAM permissions than the consumers or producers because the consumers and producers usually should not need access to the APIs used for resharding. For more information about IAM permissions for Kinesis Data Streams, see Controlling access to Amazon Kinesis Data Streams resources using IAM.

For more information about resharding, see How do I change the number of open shards in Kinesis Data Streams?

Topics

- Decide on a strategy for resharding
- Split a shard
- Merge two shards
- Complete the resharding action

Decide on a strategy for resharding

The purpose of resharding in Amazon Kinesis Data Streams is to enable your stream to adapt to changes in the rate of data flow. You split shards to increase the capacity (and cost) of your stream. You merge shards to reduce the cost (and capacity) of your stream.

One approach to resharding could be to split every shard in the stream—which would double the stream's capacity. However, this might provide more additional capacity than you actually need and therefore create unnecessary cost.

You can also use metrics to determine which are your *hot* or *cold* shards, that is, shards that are receiving much more data, or much less data, than expected. You could then selectively split the hot shards to increase capacity for the hash keys that target those shards. Similarly, you could merge cold shards to make better use of their unused capacity.

You can obtain some performance data for your stream from the Amazon CloudWatch metrics that Kinesis Data Streams publishes. However, you can also collect some of your own metrics for your streams. One approach would be to log the hash key values generated by the partition keys for your data records. Recall that you specify the partition key at the time that you add the record to the stream.

```
putRecordRequest.setPartitionKey( String.format( "myPartitionKey" ) );
```

Kinesis Data Streams uses MD5 to compute the hash key from the partition key. Because you specify the partition key for the record, you could use MD5 to compute the hash key value for that record and log it.

You could also log the IDs of the shards that your data records are assigned to. The shard ID is available by using the `getShardId` method of the `putRecordResults` object returned by the `putRecords` method, and the `putRecordResult` object returned by the `putRecord` method.

```
String shardId = putRecordResult.getShardId();
```

With the shard IDs and the hash key values, you can determine which shards and hash keys are receiving the most or least traffic. You can then use resharding to provide more or less capacity, as appropriate for these keys.

Split a shard

To split a shard in Amazon Kinesis Data Streams, you need to specify how hash key values from the parent shard should be redistributed to the child shards. When you add a data record to a stream, it is assigned to a shard based on a hash key value. The hash key value is the MD5 hash of the partition key that you specify for the data record at the time that you add the data record to the stream. Data records that have the same partition key also have the same hash key value.

The possible hash key values for a given shard constitute a set of ordered contiguous non-negative integers. This range of possible hash key values is given by the following:

```
shard.getHashKeyRange().getStartingHashKey();
shard.getHashKeyRange().getEndingHashKey();
```

When you split the shard, you specify a value in this range. That hash key value and all higher hash key values are distributed to one of the child shards. All the lower hash key values are distributed to the other child shard.

The following code demonstrates a shard split operation that redistributes the hash keys evenly between each of the child shards, essentially splitting the parent shard in half. This is just one possible way of dividing the parent shard. You could, for example, split the shard so that the lower one-third of the keys from the parent go to one child shard and the upper two-thirds of the keys go to the other child shard. However, for many applications, splitting shards in half is an effective approach.

The code assumes that `myStreamName` holds the name of your stream and the object variable `shard` holds the shard to split. Begin by instantiating a new `splitShardRequest` object and setting the stream name and shard ID.

```
SplitShardRequest splitShardRequest = new SplitShardRequest();
splitShardRequest.setStreamName(myStreamName);
splitShardRequest.setShardToSplit(shard.getShardId());
```

Determine the hash key value that is half-way between the lowest and highest values in the shard. This is the starting hash key value for the child shard that will contain the upper half of the hash keys from the parent shard. Specify this value in the `setNewStartingHashKey` method. You need specify only this value. Kinesis Data Streams automatically distributes the hash keys below this value to the other child shard that is created by the split. The last step is to call the `splitShard` method on the Kinesis Data Streams client.

```
BigInteger startingHashKey = new
 BigInteger(shard.getHashKeyRange().getStartingHashKey());
BigInteger endingHashKey   = new
 BigInteger(shard.getHashKeyRange().getEndingHashKey());
String newStartingHashKey  = startingHashKey.add(endingHashKey).divide(new
 BigInteger("2")).toString();

splitShardRequest.setNewStartingHashKey(newStartingHashKey);
client.splitShard(splitShardRequest);
```

The first step after this procedure is shown in Wait for a stream to become active again.

Merge two shards

A shard merge operation takes two specified shards and combines them into a single shard. After the merge, the single child shard receives data for all hash key values covered by the two parent shards.

Shard Adjacency

To merge two shards, the shards must be *adjacent*. Two shards are considered adjacent if the union of the hash key ranges for the two shards forms a contiguous set with no gaps. For example, suppose that you have two shards, one with a hash key range of 276...381 and the other with a hash key range of 382...454. You could merge these two shards into a single shard that would have a hash key range of 276...454.

To take another example, suppose that you have two shards, one with a hash key range of 276..381 and the other with a hash key range of 455...560. You could not merge these two shards because there would be one or more shards between these two that cover the range 382..454.

The set of all OPEN shards in a stream—as a group—always spans the entire range of MD5 hash key values. For more information about shard states—such as CLOSED—see Consider data routing, data persistence, and shard state after a reshard.

To identify shards that are candidates for merging, you should filter out all shards that are in a CLOSED state. Shards that are OPEN—that is, not CLOSED—have an ending sequence number of null. You can test the ending sequence number for a shard using:

```
if( null == shard.getSequenceNumberRange().getEndingSequenceNumber() )
{
  // Shard is OPEN, so it is a possible candidate to be merged.
}
```

After filtering out the closed shards, sort the remaining shards by the highest hash key value supported by each shard. You can retrieve this value using:

```
shard.getHashKeyRange().getEndingHashKey();
```

If two shards are adjacent in this filtered, sorted list, they can be merged.

Code for the Merge Operation

The following code merges two shards. The code assumes that myStreamName holds the name of your stream and the object variables shard1 and shard2 hold the two adjacent shards to merge.

For the merge operation, begin by instantiating a new mergeShardsRequest object. Specify the stream name with the setStreamName method. Then specify the two shards to merge using the setShardToMerge and setAdjacentShardToMerge methods. Finally, call the mergeShards method on the Kinesis Data Streams client to carry out the operation.

```
MergeShardsRequest mergeShardsRequest = new MergeShardsRequest();
mergeShardsRequest.setStreamName(myStreamName);
mergeShardsRequest.setShardToMerge(shard1.getShardId());
mergeShardsRequest.setAdjacentShardToMerge(shard2.getShardId());
client.mergeShards(mergeShardsRequest);
```

The first step after this procedure is shown in Wait for a stream to become active again.

Complete the resharding action

After any kind of resharding procedure in Amazon Kinesis Data Streams, and before normal record processing resumes, other procedures and considerations are required. The following sections describe these.

Topics

- Wait for a stream to become active again
- Consider data routing, data persistence, and shard state after a reshard

Wait for a stream to become active again

After you call a resharding operation, either `splitShard` or `mergeShards`, you must wait for the stream to become active again. The code to use is the same as when you wait for a stream to become active after creating a stream. That code is as follows:

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime )
{
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus =
 describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
   //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime )
{
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
```

```
}
```

Consider data routing, data persistence, and shard state after a reshard

Kinesis Data Streams is a real-time data streaming service. Your applications should assume that data is flowing continuously through the shards in your stream. When you reshard, data records that were flowing to the parent shards are re-routed to flow to the child shards based on the hash key values that the data-record partition keys map to. However, any data records that were in the parent shards before the reshard remain in those shards. The parent shards do not disappear when the reshard occurs. They persist along with the data they contained before the reshard. The data records in the parent shards are accessible using the `getShardIterator` and `getRecords` operations in the Kinesis Data Streams API, or through the Kinesis Client Library.

> **Note**
>
> Data records are accessible from the time they are added to the stream to the current retention period. This holds true regardless of any changes to the shards in the stream during that time period. For more information about a stream's retention period, see Change the data retention period.

In the process of resharding, a parent shard transitions from an `OPEN` state to a `CLOSED` state to an `EXPIRED` state.

- **OPEN**: Before a reshard operation, a parent shard is in the `OPEN` state, which means that data records can be both added to the shard and retrieved from the shard.
- **CLOSED**: After a reshard operation, the parent shard transitions to a `CLOSED` state. This means that data records are no longer added to the shard. Data records that would have been added to this shard are now added to a child shard instead. However, data records can still be retrieved from the shard for a limited time.
- **EXPIRED**: After the stream's retention period has expired, all the data records in the parent shard have expired and are no longer accessible. At this point, the shard itself transitions to an `EXPIRED` state. Calls to `getStreamDescription().getShards` to enumerate the shards in the stream do not include `EXPIRED` shards in the list shards returned. For more information about a stream's retention period, see Change the data retention period.

After the reshard has occurred and the stream is again in an `ACTIVE` state, you could immediately begin to read data from the child shards. However, the parent shards that remain after the reshard might still contain data that you haven't read yet that was added to the stream before the reshard. If you read data from the child shards before having read all data from the parent shards, you could read data for a particular hash key out of the order given by the data records' sequence numbers. Therefore, assuming that the order of the data is important, you should, after a reshard, always continue to read data from the parent shards until it is exhausted. Only then should you begin reading data from the child shards. When `getRecordsResult.getNextShardIterator` returns `null`, it indicates that you have read all the data in the parent shard.

Change the data retention period

Amazon Kinesis Data Streams supports changes to the data record retention period of your data stream. A Kinesis data stream is an ordered sequence of data records meant to be written to and read from in real time. Data records are therefore stored in shards in your stream temporarily. The time period from when a record is added to when it is no longer accessible is called the *retention period*. A Kinesis data stream stores records from 24 hours by default, up to 8760 hours (365 days).

You can update the retention period via the Kinesis Data Streams console or by using the IncreaseStreamRetentionPeriod and the DecreaseStreamRetentionPeriod operations. With the Kinesis Data Streams console, you can bulk edit the retention period of more than one data stream at the same time. You can increase the retention period up to a maximum of 8760 hours (365 days) using the IncreaseStreamRetentionPeriod operation or the Kinesis Data Streams console. You can decrease the retention period down to a minimum of 24 hours using the DecreaseStreamRetentionPeriod operation or the Kinesis Data Streams console. The request syntax for both operations includes the stream name and the retention period in hours. Finally, you can check the current retention period of a stream by calling the DescribeStream operation.

The following is an example of changing the retention period using the AWS CLI:

```
aws kinesis increase-stream-retention-period --stream-name retentionPeriodDemo --
retention-period-hours 72
```

Kinesis Data Streams stops making records inaccessible at the old retention period within several minutes of increasing the retention period. For example, changing the retention period from 24 hours to 48 hours means that records added to the stream 23 hours 55 minutes prior are still available after 24 hours.

Kinesis Data Streams almost immediately makes records older than the new retention period inaccessible upon decreasing the retention period. Therefore, take great care when calling the DecreaseStreamRetentionPeriod operation.

Set your data retention period to ensure that your consumers are able to read data before it expires, if problems occur. You should carefully consider all possibilities, such as an issue with your record processing logic or a downstream dependency being down for a long period of time. Think of the retention period as a safety net to allow more time for your data consumers to recover. The retention period API operations allow you to set this up proactively or to respond to operational events reactively.

Additional charges apply for streams with a retention period set above 24 hours. For more information, see Amazon Kinesis Data Streams Pricing.

Tag your Amazon Kinesis Data Streams resources

You can assign your own metadata to streams and enhanced fan-out consumers you create in Amazon Kinesis Data Streams in the form of *tags*. A tag is a key-value pair that you define for a stream. Using tags is a simple yet powerful way to manage AWS resources and organize data, including billing data.

Contents

- Review tag basics
- Track costs using tagging
- Understand tag restrictions
- Tag streams using the Kinesis Data Streams console
- Tag streams using the AWS CLI
- Tag streams using the Kinesis Data Streams APIs
- Tag consumers using the AWS CLI
- Tag consumers using the Kinesis Data Streams APIs

Review tag basics

The Kinesis Data Streams resources you can tag include data streams and enhanced fan-out consumers. You use the Kinesis Data Streams console, AWS CLI, or Kinesis Data Streams API to complete the following tasks:

- Create a resource with tags
- Add tags to a resource
- List the tags for your resources
- Remove tags from a resource

> **Note**
>
> You can't apply tags to enhanced fan-out consumers using the Kinesis Data Streams console. For applying tags to consumers, use AWS CLI or Kinesis Data Streams API.

You can use tags to categorize your resources. For example, you can categorize resources by purpose, owner, or environment. Because you define the key and value for each tag, you can create a custom set of categories to meet your specific needs. For example, you might define a set of tags that helps you track resources by owner and associated application. Here are several examples of tags:

- Project: Project name
- Owner: Name
- Purpose: Load testing
- Application: Application name
- Environment: Production

> **Important**
>
> - To add tags while creating a stream, you must include the `kinesis:CreateStream` and `kinesis:AddTagsToStream` permissions for that stream. You **can't use** the `kinesis:TagResource` permission to tag streams while creating them.
> - To add tags during consumer registration, you must include the `kinesis:TagResource` and `kinesis:RegisterStreamConsumer` permissions.

Track costs using tagging

You can use tags to categorize and track your AWS costs. When you apply tags to your Kinesis Data Streams resources, your AWS cost allocation report includes usage and costs aggregated by tags. You can apply tags that represent business categories, such as cost centers, application names, or owners, to organize your costs across multiple services. For more information, see Use Cost Allocation Tags for Custom Billing Reports in the *AWS Billing User Guide*.

Understand tag restrictions

The following restrictions apply to tags:

Basic restrictions

- The maximum number of tags for each resource is 50.
- Tag keys and values are case-sensitive.
- You can't change or edit tags for a deleted resource.

Tag key restrictions

- Each tag key must be unique. If you add a tag with a key that's already in use, your new tag overwrites the existing key-value pair.
- You can't start a tag key with `aws:` because this prefix is reserved for use by AWS. AWS creates tags that begin with this prefix on your behalf, but you can't edit or delete them.
- Tag keys must be between 1 and 128 Unicode characters in length.
- Tag keys must consist of the following characters: Unicode letters, digits, white space, and the following special characters: `_ . / = + - @`.

Tag value restrictions

- Tag values must be between 0 and 255 Unicode characters in length.
- Tag values can be blank. Otherwise, they must consist of the following characters: Unicode letters, digits, white space, and any of the following special characters: `_ . / = + - @`.

Tag streams using the Kinesis Data Streams console

You can add, update, list, and remove tags on your streams using the Kinesis Data Streams console.

To view the tags for a stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. On the left navigation pane, choose **Data streams**.
3. On the **Data streams** page, choose the stream that you want to tag.
4. On the Stream details page, choose **Configuration**.
5. In the **Tags** section, view the tags applied to the stream.

To create a data stream with a tag

1. Open the Kinesis Data Streams console.
2. On the left navigation pane, choose **Data streams**.
3. Choose **Create data stream**.
4. On the **Create data stream** page, enter a name for your data stream.
5. For **Data stream capacity**, choose either the **On-demand** or the **Provisioned** capacity mode.

   For more information about capacity modes, see Choose the right mode to stream in.
6. In the **Tags** section, do the following:

   a. Choose **Add new tag**.

   b. For **Key**, enter the tag and optionally specify a value in the **Value** field.

      If you see an error, either the tag key or value that you specified doesn't meet the tag restrictions. For more information, see Understand tag restrictions.
7. Choose **Create data stream**.

To add or update a tag on a stream

1. Open the Kinesis Data Streams console.
2. On the left navigation pane, choose **Data streams**.
3. On the **Data streams** page, choose the stream to which you want to add or update tags.
4. On the Stream details page, choose **Configuration**.
5. In the **Tags** section, choose **Manage tags**.

6. Under **Tags**, do one of the following:

   - To add a new tag, choose **Add new tag**, and then enter the tag's **Key** and **Value** data. Repeat this step as many times as necessary.

     The maximum number of tags you can add for each stream is 50.
   - To update an existing tag, enter a new tag value in the **Value** field of that tag's **Key**.

   If you see an error, either the tag key or value that you specified doesn't meet the tag restrictions. For more information, see Understand tag restrictions.
7. Choose **Save changes**.

To remove a tag from a stream

1. Open the Kinesis Data Streams console.
2. On the left navigation pane, choose **Data streams**.
3. On the **Data streams** page, choose the stream from which you want to remove tags.
4. On the Stream details page, choose **Configuration**.
5. In the **Tags** section, choose **Manage tags**.
6. Find the tag **Key** and **Value** pair that you want to remove. Then, choose **Remove**.
7. Choose **Save changes**.

Tag streams using the AWS CLI

You can add, list, and remove tags on your streams using the AWS CLI. For examples, see the following documentation.

create-stream

Creates a stream with tags.

add-tags-to-stream

Adds or updates tags for the specified stream.

list-tags-for-stream

Lists the tags for the specified stream.

remove-tags-from-stream

Removes tags from the specified stream.

Tag streams using the Kinesis Data Streams APIs

You can add, list, and remove tags on your streams using the Kinesis Data Streams APIs. For examples, see the following documentation:

CreateStream

Creates a stream with tags.

AddTagsToStream

Adds or updates tags for the specified stream.

ListTagsForStream

Lists the tags for the specified stream.

RemoveTagsFromStream

Removes tags from the specified stream.

Tag consumers using the AWS CLI

You can add, list, and remove tags on your consumers using the AWS CLI. For examples, see the following documentation:

register-stream-consumer

Registers a consumer for a Kinesis data stream with tags.

tag-resource

Adds or updates tags for the specified Kinesis resource.

list-tags-for-resource

Lists the tags for the specified Kinesis resource.

untag-resource

Removes tags from the specified Kinesis resource.

Tag consumers using the Kinesis Data Streams APIs

You can add, list, and remove tags on your consumers using the Kinesis Data Streams APIs. For examples, see the following documentation:

RegisterStreamConsumer

Registers a consumer for a Kinesis data stream with tags.

TagResource

Adds or updates tags for the specified Kinesis resource.

ListTagsForResource

Lists the tags for the specified Kinesis resource.

UntagResource

Removes tags from the specified Kinesis resource.

Handle large records

Amazon Kinesis Data Streams supports records up to 10 mebibytes (MiBs). This capability is recommended for processing intermittent data payloads that exceed the default 1 MiB record size limit. The default maximum record size for existing, and newly created streams is set to 1 MiB.

This feature benefits Internet of Things (IoT) applications, change data capture (CDC) pipelines, and machine learning workflows that require processing occasional larger data payloads. To start using large records in your stream, update your stream's maximum record size limit.

Important

Individual shard throughput limit of 1 MB/s for writes, and 2 MB/s for reads remain unchanged with support for larger record sizes. Kinesis Data Streams is designed to accommodate intermittent large records alongside a baseline traffic of records less than, or equal to 1 MiB. It is not designed to accommodate sustained high-volume large record ingestion.

How large records work

Amazon Kinesis Data Streams accepts records up to 10 MiB in size. Your stream accommodates large records by temporarily bursting beyond its sustained write throughput, then returning to its baseline rate over time. This burst capacity is continuously replenished, so your stream can handle intermittent large records alongside normal traffic without any manual capacity adjustments.

To visualize this behavior, think of your stream's write capacity as a tank that refills at a steady rate. When you send a large record, such as a 10 MiB record, the tank is temporarily depleted. It then begins refilling immediately, which means you can continue to send smaller records as capacity becomes available.

The rate at which capacity is replenished depends on several factors:

- The size of the large records
- The size of the baseline records
- The overall traffic pattern on the stream
- Your chosen partition key strategy

For best results, use a uniformly distributed partition key to spread large records across the stream's available capacity.

In on-demand mode, Kinesis Data Streams manages capacity automatically. Your stream scales its throughput up and down based on your traffic patterns, and large record burst capacity is handled transparently. You don't need to provision or manage capacity to use large records. For more information on how on-demand mode scales, see On-demand mode features and use cases.

Update your stream to use large records

To process larger records with Kinesis Data Streams

1. Navigate to the Kinesis Data Streams console.
2. Select your stream, and go to the **Configuration** tab.
3. Click **Edit**, which is located next to **Maximum record size**.
4. Set your maximum record size (up to 10 MiB).
5. Save your changes.

This setting only adjusts the maximum record size for this Kinesis data stream. Before increasing this limit, verify that all downstream applications can handle larger records.

You can also update this setting using the AWS CLI:

```
aws kinesis update-max-record-size \ --stream-arn  \
        --max-record-size-in-ki-b 5000
```

Optimize your stream performance with large records

Large records are designed for intermittent use. For best results, keep large records to less than 2% of your overall traffic. Because the stream temporarily bursts beyond its sustained throughput to deliver a large record, sending large records too frequently can reduce the capacity available for your baseline traffic. For more information on optimizing your stream performance with large records, see Throttling and best practices for optimal performance.

Mitigate throttling with large records

Because large records temporarily consume burst capacity, your stream might throttle subsequent writes until capacity is replenished. The following steps help reduce throttling:

To mitigate throttling

1. Implement retry logic with exponential back-off in your producer application.
2. Use randomized partition keys to distribute large records across the stream's available capacity.
3. Store payloads in Amazon S3 and send only metadata references to the stream for continuous streams of large records. For more information, see Processing large records with Amazon Kinesis Data Streams.

Handle large records using the Kinesis Data Streams APIs

Large record support introduces one new API, and updates two existing control plane APIs to handle records up to 10 MiBs.

API for modifying record size:

- `UpdateMaxRecordSize`: Configures the maximum record size limit for existing streams up to 10 MiBs.

Updates to existing APIs:

- `CreateStream`: Adds the optional `MaxRecordSizeInKiB` parameter for setting record size limits during the stream creation.
- `DescribeStreamSummary`: Returns the `MaxRecordSizeInKiB` field to show the current stream configuration.

All APIs listed maintain backward compatibility for existing streams. For complete API documentation, see the Amazon Kinesis Data Streams Service API Reference.

AWS components compatible with large records

The following AWS components are compatible with large records:

Component	Description
AWS SDK	AWS SDK supports handling large records. You can update your stream's maximum record size up-to 10 MiB using available methods in the AWS SDKs. For more information, see Using this service with an AWS SDK.
Kinesis Consumer Library (KCL)	Starting with version 2.x, KCL supports handling large records. To use large record support, update the `maxRecordSize` of your stream, and use KCL. For more information, see Use Kinesis Client Library.
Kinesis Producer Library (KPL)	Starting with version 1.0.5, KPL supports handling large records. To use large record support, update the `maxRecordSize` of your stream, and use KPL. For more informati on, see Develop producers using the Amazon Kinesis Producer Library (KPL).
Amazon EMR	Amazon EMR with Apache Spark supports handling large records up to the Kinesis Data Streams limit (10 MiBs). To use large record

Component	Description
	support, use the `readStream` function. For more information, see Amazon EMR and Amazon Kinesis integration.

Component	Description
Amazon Data Firehose	When used with Kinesis Data Streams, the Amazon Data Firehose behavior with large records depends on the delivery destination: • Amazon S3: Delivery of large records is supported without any additional configura tion. When you use the data format conversion, delivery of large records is supported with Firehose. When you use dynamic partitioning, delivery of large records is not supported with Firehose. • Lambda: We don't recommend using large records with Firehose when it triggers Lambda functions downstream. This may lead to intermittent failures. • HTTP: Delivery of large records is not supported with Firehose. • Snowflake: Delivery of large records is not supported with Firehose. • Amazon Redshift: Delivery of large records is not supported with Firehose. For applications requiring delivery to Snowflake or Redshift with large records, deliver the data to Amazon S3 first. After that, use Extract, Transform, Load (ETL) processes to load the data. For all other destinations, test the behavior with large records in a proof-of-concept environment before scaling to production usage. Handling large records varies by destination.

Component	Description
AWS Lambda	AWS Lambda supports payloads up to 6 MiBs. This limit includes the Kinesis payload converted to base-64 encoding, and the metadata associated with Event Source Mapping (ESM). For records less than 6 MiBs, Lambda processes them using ESM with no additional configuration required. For records larger than 6 MiBs, Lambda processes them using an on-failure destination. You must configure an on-failure destination using ESM to handle records that exceed Lambda's processing limits. Each event sent to the on-failure destination is a JSON document that contains metadata regarding the failed invocation. It is recommended to create an on-failure destination in the ESM, regardless of record size. This ensures that no records are discarded . For more information, see Configuring destinations for failed invocations.
Amazon Redshift	When streaming data from Kinesis Data Streams to Amazon Redshift, check the maximum record size that Amazon Redshift supports. For details, see Getting started with streaming ingestion from Amazon Kinesis Data Streams in the *Amazon Redshift Database Developer Guide*. Records that exceed this limit are not processed. For details on logging information for records that are not processed , see SYS_STREAM_SCAN_ERRORS in the *Amazon Redshift Database Developer Guide*.

Component	Description
Flink connector for Kinesis Data Streams	There are two approaches for consuming data from Kinesis Data Streams: the Kinesis source connector, and the Kinesis sink connector. The source connector supports handling records less than 1 MiB, and up to 10 MiBs. Do not use the sink connector for records larger than 1 MiB. For more information, see Use connectors to move data in Amazon Managed Service for Apache Flink with the DataStream API.

Regions where large records are supported

This Amazon Kinesis Data Streams feature is available only in the following AWS Regions:

AWS Region	Region Name
eu-north-1	Europe (Stockholm)
me-south-1	Middle East (Bahrain)
ap-south-1	Asia Pacific (Mumbai)
eu-west-3	Europe (Paris)
ap-southeast-3	Asia Pacific (Jakarta)
us-east-2	US East (Ohio)
af-south-1	Africa (Cape Town)
eu-west-1	Europe (Ireland)
me-central-1	Middle East (UAE)
eu-central-1	Europe (Frankfurt)
sa-east-1	South America (São Paulo)

AWS Region	Region Name
ap-east-1	Asia Pacific (Hong Kong)
ap-south-2	Asia Pacific (Hyderabad)
us-east-1	US East (N. Virginia)
ap-northeast-2	Asia Pacific (Seoul)
ap-northeast-3	Asia Pacific (Osaka)
eu-west-2	Europe (London)
ap-southeast-4	Asia Pacific (Melbourne)
ap-northeast-1	Asia Pacific (Tokyo)
us-west-2	US West (Oregon)
us-west-1	US West (N. California)
ap-southeast-1	Asia Pacific (Singapore)
ap-southeast-2	Asia Pacific (Sydney)
il-central-1	Israel (Tel Aviv)
ca-central-1	Canada (Central)
ca-west-1	Canada West (Calgary)
eu-south-2	Europe (Spain)
cn-northwest-1	China (Ningxia)
eu-central-2	Europe (Zurich)
us-gov-east-1	AWS GovCloud (US-East)
us-gov-west-1	AWS GovCloud (US-West)

Perform resilience testing with AWS Fault Injection Service

AWS Fault Injection Service is a fully managed service that helps you perform fault injection experiments on your AWS workloads. AWS FIS integration with Amazon Kinesis Data Streams enables you to test your application resilience against common Amazon Kinesis Data Streams API errors in a controlled environment. This capability allows you to validate error handling, retry logic, and monitor systems before encountering failures. For more information, see What is AWS Fault Injection Service?.

Actions

- API internal error: This injects internal errors into requests made by the the target IAM role. The specific response depends on each service and API. The action `aws:fis:inject-api-internal-error` creates `InternalFailure` errors (HTTP 500).
- API throttle error: This injects internal errors into requests made by the the target IAM role. The specific response depends on each service and API. The action `aws:fis:inject-api-throttle-error` creates `ThrottlingException` errors (HTTP 400).
- API unavailable error: This injects internal errors into requests made by the the target IAM role. The specific response depends on each service and API. The action `aws:fis:inject-api-unavailable-error` creates `ServiceUnavailable` errors (HTTP 503).
- API provisioned throughput exception: This injects internal errors into requests made by the the target IAM role. The specific response depends on each service and API. The action `aws:kinesis:inject-api-provisioned-throughput-exception` creates `ProvisionedThroughputExceededException` errors (HTTP 400).
- API expired iterator exception: This injects internal errors into requests made by the the target IAM role. The specific response depends on each service and API. The action `aws:kinesis:inject-api-expired-iterator-exception` creates `ExpiredIteratorException` errors (HTTP 400).

For more information, see Amazon Kinesis Data Streams actions.

Considerations

- You can use actions above with both provisioned, and on-demand offerings for Amazon Kinesis Data Streams.
- Your streaming resumes once the experiment completes based on the duration selected. You can also stop a running experiment before it completes. Alternatively, you can define a stop

condition to stop the experiment based on alarms that define the application health in a Amazon CloudWatch Application Insights.
- You can test up to 280 streams.

For more information on regional support, see AWS Fault Injection Service endpoints and quotas.

Provisioned throughput exception errors

Provisioned throughput exceeded exception errors (HTTP 400) occur when the request rate for a Kinesis stream surpasses the throughput limits of one or more shards. Each shard has specific read, and write capacity limits, and exceeding those limits triggers this exception. Scenarios leading to this exception include: sudden spikes in data ingestion or consumption, insufficient shard capacity for the data volume being processed, or uneven distribution of partition keys.

Recommendations for handling exceptions

- Implement exponential back-off and re-try mechanisms.
- Increase the number of shards to accommodate higher throughput.
- Ensure that there is proper distribution of partition keys.
- Monitor stream metrics.

Additionally, using the Kinesis on-demand capacity mode helps to automatically adjust workloads, and minimize the occurrence of this exception. For more information, see What is AWS Fault Injection Service?

> **Note**
>
> Improper distribution issues are outside of on-demand mode capability of automatic scaling.

To perform a basic experiment

1. Use baseline metrics: record normal throughput patterns before testing.
2. Create an experiment: use the `aws:kinesis:inject-api-provisioned-throughput-exception` action.
3. Configure the intensity: start with 25% request throttling.

4. Monitor the responses: verify re-try logic with exponential back-off.
5. Validate the scaling: confirm that auto-scaling triggers the activation.
6. Check the alarms: ensure that the `CloudWatch` alarms are running as expected.

Applications should implement proper back-off strategies, monitor `WriteProvisionedThroughputExceeded`, and `ReadProvisionedThroughputExceeded` metrics, and trigger shard scaling when appropriate.

Action details

- **Resource Type**: IAM Role ARN
- **Target Operations**: `PutRecord`, `PutRecords`, `GetRecords`
- **Error Code**: `ProvisionedThroughputExceededException`(HTTP 400)
- **Description**: simulates scenarios where request rate exceeds shard capacity limits, testing application throttling, and scaling responses.

Parameters

- **IAM Role ARN**: the role that your application uses for Kinesis Data Streams operations.
- **Operations**: target operations: `PutRecord`, `PutRecords`, `GetRecords`.
- **Resource List**: the specific stream names or shard identifiers.
- **Duration**: the experiment duration, which is the duration from one minute to 12 hours. In the AWS FIS API, the value is a string in ISO 8601 format. For example, PT1M represents one minute. In the AWS FIS console, you enter the number of seconds, minutes, or hours.
- **Intensity**: the percentage of requests to throttle.

Required permissions

- `kinesis:InjectApiError`

Example experiment template

The following example shows a provisioned throughput exception for all requests up to 5 Kinesis Data streams with the specified tag. AWS FIS selects the streams to affect at random. After 5 minutes the fault is removed.

```
{
    "description": "Kinesis stream experiment",
    "targets": {
        "KinesisStreams-Target-1": {
            "resourceType": "aws:kinesis:stream",
            "resourceTags": {
                  "tag-key": "tag-value"
            },
            "selectionMode": "COUNT(5)"
        }
    },
    "actions": {
         "kinesis": {
              "actionId": "aws:kinesis:stream-provisioned-throughput-exception",
              "description": "my-stream",
              "parameters": {
                   "duration": "PT5M",
                   "percentage": "100",
                   "service": "kinesis"
              },
              "targets": {
                    "KinesisStreams": "KinesisStreams-Target-1"
              }
         }
   },
   "stopConditions": [
         {
              "source": "none"
         }
   ],
   "roleArn": "arn:aws:iam::111122223333:role/role-name",
   "tags": {},
   "experimentOptions": {
       "accountTargeting": "single-account",
       "emptyTargetResolutionMode": "fail"
   }
}
```

Experiment role permissions example

The following permission allows you to run the `aws:kinesis:stream-provisioned-throughput-exception` and `aws:kinesis:stream-expired-iterator-exception` actions on a specific stream that impact 50% of requests.

Expired iterator exception errors

Expired iterator exception errors (HTTP 400) occur when the shard iterator is expired, and is no longer used to retrieve stream records when calling `GetRecords`. This occurs when there are delays between read operations, which are caused by long-running data processing tasks, network issues, or application downtime.

> **Note**
>
> A shard iterator is valid for 5 minutes after the time it's issued.

Recommendations for handling exceptions

- Refreshing shard iterators before they expire.
- Incorporating error handling to obtain new iterators.
- Utilizing the Kinesis Kinesis Client Library (KCL) which automatically manages shard iterator expiration.

For more information, see What is AWS Fault Injection Service?

To perform a basic experiment

1. Create an experiment template: use the AWS FIS console.
2. Select the action: use the `aws:kinesis:inject-api-expired-iterator-exception` action.
3. Configure the targets: specify the IAM role and Kinesis Data Streamsoperations.
4. Set the duration: start with 5-10 minutes for initial testing.
5. Add stop conditions: stop conditions for AWS FIS.
6. Run the experiment: monitor the application behavior.

Action details

- **Resource Type**: IAM Role ARN
- **Target Operations**: `GetRecords`
- **Error Code**: `ExpiredIteratorException`(HTTP 400)

- **Description**: the provided iterator exceeds the maximum age allowed, simulating scenarios where record processing is too slow or checkpointing logic fails.

Parameters

- **IAM Role ARN**: the role that your application uses for Kinesis Data Streams operations.
- **Operations**: target operations: `GetRecords`
- **Resource List**: the specific stream names or ARNs.
- **Duration**: the experiment duration. This is configurable.
- **Intensity**: the percentage of requests to throttle.

Required permissions

- `kinesis:InjectApiError`

Service-managed record distribution strategy

Amazon Kinesis Data Streams offers two record distribution strategies that determine how records are distributed across shards in a stream:

- **User-managed partition keys** (default) – You provide a partition key with each record, and Kinesis Data Streams uses the partition key hash to determine the shard placement. This is the traditional behavior for workloads that require ordering guarantees for related records.
- **Service-managed record distribution** – Kinesis Data Streams automatically distributes records evenly across shards using service-managed algorithms. You do not need to provide partition keys. This is designed for stateless workloads where record ordering is not required.

How service-managed record distribution works

When you enable the service-managed record distribution strategy on a stream, Kinesis Data Streams takes over the responsibility of distributing records across shards. The service uses internal algorithms to spread records evenly and avoid hot shards. This eliminates common issues with customer-generated partition keys, such as uneven traffic distribution. It also prevents rotating hot key throttling.

Key characteristics of service-managed record distribution:

- **Stream-level setting** – The record distribution strategy is configured at the stream level and applies to all records written to that stream.
- **On-demand streams only** – Service-managed record distribution is available for on-demand streams (On-Demand Standard and On-Demand Advantage). It is not supported on provisioned streams.
- **No ordering guarantees** – Because the service distributes records across shards using internal algorithms, records with the same business entity do not necessarily arrive at the same shard. Do not use this mode for workloads that depend on ordering.
- **Partition keys are ignored** – When service-managed mode is enabled, any partition key values provided in PutRecord or PutRecords API calls are ignored. The service handles distribution internally.
- **No additional cost** – Service-managed record distribution is available at no additional cost. Standard Kinesis Data Streams pricing applies.

When to use service-managed record distribution

Use service-managed record distribution for stateless streaming workloads where record ordering is not required:

- Log aggregation
- Metrics collection
- IoT telemetry
- Clickstream analytics

Continue using customer-managed partition keys for stateful workloads that require ordering guarantees:

- Database change data capture (CDC) where updates to the same row must be processed sequentially
- Financial transaction processing where account-level ordering is required
- Session-based analytics where all events from a user session must be processed together
- Any application where business logic depends on processing related records in the order they were produced

Comparison of record distribution strategies

The following table compares the two record distribution strategies available in Kinesis Data Streams.

Record distribution strategy comparison

Characteristic	User-managed partition keys	Service-managed record distribution
Partition key required	Yes – you must provide a partition key with each record	No – the service distributes records automatically
Record ordering	Records with the same partition key are delivered to the same shard in order	No ordering guarantees – records are distributed for optimal throughput

Characteristic	User-managed partition keys	Service-managed record distribution
Shard distribution	Depends on your partition key strategy – poorly distributed keys can cause hot shards	Even distribution managed by service algorithms – eliminates hot shards
Supported stream modes	On-Demand Standard, On-Demand Advantage, and Provisioned	On-Demand Standard and On-Demand Advantage only
Best for	Stateful workloads requiring ordering (CDC, financial transactions, session analytics)	Stateless workloads (logs, metrics, telemetry, clickstream)
Producer code changes required	N/A (existing behavior)	Refer the section KPL and KCL with service-managed streams

Transition behavior

You can switch between service-managed and user-managed record distribution at any time. The mode change takes effect immediately without downtime or data loss for records already written. Producer and consumer applications continue to work with the following exceptions:

- After you switch to user-managed distribution, producers that omit a partition key fail and must be updated to supply one.
- After you switch to service-managed distribution and producers begin omitting the partition key, consumers that assume a partition key is always present can fail when they read a record that has none. Verify that your consumers handle an absent partition key before you enable AUTO. For more information, see the AWS SDK considerations in Important considerations and KPL and KCL with service-managed streams.

During the transition:

- Records already in the stream retain their original shard assignments based on the previous strategy.
- New records arriving after the mode change are distributed according to the new strategy.

- Consumers continue reading records from shards in sequence regardless of how those records were distributed.

For instructions on enabling or disabling service-managed record distribution, see Configure service-managed record distribution.

Switching strategies changes how records are placed on shards going forward, which affects per–partition-key ordering:

- **User-managed to service-managed (USER_PARTITION_KEY → AUTO)** – Records that previously shared a partition key were routed to the same shard and ordered relative to one another. After the switch, new records are distributed evenly across shards regardless of partition key, so records sharing a partition key are no longer guaranteed to land on the same shard or be processed in order.
- **Service-managed to user-managed (AUTO → USER_PARTITION_KEY)** – After the switch, new records are again routed by partition key hash, so records with the same partition key resume landing on the same shard. Producers must supply a partition key after the switch; PutRecord and PutRecords calls that omit it fail. Records written while the stream was in AUTO mode keep their original shard placement and are not redistributed, so a single partition key's records can be split across the pre-switch (AUTO) and post-switch shards.

Important considerations

- **KPL upgrade for null partition keys** – Customers who want to send records without a partition key should upgrade to the latest KPL version. For customers who continue to provide partition keys, no KPL or KCL changes are required. For more information, see KPL and KCL with service-managed streams.
- **AWS SDK considerations** – If you use the AWS SDK directly (rather than the KPL or KCL) to produce to or consume from a service-managed stream, review the following before you enable AUTO:
  - **Producers** – Upgrade to the latest SDK version to omit the partition key when writing to a service-managed stream. On the latest SDK, `PartitionKey` is optional, so you can call `PutRecord` and `PutRecords` without it. If you continue to supply a partition key, no SDK change is required — the service accepts the value and ignores it for shard placement. Before you enable AUTO and start sending null partition keys on a stream that existing consumers read from, update those consumers to handle an absent `PartitionKey`.

  - **Consumers** – On a service-managed stream, `GetRecords` and `SubscribeToShard` return `PartitionKey` as an absent (null) value for records written without one. Any consumer that reads `PartitionKey` and assumes it is always present can fail at runtime when it reads such a record — this applies regardless of SDK version. Upgrade to the latest SDK version to read these records safely, and update your code to handle an absent value. For the AWS SDK for Go, AWS SDK for Kotlin, and AWS SDK for Rust, upgrading additionally changes `PartitionKey` to an optional type, so an unhandled absent value surfaces as a compile-time error when you upgrade. Consumers that do not read `PartitionKey` are unaffected.
- **Aggregation not supported** – KPL aggregation is not supported on service-managed streams in any KPL version. Consider disabling aggregation if you want to use service-managed record distribution. For details, see KPL and KCL with service-managed streams.
- **Cannot be mixed within a stream** – The record distribution strategy applies to the entire stream. You cannot use both service-managed and user-managed partition keys on the same stream simultaneously. For workloads that require both ordered and unordered data, create separate streams.
- **No new service limits** – Service-managed record distribution does not introduce any new limits. Existing Kinesis Data Streams limits continue to apply, including the 1 MB per record maximum, 1,000 records per second per shard for writes, and 500 records per PutRecords request.

Configure service-managed record distribution

You can enable service-managed record distribution when creating a new stream or update the configuration on an existing on-demand stream. The record distribution strategy is a stream-level setting that takes effect immediately.

Prerequisites

Before you configure service-managed record distribution, ensure the following:

- Your stream uses **on-demand** capacity mode (On-Demand Standard or On-Demand Advantage). Service-managed record distribution is not supported on provisioned streams.
- If you use KPL aggregation, ensure the KPL does not aggregate records when writing to a service-managed stream. You can either explicitly disable aggregation, or upgrade to the latest KPL version and grant it the `kinesis:DescribeStreamSummary` permission so that it detects the stream and disables aggregation automatically. Aggregation is not supported on service-

managed streams in any KPL version. For more information, see KPL and KCL with service-managed streams.

- If you want to send records without a partition key (null partition key), upgrade to the latest KPL version. For customers who continue providing partition keys, no KPL or KCL changes are required.

Create a stream with service-managed record distribution

To create a new stream with service-managed record distribution, include the `RecordDistributionStrategy` parameter set to `AUTO` in your CreateStream API call. If you do not specify this parameter, the stream defaults to `USER_PARTITION_KEY` (user-managed partition keys).

Example Create a stream with service-managed distribution (AWS CLI)

```
aws kinesis create-stream \
    --stream-name my-telemetry-stream \
    --stream-mode-details StreamMode=ON_DEMAND \
    --record-distribution-strategy AUTO
```

Example Create a stream with service-managed distribution (AWS SDK for Java)

```
CreateStreamRequest request = CreateStreamRequest.builder()
    .streamName("my-telemetry-stream")
    .streamModeDetails(StreamModeDetails.builder()
        .streamMode(StreamMode.ON_DEMAND)
        .build())
    .recordDistributionStrategy(RecordDistributionStrategy.AUTO)
    .build();

kinesisClient.createStream(request);
```

Enable service-managed record distribution on an existing stream

To enable service-managed record distribution on an existing on-demand stream, use the `UpdateStreamRecordDistributionStrategy` API. The change takes effect immediately. Before you enable `AUTO`, verify that your consumers can handle an absent (null) partition key, and ensure any KPL producers do not aggregate records. For more information, see Safe migration patterns.

Example Enable service-managed distribution on an existing stream (AWS CLI)

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream \
    --record-distribution-strategy AUTO
```

Example Enable service-managed distribution (AWS SDK for Java)

```
UpdateStreamRecordDistributionStrategyRequest request =
    UpdateStreamRecordDistributionStrategyRequest.builder()
        .streamName("my-telemetry-stream")
        .recordDistributionStrategy(RecordDistributionStrategy.AUTO)
        .build();

kinesisClient.updateStreamRecordDistributionStrategy(request);
```

You can also specify the stream by name instead of ARN:

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-name my-telemetry-stream \
    --record-distribution-strategy AUTO
```

Verify the current record distribution strategy

To check which record distribution strategy a stream is currently using, call the `DescribeStreamSummary` API. The response includes the `RecordDistributionStrategy` field.

Example Check record distribution strategy (AWS CLI)

```
aws kinesis describe-stream-summary \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream
```

The response includes:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "my-telemetry-stream",
        "StreamARN": "arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-
stream",
```

```
        "StreamStatus": "ACTIVE",
        "StreamModeDetails": {
            "StreamMode": "ON_DEMAND"
        },
        "RecordDistributionStrategy": "AUTO",
        ...
    }
}
```

Configure the record distribution strategy in the console

You can view and configure the record distribution strategy only for streams that use on-demand capacity mode in the Kinesis Data Streams console. If your stream uses provisioned capacity mode, the console displays a dash to indicate that the setting isn't applicable, and you can't change the strategy.

Choose a record distribution strategy when you create a stream

1. Open the Kinesis Data Streams console at https://console.aws.amazon.com/kinesis.
2. Choose **Create data stream**.
3. Enter a name for your data stream.
4. Under **Data stream capacity**, choose **On-demand** capacity mode. The **Record distribution strategy** panel appears below the capacity panel. This panel is hidden when **Provisioned** mode is selected.
5. In the **Record distribution strategy** panel, choose one of the following:

   - **Auto** – Kinesis Data Streams distributes records evenly across shards and ignores any partition key that producers supply. Choose this option for stateless workloads that do not require partition-key ordering.
   - **User set** – Producers must supply a partition key, and Kinesis Data Streams uses it to determine shard placement. This is the default.
6. Choose **Create data stream**.

View and edit the record distribution strategy for an existing stream

1. Open the Kinesis Data Streams console at https://console.aws.amazon.com/kinesis.
2. Choose the data stream you want to update.

3. Choose the **Configuration** tab. The current record distribution strategy is shown in the **Data stream capacity** panel.
4. Choose **Edit** next to **Record distribution strategy**.
5. Choose **Auto** or **User set**, then save your changes. The change takes effect immediately.

> **Note**
>
> The console displays a warning when you switch strategies, reminding you that service-managed (**Auto**) distribution eliminates partition-key ordering guarantees, and that user-managed (**User set**) distribution requires producers to supply a partition key.

You can also view the record distribution strategy in the following places in the console:

- The **Record distribution strategy** column in the data streams table shows **Auto**, **User set**, or a dash for provisioned streams.
- The **Data stream summary** panel on the stream detail page shows the current record distribution strategy.

Switch back to user-managed partition keys

To switch a stream back to user-managed partition keys, call the `UpdateStreamRecordDistributionStrategy` API with `RecordDistributionStrategy` set to `USER_PARTITION_KEY`.

```
aws kinesis update-stream-record-distribution-strategy \
    --stream-arn arn:aws:kinesis:us-east-1:123456789012:stream/my-telemetry-stream \
    --record-distribution-strategy USER_PARTITION_KEY
```

The change takes effect immediately. All subsequent PutRecord and PutRecords API calls require a partition key, and the service uses the provided partition key to determine shard placement.

> **Important**
>
> Switching to user-managed partition keys restores per-partition-key ordering — records with the same partition key resume landing on the same shard. Before switching, ensure

that your producers are providing meaningful partition keys. If producers are still sending records without partition keys after you switch to user-managed mode, the API calls fail.

Safe migration patterns

Switching the record distribution strategy on a stream takes effect immediately and propagates to all infrastructure within approximately one second. There is no notification, drain, or quiesce mechanism. Active consumers processing records mid-batch see an immediate behavior change. Plan your migration carefully using the following guidance.

Migrate from user-managed to service-managed (USER_PARTITION_KEY → AUTO)

Follow these steps to safely enable service-managed record distribution on a production stream:

1. **Verify consumer compatibility** — Ensure all consumers can handle `null` partition keys in the `GetRecords` and `SubscribeToShard` responses. Test in a non-production environment first.
2. **Handle KPL aggregation** — If any producers use KPL aggregation, ensure the KPL does not aggregate records when writing to the service-managed stream. Either explicitly disable aggregation, or upgrade to the latest KPL version and grant it the `kinesis:DescribeStreamSummary` permission so that it detects the stream and disables aggregation automatically. Aggregation is not supported on service-managed streams. See KPL and KCL with service-managed streams.
3. **Notify consumer teams** — Coordinate with all teams consuming from the stream. Inform them that partition keys will become `null` (or ignored) and that record ordering across shards is no longer guaranteed.
4. **Switch during low-traffic periods** — While the switch itself causes no data loss, performing it during low traffic reduces the number of in-flight records affected by the behavior change.
5. **Enable service-managed distribution** — Call the `UpdateStreamRecordDistributionStrategy` API to set `RecordDistributionStrategy` to `AUTO`.
6. **Monitor after switching** — Watch Amazon CloudWatch metrics for the stream: `IncomingRecords`, `IncomingBytes` (per shard, to verify even distribution), `WriteProvisionedThroughputExceeded` (should drop to zero), and `GetRecords.IteratorAgeMilliseconds` (to verify consumers are keeping up).

Migrate from service-managed to user-managed (AUTO → USER_PARTITION_KEY)

Follow these steps to safely switch a production stream back to user-managed partition keys:

1. **Verify all producers provide partition keys** — After switching, PutRecord and PutRecords calls that omit the `PartitionKey` field will fail with a validation error. Confirm that all producers are sending meaningful partition keys before switching.
2. **Verify consumer logic handles partition key changes** — Consumers that were processing `null` partition keys will now receive actual values. If your consumer logic uses partition keys for routing or grouping, verify it handles the transition correctly.
3. **Notify all consuming teams** — Inform teams that records will resume being distributed by partition key hash. Related records with the same partition key will again be sent to the same shard.
4. **Switch during low-traffic periods** — Minimize the number of in-flight records affected by the behavior change.
5. **Switch to user-managed** — Call the `UpdateStreamRecordDistributionStrategy` API to set `RecordDistributionStrategy` to `USER_PARTITION_KEY`.
6. **Monitor after switching** — Watch for `WriteProvisionedThroughputExceeded` (hot shards may return if partition key distribution is uneven) and consumer lag metrics.

Migration considerations

- **Immediate effect** — The strategy change reaches all service infrastructure within about one second. The service does not roll out the change gradually or use a canary mechanism. Active consumers processing a batch of records may observe the behavior change mid-batch.
- **No consumer notification** — The service does not send notifications to active consumers when the strategy changes. Consumers are not drained or quiesced. Design your consumer applications to handle `null` partition keys gracefully regardless of the current stream mode.
- **Console warning** — The Kinesis Data Streams console displays a warning banner when you switch between modes, reminding you that service-managed mode eliminates ordering guarantees and that user-managed mode requires producers to provide partition keys.
- **Reversible** — You can switch back at any time. If you observe issues after a migration, switch the stream back to the previous mode immediately.
- **Test in non-production first** — Always test the migration in a non-production environment with representative traffic patterns before applying to production streams.

Error handling

The following errors can occur when configuring the record distribution strategy:

Configuration errors

Error	Cause	Status
`InvalidArgumentException`	Attempting to enable service-managed record distribution on a provisioned stream. Service-managed record distribution is only supported for on-demand streams. The service returns a message similar to the following: `RecordDistributionStrategy=AUTO is only supported for on-demand streams. Stream [StreamName] is using PROVISIONED mode.`	400
`ResourceNotFoundException`	The specified stream does not exist.	400
`LimitExceededException`	Too many concurrent stream configuration changes. Wait and retry.	400

IAM permissions

To configure the record distribution strategy, the IAM principal must have the following permissions:

- `kinesis:CreateStream` – to create a stream with service-managed record distribution
- `kinesis:UpdateStreamRecordDistributionStrategy` – to change the record distribution strategy on an existing stream
- `kinesis:DescribeStreamSummary` – to view the current record distribution strategy. This permission is also required by the updated KPL, which calls `DescribeStreamSummary` to detect whether a stream uses service-managed record distribution. Grant it to any KPL producer that writes to a service-managed stream. For more information, see Grant DescribeStreamSummary permission to the KPL.

Write records to service-managed streams

When a stream uses service-managed record distribution, Kinesis Data Streams automatically distributes records across shards. You can write records using the PutRecord and PutRecords APIs without providing partition keys. Existing producers that already provide partition keys continue to work without modification — the service ignores the provided values and uses its own distribution algorithms.

> **Important**
>
> If your producers use the KPL with aggregation enabled (the default in existing KPL versions), you must disable aggregation before writing to a service-managed stream, or upgrade to the latest KPL version. Aggregation is not supported on service-managed streams, and aggregated records that are written can be silently dropped by KCL consumers during de-aggregation. For details, see KPL and KCL with service-managed streams.

Producer API behavior with service-managed distribution

When service-managed record distribution is enabled on a stream, the following behavior applies to the PutRecord and PutRecords APIs:

- **PartitionKey** – Optional. If provided, the value is ignored for shard placement. The service distributes the record using internal algorithms.
- **ExplicitHashKey** – If provided, the value is ignored for shard placement, the same as `PartitionKey`. The service distributes the record using internal algorithms.
- **SequenceNumberForOrdering** (PutRecord only) – Must not be provided. `PutRecord` requests that include this field are rejected with an `InvalidArgumentException`. Ordering is not applicable for service-managed distribution. This field does not exist on `PutRecords`.

> **Note**
>
> Existing producer applications that provide partition keys continue to work without modification when service-managed mode is enabled. The service ignores the partition key values, which means you can enable this feature on production streams without coordinating code deployments across your teams.

If you use the KPL to send records without a partition key, upgrade to the latest KPL version. If you use the AWS SDK directly, no SDK change is required as long as you continue to supply a partition key; to omit the partition key, you must upgrade to the latest SDK version. For more information, see Important considerations and KPL and KCL with service-managed streams.

Examples

Example Write a single record without a partition key (AWS CLI)

```
aws kinesis put-record \
    --stream-name my-telemetry-stream \
    --data "eyJtZXRyaWMiOiAiY3B1X3V0aWxpemF0aW9uIiwgInZhbHVlIjogNzIuNX0="
```

Example Write a single record without a partition key (AWS SDK for Java)

```
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-telemetry-stream")
    .data(SdkBytes.fromUtf8String("{\"metric\": \"cpu_utilization\", \"value\":
 72.5}"))
    .build();

PutRecordResponse response = kinesisClient.putRecord(request);
```

Example Write a batch of records without partition keys (AWS SDK for Java)

```
List<PutRecordsRequestEntry> entries = new ArrayList<>();
entries.add(PutRecordsRequestEntry.builder()
    .data(SdkBytes.fromUtf8String("{\"event\": \"page_view\", \"page\": \"/home\"}"))
    .build());
entries.add(PutRecordsRequestEntry.builder()
    .data(SdkBytes.fromUtf8String("{\"event\": \"page_view\", \"page\": \"/products
\"}"))
    .build());

PutRecordsRequest request = PutRecordsRequest.builder()
    .streamName("my-telemetry-stream")
    .records(entries)
    .build();
```

```
PutRecordsResponse response = kinesisClient.putRecords(request);
```

Example Existing producer with partition key (continues to work)

```
// This producer code continues to work without modification.
// The partition key is ignored by the service when service-managed
// record distribution is enabled.
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-telemetry-stream")
    .partitionKey("device-12345")
    .data(SdkBytes.fromUtf8String("{\"temp\": 22.1}"))
    .build();

PutRecordResponse response = kinesisClient.putRecord(request);
```

Error handling

The following errors are specific to writing records to service-managed streams:

Producer errors for service-managed streams

Error	Cause	Status
`InvalidArgumentException`	`SequenceNumberForOrdering` was provided in a `PutRecord` request. This field must not be included when writing to a service-managed stream, because ordering is not applicable. (`PutRecords` has no `SequenceNumberForOrdering` field.) A supplied `ExplicitHashKey` is ignored and does not cause an error. The service returns a message similar to the following: `Stream has SERVICE_DEFINED PartitionKey enabled. SequenceNumberForOrdering must not be provided.`	400

Migrate existing producers

You do not need to modify existing producers when enabling service-managed record distribution on a stream. Existing producers that provide partition keys continue to function — the service ignores the partition key values and distributes records using internal algorithms.

To migrate producers over time:

1. Enable service-managed record distribution on your stream (see Configure service-managed record distribution).
2. Verify that your stream is receiving records and distributing them evenly by monitoring CloudWatch metrics (see `IncomingRecords` and `IncomingBytes` per shard).
3. (Optional) Update your producer code to remove partition key logic over time. This is not required for the feature to work but simplifies your codebase.

> **Important**
>
> If your producers use KPL aggregation, note that aggregation is not supported on service-managed streams in any KPL version. You must disable aggregation before enabling service-managed record distribution. Aggregated records that are written to a service-managed stream can be silently dropped by KCL consumers during de-aggregation, with no error surfaced by the producer or consumer. For details, see KPL and KCL with service-managed streams.

Best practices

- **Use PutRecords for batching** – Continue using the PutRecords API for batch writes. Batching reduces API call overhead and improves throughput regardless of the record distribution strategy.
- **Remove SequenceNumberForOrdering** – This `PutRecord` field is rejected by service-managed streams (`PutRecords` has no such field). A supplied `ExplicitHashKey` is ignored rather than rejected, but you should remove it as well because it has no effect. Update your producer code to remove these fields.

- **Monitor shard utilization** – Use CloudWatch metrics to verify that records are evenly distributed across shards. Monitor `WriteProvisionedThroughputExceeded` to confirm no hot shards are occurring.

Read records from service-managed streams

Consumers reading from service-managed streams use the same APIs as streams with user-managed partition keys. The primary difference is in how the `PartitionKey` field appears in the response. Consumers continue to read records from shards in sequence regardless of the distribution strategy.

GetRecords API behavior

When reading records from a service-managed stream using the GetRecords API, the response includes the following behavior for each record:

- **PartitionKey** – Returns `null` if the producer did not provide a partition key when writing the record. If the producer provided a partition key (which was ignored for shard placement), the original value is returned as-is.
- **SequenceNumber** – Returned normally. Sequence numbers continue to be assigned to each record regardless of distribution strategy.
- **ApproximateArrivalTimestamp** – Returned normally.
- **Data** – Returned normally. Record data is unaffected by the distribution strategy.

Example GetRecords response for a service-managed stream


```
{
    "Records": [
        {
            "SequenceNumber": "49654108311168735921334927185702664421081712646",
            "ApproximateArrivalTimestamp": 1719849600.123,
            "Data": "eyJtZXRyaWMiOiAiY3B1X3V0aWxpemF0aW9uIiwgInZhbHVlIjogNzIuNX0=",
            "PartitionKey": null,
            "EncryptionType": "KMS"
        },
        {
            "SequenceNumber": "49654108311168735921334927186891775532192823757",
            "ApproximateArrivalTimestamp": 1719849600.456,
```

```
            "Data": "eyJldmVudCI6ICJwYWdlX3ZpZXciLCAicGFnZSI6ICIvaG9tZSJ9",
            "PartitionKey": "device-12345",
            "EncryptionType": "KMS"
        }
    ],
    "NextShardIterator": "EXAMPLEShardIterator1234567890EXAMPLE",
    "MillisBehindLatest": 0
}
```

> **Note**
>
> In the example above, the first record was written without a partition key (the producer omitted it), so `PartitionKey` is `null`. The second record was written with a partition key (which was ignored for distribution purposes), so the original value is returned.

SubscribeToShard API behavior

When using enhanced fan-out with the `SubscribeToShard` API, the behavior is the same as GetRecords. The `PartitionKey` field in each record in the `SubscribeToShardEvent` follows the same rules:

- Returns `null` if the producer did not provide a partition key.
- Returns the original value if the producer provided one (even though it was ignored for shard placement).

Consumer considerations

When consuming from service-managed streams, keep the following in mind:

- **Handle null partition keys** – Your consumer application should handle `null` partition key values gracefully. Do not use partition keys from service-managed streams for any business logic.
- **Shard iteration is unchanged** – You continue to use shard iterators to read records. The mechanism for traversing shards, handling resharding, and managing checkpoints remains the same.
- **No ordering assumptions** – Do not assume any ordering relationship between records in a service-managed stream. Records from the same producer or business entity may appear on different shards.

- **Downstream services work as-is** – AWS Lambda, Amazon Data Firehose, Amazon Managed Service for Apache Flink, and other AWS services that consume from Kinesis Data Streams continue to work without modification. However, if your downstream processing logic relies on partition keys for routing or grouping, you should review that logic.

Monitor consumption from service-managed streams

Use the following Amazon CloudWatch metrics to monitor consumption from service-managed streams:

- **GetRecords.Records** – The number of records retrieved per GetRecords call. Use this to verify that consumption is proceeding normally.
- **GetRecords.IteratorAgeMilliseconds** – How far behind the consumer is from the latest record. Use this to detect consumer lag.
- **ReadProvisionedThroughputExceeded** – The number of GetRecords calls throttled for the shard. This should not change based on the distribution strategy.

KPL and KCL with service-managed streams

Customers who want to send records without a partition key (null partition key) to service-managed streams should upgrade to the latest KPL version. Customers who continue to provide a partition key do not need KCL changes, but they must ensure the KPL does not aggregate records when writing to a service-managed stream.

> **Important**
>
> Existing KPL versions aggregate records by default. Aggregation is not supported on service-managed streams, and aggregated records that are written can be silently dropped by KCL consumers during de-aggregation, with no error surfaced by the producer or consumer. If you write to a service-managed stream with the KPL, either upgrade to the latest KPL version and grant it the `kinesis:DescribeStreamSummary` permission (so it detects the stream and stops aggregating), or explicitly disable aggregation. For details, see Grant DescribeStreamSummary permission to the KPL and Aggregation is not supported on service-managed streams.

Upgrade KPL to send null partition keys

To send records without providing a partition key to a service-managed stream, you must upgrade to the latest KPL version. The updated KPL can detect whether a stream uses service-managed record distribution (AUTO) and handle null partition keys accordingly.

Customers using Kinesis Producer Library can simply upgrade to the latest version at the Amazon Kinesis Producer Library GitHub repository to benefit from this capability.

The updated KPL detects the stream's record distribution strategy by calling the `DescribeStreamSummary` API. Grant the `kinesis:DescribeStreamSummary` permission to the identity the KPL uses. Without it, the KPL cannot detect a service-managed stream and falls back to aggregating records. For details, see Grant DescribeStreamSummary permission to the KPL.

Grant DescribeStreamSummary permission to the KPL

The updated KPL calls the `DescribeStreamSummary` API to detect whether a stream uses service-managed record distribution (AUTO). Ensure that the IAM principal your producer uses has the `kinesis:DescribeStreamSummary` permission for the stream. This permission is included in the minimum-permissions policy for the KPL; if you follow that policy, no change is required.

> **Important**
>
> If the KPL cannot call `DescribeStreamSummary`, it cannot reliably detect that a stream uses service-managed record distribution. This can happen, for example, if the `kinesis:DescribeStreamSummary` permission is not granted. In this case, the KPL falls back to aggregating records. Aggregation is not supported on service-managed streams, so the aggregated records that are written can be silently dropped by KCL consumers during de-aggregation, with no error surfaced by the producer or consumer. Grant the `kinesis:DescribeStreamSummary` permission before you enable service-managed record distribution on a stream that KPL producers write to.

No changes required for existing producers and consumers

If you continue to provide a partition key in your PutRecord and PutRecords API calls, and your producer does not use KPL aggregation, no KPL or KCL changes are required. Your existing producer and consumer applications continue to work without modification when service-managed

record distribution is enabled on a stream — the service ignores the partition key values you provide and distributes records using internal algorithms.

If your producer uses KPL aggregation (the default in existing KPL versions), you must disable aggregation or upgrade the KPL before writing to a service-managed stream. See Aggregation is not supported on service-managed streams.

KCL requires no changes regardless of whether you send null partition keys or not. KCL continues to read records from shards in sequence as it does today.

Aggregation is not supported on service-managed streams

KPL aggregation is not supported on streams configured with service-managed record distribution, in any KPL version. Aggregated records are not rejected at write time. The failure surfaces in one of two ways, depending on whether the KPL treats the write as successful:

- **Silent drop at the consumer** – KPL aggregation groups multiple user records into a single Kinesis Data Streams record based on the partition key hash, and expects those records to land on the shard the KPL predicts. On a service-managed stream, the service places records using its own algorithm rather than the partition key, which breaks the KPL and KCL aggregation contract. Aggregated records that are written are silently dropped by KCL consumers during de-aggregation, with no error surfaced by the producer or consumer.
- **Reported write failure at the producer** – When the aggregated records the KPL sends are not committed to the shard the KPL predicts, the KPL retries them for approximately 30 seconds and then reports a write failure to your application through its callbacks.

To avoid both conditions, disable aggregation, or upgrade to the latest KPL version and grant it the `kinesis:DescribeStreamSummary` permission so that it detects service-managed streams and disables aggregation automatically. You can also set `setRecordDistributionStrategyDefault` to `AUTO` so the KPL does not aggregate before it discovers the stream's strategy.

> **Note**
>
> For on-demand streams, KPL aggregation provides no cost benefit because pricing is based on throughput rather than the number of PUT requests. Consider disabling aggregation if you want to use service-managed record distribution.

KCL considerations

If you use the Kinesis Client Library (KCL) to consume from service-managed streams, no KCL changes are required. KCL continues to process records from shards in sequence as it does today, and handles records with a null partition key without modification.

The KCL itself requires no changes to consume from a service-managed stream, and you do not need to upgrade the KCL. However, if a producer writes KPL-aggregated records to a service-managed stream (for example, because aggregation was not disabled), the KCL de-aggregation path can silently drop those records, because the aggregation contract does not hold when the service places records by its own algorithm. Ensure producers disable aggregation, or upgrade the KPL and grant it the `kinesis:DescribeStreamSummary` permission, so that aggregated records are never written to a service-managed stream.

Summary

KPL and KCL compatibility summary

Scenario	Action required	Details
Send records with null partition key	Upgrade KPL	Upgrade to the latest KPL version to support sending records without a partition key
Send records with partition key (existing behavior)	No changes	Existing KPL works as-is. The service ignores the partition key on service-managed streams
KCL consumers	No changes	KCL works as-is regardless of how records were produced
Upgraded KPL producers	Grant `kinesis:DescribeStreamSummary`	The updated KPL calls `DescribeStreamSummary` to detect service-managed streams. Without this permission, the KPL falls back to aggregating. Aggregation is not supported on service-managed streams, so the aggregated records that are written can be silently

Scenario	Action required	Details
		dropped by KCL consumers during de-aggregation.
KPL aggregation on service-managed streams	Not supported	Aggregation is not supported on service-managed streams in any KPL version. Aggregated records that are written can be silently dropped by KCL consumers during de-aggregation. Disable aggregation or upgrade the KPL.

Dependency integration case studies

Service-managed record distribution works with AWS services that produce to or consume from Amazon Kinesis Data Streams. This section describes how common AWS service integrations behave when a stream uses service-managed record distribution, and any changes you need to make.

AWS Lambda triggers with service-managed streams

AWS Lambda event source mappings work with service-managed streams without modification. Lambda continues to poll shards and invoke your function with batches of records. The `partitionKey` field in the event record may be `null` for records written without a partition key. Ensure your Lambda function handles null partition keys gracefully.

Amazon Data Firehose consuming from service-managed streams

Amazon Data Firehose delivery streams that source data from a Kinesis Data Streams stream configured with service-managed record distribution continue to work without changes. Firehose reads records from shards and delivers them to destinations regardless of how records were distributed across shards. No configuration changes are required.

When records are written to a service-managed stream without a partition key, Amazon Data Firehose surfaces the partition key as an *empty string* (`""`) rather than `null` in the following customer-facing cases:

- **Transformation Lambda function input** – If you use a Lambda function to transform records, the `partitionKey` field in the Lambda input event is an empty string for records written without a partition key. This avoids compatibility issues with common parsing libraries (such as AWS Lambda Powertools) that perform non-null validation on the `partitionKey` field.
- **Snowflake destination metadata** – If you deliver to a Snowflake destination, the partition key metadata column contains an empty string for records written without a partition key.

> **Note**
>
> If you opt into service-managed record distribution and do not provide partition keys, expect an empty string ("") rather than your previously supplied partition key value in these locations. Do not rely on the partition key value for business logic when using service-managed record distribution.

Amazon Managed Service for Apache Flink consuming from service-managed streams

Amazon Managed Service for Apache Flink is a hosting platform that runs your Flink application; the Kinesis connector you choose determines compatibility with service-managed streams.

- **Current connector (`flink-connector-aws-kinesis-streams`)** – Consumes from service-managed streams without modification, including records written with a null partition key. If your application uses partition keys for keyed state or windowing, be aware that partition keys may be `null`; review your logic to ensure it does not depend on partition key values for processing correctness.
- **Legacy connector (`flink-connector-kinesis`)** – Not recommended for service-managed streams. The legacy connector uses the KPL with aggregation, and aggregation is not supported on service-managed streams. In addition, a custom deserializer that reads the partition key must guard against `null`. If you use the legacy connector, migrate to the current connector before enabling service-managed record distribution.

CloudWatch metric streams with service-managed streams

Amazon CloudWatch metric streams can deliver metric data to a Kinesis Data Streams stream. The metric stream producer sets a partition key, and its consumer reads the partition key back

for deduplication and routing. On a service-managed stream, the partition key is ignored only for shard placement and is still returned to consumers as supplied, so deduplication logic that reads the key back continues to work. No changes are required. Note that service-managed record distribution is available only for on-demand streams; metric streams that use provisioned streams continue to use user-managed partition keys.

CloudFront real-time logs to service-managed streams

Amazon CloudFront can publish real-time logs directly to an Kinesis Data Streams stream through its native integration. Amazon CloudFront produces log records to the stream; it does not use the KPL and has no aggregation logic that depends on associating a partition key with a specific shard. Although the records Amazon CloudFront generates include a partition key, the value is not used for routing on a service-managed stream. Enabling service-managed record distribution on a stream that receives Amazon CloudFront real-time logs requires no changes and has no impact on log delivery. For more information about Amazon CloudFront real-time logs, see Real-time logs in the *Amazon CloudFront Developer Guide*.

Amazon EMR with service-managed streams

Amazon EMR uses the Spark connector to read from and write to Kinesis Data Streams streams. The connector has no logic that depends on the partition key value for reading or writing data. When Amazon EMR consumes from a service-managed stream, no changes are required.

> **Important**
>
> When Amazon EMR *produces* to a stream, the Spark connector writes through the KPL with aggregation enabled. Aggregation is not supported on service-managed streams. An Amazon EMR producer that uses a Spark connector with aggregation enabled can still write to a service-managed stream, but the aggregated records can be silently dropped by KCL consumers during de-aggregation, with no error surfaced by the producer or consumer. To produce to a service-managed stream from Amazon EMR safely, use a connector version that lets you disable KPL aggregation, or disable aggregation in your job configuration. For more information, see KPL and KCL with service-managed streams.

Amazon EventBridge Pipes with service-managed streams

Amazon EventBridge Pipes can use a Kinesis Data Streams stream as a target (Pipes writes records with the `PutRecords` API) and as a source (a Pipes poller reads records and forwards them). EventBridge Pipes uses the AWS SDK directly, without the KPL or KCL and without aggregation.

- **Service-managed stream as a target** – Works without changes. EventBridge Pipes always sends a non-null partition key (the configured value, or the pipe execution ID). On a service-managed stream, the supplied key is ignored for routing and records are distributed evenly across shards.
- **Service-managed stream as a source** – EventBridge Pipes consumes records from a service-managed stream, including records written without a partition key. Ensure any downstream targets or enrichment steps in your pipe do not require a partition key value.

Amazon DynamoDB with Kinesis data streams

You can stream Amazon DynamoDB change data capture (CDC) records to a Kinesis Data Streams stream that you own. DynamoDB always sets a non-null partition key (a hash of the item's key) and never reads records back from the stream. On a service-managed stream, DynamoDB continues to work without changes.

Be aware of one behavior change: on a user-managed stream, all change records for a single item map to the same shard because they share a partition key. On a service-managed stream, the partition key is ignored for routing, so change records for a single item are distributed across shards. DynamoDB makes no ordering guarantee based on shard placement. Consumers order records by using the `ApproximateCreationDateTime` field, so this behavior does not affect correctness for consumers that order records by timestamp.

Amazon Redshift streaming ingestion from service-managed streams

Amazon Redshift streaming ingestion lets you create a materialized view over a Kinesis Data Streams stream. The materialized view schema includes a partition key column. When you ingest from a service-managed stream, records written without a partition key appear with a blank partition key value in the materialized view. No changes are required, and both null and non-null partition key records are ingested successfully.

Amazon Kinesis Agent with service-managed streams

The Amazon Kinesis Agent is a producer-only application that reads text files and writes records to a stream with the `PutRecords` API. It always sets a non-null partition key (a random value by default, or an MD5 hash of the record) and uses the AWS SDK directly without the KPL or aggregation. On a service-managed stream, the supplied partition key is ignored for routing and the agent's writes flow through unchanged. No changes are required.

Amazon EventBridge event bus with service-managed streams

An Amazon EventBridge event bus can deliver events to a Kinesis Data Streams stream as a target and consume from a stream as a source. As a producer, an event bus always sends a non-null partition key (it generates a random value when no Kinesis parameters are configured) and uses the AWS SDK directly without the KPL or aggregation. As a consumer, it has no logic that depends on the partition key value or shard placement. On a service-managed stream, the supplied partition key is ignored for routing and null partition key records are consumed cleanly. No changes are required.

Streaming tables and Amazon S3 delivery with service-managed streams

You can deliver records from a Kinesis Data Streams stream to streaming tables and to Amazon Simple Storage Service (Amazon S3) delivery destinations. On a service-managed stream, records written without a partition key are delivered normally, along with records that include a partition key. No changes are required.

Write data to Amazon Kinesis Data Streams

A *producer* is an application that writes data to Amazon Kinesis Data Streams. You can build producers for Kinesis Data Streams using the AWS SDK for Java and the Kinesis Producer Library (KPL).

If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in What is Amazon Kinesis Data Streams? and Use the AWS CLI to perform Amazon Kinesis Data Streams operations.

> **Important**
>
> Kinesis Data Streams supports changes to the data record retention period of your data stream. For more information, see Change the data retention period.

To put data into the stream, you must specify the name of the stream, a partition key, and the data blob to be added to the stream. The partition key is used to determine which shard in the stream the data record is added to.

All the data in the shard is sent to the same worker that is processing the shard. Which partition key you use depends on your application logic. The number of partition keys should typically be much greater than the number of shards. This is because the partition key is used to determine how to map a data record to a particular shard. If you have enough partition keys, the data can be evenly distributed across the shards in a stream.

Topics

- Develop producers using the Amazon Kinesis Producer Library (KPL)
- Develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java
- Write to Amazon Kinesis Data Streams using Kinesis Agent
- Write to Kinesis Data Streams using other AWS services
- Write to Kinesis Data Streams using third-party integrations
- Troubleshoot Amazon Kinesis Data Streams producers
- Optimize Kinesis Data Streams producers

Develop producers using the Amazon Kinesis Producer Library (KPL)

An Amazon Kinesis Data Streams producer is an application that puts user data records into a Kinesis data stream (also called *data ingestion*). The Amazon Kinesis Producer Library (KPL) simplifies producer application development, letting developers achieve high write throughput to a Kinesis data stream.

You can monitor the KPL with Amazon CloudWatch. For more information, see Monitor the Kinesis Producer Library with Amazon CloudWatch.

Topics

- Review the role of the KPL
- Realize the advantages of using the KPL
- Understand when not to use the KPL
- Install the KPL
- Migrate from KPL 0.x to KPL 1.x
- Transition to Amazon Trust Services (ATS) certificates for the KPL
- KPL supported platforms
- KPL key concepts
- Integrate the KPL with producer code
- Write to your Kinesis data stream using the KPL
- Configure the Amazon Kinesis Producer Library
- Implement consumer de-aggregation
- Use the KPL with Amazon Data Firehose
- Use the KPL with the AWS Glue Schema Registry
- Configure the KPL proxy configuration
- KPL version lifecycle policy

> **Note**
>
> It is recommended that you upgrade to the latest KPL version. KPL is regularly updated with newer releases that include the latest dependency and security patches, bug fixes,

and backward-compatible new features. For more information, see https://github.com/awslabs/amazon-kinesis-producer/releases/.

Review the role of the KPL

The KPL is an easy-to-use, highly configurable library that helps you write to a Kinesis data stream. It acts as an intermediary between your producer application code and the Kinesis Data Streams API actions. The KPL performs the following primary tasks:

- Writes to one or more Kinesis data streams with an automatic and configurable retry mechanism
- Collects records and uses `PutRecords` to write multiple records to multiple shards per request
- Aggregates user records to increase payload size and improve throughput
- Integrates seamlessly with the Kinesis Client Library (KCL) to de-aggregate batched records on the consumer
- Submits Amazon CloudWatch metrics on your behalf to provide visibility into producer performance

Note that the KPL is different from the Kinesis Data Streams API that is available in the AWS SDKs. The Kinesis Data Streams API helps you manage many aspects of Kinesis Data Streams (including creating streams, resharding, and putting and getting records), while the KPL provides a layer of abstraction specifically for ingesting data. For information about the Kinesis Data Streams API, see the Amazon Kinesis API Reference.

Realize the advantages of using the KPL

The following list represents some of the major advantages to using the KPL for developing Kinesis Data Streams producers.

The KPL can be used in either synchronous or asynchronous use cases. We suggest using the higher performance of the asynchronous interface unless there is a specific reason to use synchronous behavior. For more information about these two use cases and example code, see Write to your Kinesis data stream using the KPL.

Performance Benefits

The KPL can help build high-performance producers. Consider a situation where your Amazon EC2 instances serve as a proxy for collecting 100-byte events from hundreds or thousands of

low power devices and writing records into a Kinesis data stream. These EC2 instances must each write thousands of events per second to your data stream. To achieve the throughput needed, producers must implement complicated logic, such as batching or multithreading, in addition to retry logic and record de-aggregation at the consumer side. The KPL performs all of these tasks for you.

Consumer-Side Ease of Use

For consumer-side developers using the KCL in Java, the KPL integrates without additional effort. When the KCL retrieves an aggregated Kinesis Data Streams record consisting of multiple KPL user records, it automatically invokes the KPL to extract the individual user records before returning them to the user.

For consumer-side developers who do not use the KCL but instead use the API operation `GetRecords` directly, a KPL Java library is available to extract the individual user records before returning them to the user.

Producer Monitoring

You can collect, monitor, and analyze your Kinesis Data Streams producers using Amazon CloudWatch and the KPL. The KPL emits throughput, error, and other metrics to CloudWatch on your behalf, and is configurable to monitor at the stream, shard, or producer level.

Asynchronous Architecture

Because the KPL may buffer records before sending them to Kinesis Data Streams, it does not force the caller application to block and wait for a confirmation that the record has arrived at the server before continuing runtime. A call to put a record into the KPL always returns immediately and does not wait for the record to be sent or a response to be received from the server. Instead, a `Future` object is created that receives the result of sending the record to Kinesis Data Streams at a later time. This is the same behavior as asynchronous clients in the AWS SDK.

Understand when not to use the KPL

The KPL can incur an additional processing delay of up to `RecordMaxBufferedTime` within the library (user-configurable). Larger values of `RecordMaxBufferedTime` results in higher packing efficiencies and better performance. Applications that cannot tolerate this additional delay might need to use the AWS SDK directly. For more information about using the AWS SDK with Kinesis Data Streams, see Develop producers using the Amazon Kinesis Data Streams API with the AWS

SDK for Java. For more information about `RecordMaxBufferedTime` and other user-configurable properties of the KPL, see Configure the Amazon Kinesis Producer Library.

Install the KPL

Amazon provides pre-built binaries of the C++ Amazon Kinesis Producer Library (KPL) for macOS, Windows, and recent Linux distributions (for supported platform details, see the next section). These binaries are packaged as part of Java .jar files and are automatically invoked and used if you are using Maven to install the package. To locate the latest versions of the KPL and KCL, use the following Maven search links:

- KPL
- KCL

The Linux binaries have been compiled with the GNU Compiler Collection (GCC) and statically linked against libstdc++ on Linux. They are expected to work on any 64-bit Linux distribution that includes a glibc version 2.5 or higher.

Users of earlier Linux distributions can build the KPL using the build instructions provided along with the source on GitHub. To download the KPL from GitHub, see Amazon Kinesis Producer Library.

> **Important**
>
> Amazon Kinesis Producer Library (KPL) 0.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KPL applications using version 0.x to the latest KPL version before January 30, 2026. To find the latest KPL version, see the KPL page on Github. For information about migrating from KPL 0.x to KPL 1.x, see Migrate from KPL 0.x to KPL 1.x.

Migrate from KPL 0.x to KPL 1.x

This topic provides step-by-step instructions to migrate your consumer from KPL 0.x to KPL 1.x. KPL 1.x introduces support for the AWS SDK for Java 2.x while maintaining interface compatibility with previous versions. You don't have to update your core data processing logic to migrate to KPL 1.x.

1. **Make sure that you have the following prerequisites:**

   - Java Development Kit (JDK) 8 or later
   - AWS SDK for Java 2.x
   - Maven or Gradle for dependency management

2. **Add dependencies**

   If you're using Maven, add the following dependency to your pom.xml file. Make sure you updated the groupId from `com.amazonaws` to `software.amazon.kinesis` and the version `1.x.x` to the latest KPL version.

   ```
   <dependency>
       <groupId>software.amazon.kinesis</groupId>
       <artifactId>amazon-kinesis-producer</artifactId>
       <version>1.x.x</version> <!-- Use the latest version -->
   </dependency>
   ```

   If you're using Gradle, add the following to your `build.gradle` file. Make sure to replace `1.x.x` with the latest KPL version.

   ```
   implementation 'software.amazon.kinesis:amazon-kinesis-producer:1.x.x'
   ```

   You can check for the latest version of the KPL on the Maven Central Repository.

3. **Update import statements for KPL**

   KPL 1.x uses the AWS SDK for Java 2.x and uses an updated package name that starts with `software.amazon.kinesis`, compared to the package name in the previous KPL that starts with `com.amazonaws.services.kinesis`.

   Replace the import for `com.amazonaws.services.kinesis` with `software.amazon.kinesis`. The following table lists the imports that you must replace.

   Import replacements

Replace:	With:
import com.amazonaws.services.kinesis.producer.Attempt;	import software.amazon.kinesis.producer.Attempt;

Replace:	With:
import com.amazonaws.services.kinesis.producer.BinaryToHexConverter;	import software.amazon.kinesis.producer.BinaryToHexConverter;
import com.amazonaws.services.kinesis.producer.CertificateExtractor;	import software.amazon.kinesis.producer.CertificateExtractor;
import com.amazonaws.services.kinesis.producer.Daemon;	import software.amazon.kinesis.producer.Daemon;
import com.amazonaws.services.kinesis.producer.DaemonException;	import software.amazon.kinesis.producer.DaemonException;
import com.amazonaws.services.kinesis.producer.FileAgeManager;	import software.amazon.kinesis.producer.FileAgeManager;
import com.amazonaws.services.kinesis.producer.FutureTimedOutException;	import software.amazon.kinesis.producer.FutureTimedOutException;
import com.amazonaws.services.kinesis.producer.GlueSchemaRegistrySerializerInstance;	import software.amazon.kinesis.producer.GlueSchemaRegistrySerializerInstance;
import com.amazonaws.services.kinesis.producer.HashedFileCopier;	import software.amazon.kinesis.producer.HashedFileCopier;
import com.amazonaws.services.kinesis.producer.IKinesisProducer;	import software.amazon.kinesis.producer.IKinesisProducer;
import com.amazonaws.services.kinesis.producer.IrrecoverableError;	import software.amazon.kinesis.producer.IrrecoverableError;
import com.amazonaws.services.kinesis.producer.KinesisProducer;	import software.amazon.kinesis.producer.KinesisProducer;
import com.amazonaws.services.kinesis.producer.KinesisProducerConfiguration;	import software.amazon.kinesis.producer.KinesisProducerConfiguration;

Replace:	With:
import com.amazonaws.services.kinesis.producer.LogInputStreamReader;	import software.amazon.kinesis.producer.LogInputStreamReader;
import com.amazonaws.services.kinesis.producer.Metric;	import software.amazon.kinesis.producer.Metric;
import com.amazonaws.services.kinesis.producer.ProcessFailureBehavior;	import software.amazon.kinesis.producer.ProcessFailureBehavior;
import com.amazonaws.services.kinesis.producer.UnexpectedMessageException;	import software.amazon.kinesis.producer.UnexpectedMessageException;
import com.amazonaws.services.kinesis.producer.UserRecord;	import software.amazon.kinesis.producer.UserRecord;
import com.amazonaws.services.kinesis.producer.UserRecordFailedException;	import software.amazon.kinesis.producer.UserRecordFailedException;
import com.amazonaws.services.kinesis.producer.UserRecordResult;	import software.amazon.kinesis.producer.UserRecordResult;
import com.amazonaws.services.kinesis.producer.protobuf.Messages;	import software.amazon.kinesis.producer.protobuf.Messages;
import com.amazonaws.services.kinesis.producer.protobuf.Config;	import software.amazon.kinesis.producer.protobuf.Config;

4. **Update import statements for AWS credentials provider classes**

   When migrating to KPL 1.x, you must update packages and classes in your imports in your KPL application code that are based on the AWS SDK for Java 1.x to corresponding ones based on the AWS SDK for Java 2.x. Common imports in the KPL application are credentials provider classes. See Credentials provider changes in the AWS SDK for Java 2.x migration guide documentation for the full list of credentials provider changes. Here is the common import change that you might need to make in your KPL applications.

   Import in KPL 0.x

```
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
```

Import in KPL 1.x

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
```

If you import any other credentials providers based on the AWS SDK for Java 1.x, you must update them to the AWS SDK for Java 2.x equivalent ones. If you didn't import any classes/packages from the AWS SDK for Java 1.x, you can ignore this step.

5. **Update the credentials provider configuration in the KPL configuration**

   The credentials provider configuration in KPL 1.x requires the AWS SDK for Java 2.x credential providers. If you are passing credentials providers for the AWS SDK for Java 1.x in the `KinesisProducerConfiguration` by overriding the default credentials provider, you must update it with the AWS SDK for Java 2.x credential providers. See Credentials provider changes in the AWS SDK for Java 2.x migration guide documentation for the full list of credentials provider changes. If you didn't override the default credentials provider in the KPL configuration, you can ignore this step.

   For example, if you are overriding the default credentials provider for the KPL with the following code:

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration();
// SDK v1 default credentials provider
config.setCredentialsProvider(new DefaultAWSCredentialsProviderChain());
```

   You must update them with the following code to use the AWS SDK for Java 2.x credentials provider:

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration();
// New SDK v2 default credentials provider
config.setCredentialsProvider(DefaultCredentialsProvider.create());
```

Transition to Amazon Trust Services (ATS) certificates for the KPL

On February 9, 2018, at 9:00 AM PST, Amazon Kinesis Data Streams installed ATS certificates. To continue to be able to write records to Kinesis Data Streams using the Amazon Kinesis Producer Library (KPL), you must upgrade your installation of the KPL to version 0.12.6 or later. This change affects all AWS Regions.

For information about the move to ATS, see How to Prepare for AWS's Move to Its Own Certificate Authority.

If you encounter problems and need technical support, create a case with the AWS Support Center.

KPL supported platforms

The Amazon Kinesis Producer Library (KPL) is written in C++ and runs as a child process to the main user process. Precompiled 64-bit native binaries are bundled with the Java release and are managed by the Java wrapper.

The Java package runs without the need to install any additional libraries on the following operating systems:

- Linux distributions with kernel 2.6.18 (September 2006) and later
- Apple iOS X 10.9 and later
- Windows Server 2008 and later

> **Important**
>
> Windows Server 2008 and later is supported for all KPL versions up to version 0.14.0. The Windows platform is NOT supported starting with KPL version 0.14.0 or higher.

Note that the KPL is 64-bit only.

Source code

If the binaries provided in the KPL installation are not sufficient for your environment, the core of the KPL is written as a C++ module. The source code for the C++ module and the Java interface are released under the Amazon Public License and are available on GitHub at Amazon Kinesis Producer

Library. Although the KPL can be used on any platform for which a recent standards-compliant C++ compiler and JRE are available, Amazon doesn't officially support any platform that is not on the supported platforms list.

KPL key concepts

The following sections contain concepts and terminology necessary to understand and benefit from the Amazon Kinesis Producer Library (KPL).

Topics

- Records
- Batching
- Aggregation
- Collection

Records

In this guide, we distinguish between *KPL user records* and *Kinesis Data Streams records*. When we use the term *record* without a qualifier, we refer to a *KPL user record*. When we refer to a Kinesis Data Streams record, we explicitly say *Kinesis Data Streams record*.

A KPL user record is a blob of data that has particular meaning to the user. Examples include a JSON blob representing a UI event on a website, or a log entry from a web server.

A Kinesis Data Streams record is an instance of the `Record` data structure defined by the Kinesis Data Streams service API. It contains a partition key, sequence number, and a blob of data.

Batching

Batching refers to performing a single action on multiple items instead of repeatedly performing the action on each individual item.

In this context, the "item" is a record, and the action is sending it to Kinesis Data Streams. In a non-batching situation, you would place each record in a separate Kinesis Data Streams record and make one HTTP request to send it to Kinesis Data Streams. With batching, each HTTP request can carry multiple records instead of just one.

The KPL supports two types of batching:

- *Aggregation* – Storing multiple records within a single Kinesis Data Streams record.
- *Collection* – Using the API operation `PutRecords` to send multiple Kinesis Data Streams records to one or more shards in your Kinesis data stream.

The two types of KPL batching are designed to coexist and can be turned on or off independently of one another. By default, both are turned on.

Aggregation

Aggregation refers to the storage of multiple records in a Kinesis Data Streams record. Aggregation allows customers to increase the number of records sent per API call, which effectively increases producer throughput.

Kinesis Data Streams shards support up to 1,000 Kinesis Data Streams records per second, or 1 MB throughput. The Kinesis Data Streams records per second limit binds customers with records smaller than 1 KB. Record aggregation allows customers to combine multiple records into a single Kinesis Data Streams record. This allows customers to improve their per shard throughput.

Consider the case of one shard in Region eusc-de-east-1 that is currently running at a constant rate of 1,000 records per second, with records that are 512 bytes each. With KPL aggregation, you can pack 1,000 records into only 10 Kinesis Data Streams records, reducing the RPS to 10 (at 50 KB each).

Collection

Collection refers to batching multiple Kinesis Data Streams records and sending them in a single HTTP request with a call to the API operation `PutRecords`, instead of sending each Kinesis Data Streams record in its own HTTP request.

This increases throughput compared to using no collection because it reduces the overhead of making many separate HTTP requests. In fact, `PutRecords` itself was specifically designed for this purpose.

Collection differs from aggregation in that it is working with groups of Kinesis Data Streams records. The Kinesis Data Streams records being collected can still contain multiple records from the user. The relationship can be visualized as such:

```
record 0 --|
```

```
record 1   |        [ Aggregation ]
    ...    |--> Amazon Kinesis record 0 --|
    ...    |                              |
record A --|                              |
                                           |
    ...                  ...               |
                                           |
record K --|                              |
record L   |                              |      [ Collection ]
    ...    |--> Amazon Kinesis record C --|--> PutRecords Request
    ...    |                              |
record S --|                              |
                                           |
    ...                  ...               |
                                           |
record AA--|                              |
record BB  |                              |
    ...    |--> Amazon Kinesis record M --|
    ...    |
record ZZ--|
```

Integrate the KPL with producer code

The Amazon Kinesis Producer Library (KPL) runs in a separate process, and communicates with your parent user process using IPC. This architecture is sometimes called a microservice, and is chosen for two main reasons:

1) Your user process will not crash even if the KPL crashes

Your process could have tasks unrelated to Kinesis Data Streams, and may be able to continue operation even if the KPL crashes. It is also possible for your parent user process to restart the KPL and recover to a fully working state (this functionality is in the official wrappers).

An example is a web server that sends metrics to Kinesis Data Streams; the server can continue serving pages even if the Kinesis Data Streams part has stopped working. Crashing the whole server because of a bug in the KPL would therefore cause an unnecessary outage.

2) Arbitrary clients can be supported

There are always customers who use languages other than the ones officially supported. These customers should also be able to use the KPL easily.

Recommended usage matrix

The following usage matrix lists the recommended settings for different users and advises you about whether and how you should use the KPL. Keep in mind that if aggregation is enabled, de-aggregation must also be used to extract your records on the consumer side.

Producer side language	Consumer side language	KCL Version	Checkpoint logic	Can you use the KPL?	Caveats
Anything but Java	*	*	*	No	N/A
Java	Java	Uses Java SDK directly	N/A	Yes	If aggregation is used, you have to use the provided de-aggregation library after `GetRecords` calls.
Java	Anything but Java	Uses SDK directly	N/A	Yes	Must disable aggregation.
Java	Java	1.3.x	N/A	Yes	Must disable aggregation.
Java	Java	1.4.x	Calls checkpoint without any arguments	Yes	None
Java	Java	1.4.x	Calls checkpoint with an explicit	Yes	Either disable aggregation, or change the code to

Producer side language	Consumer side language	KCL Version	Checkpoint logic	Can you use the KPL?	Caveats
			sequence number		use extended sequence numbers for checkpoin ting.
Java	Anything but Java	1.3.x + Multilang uage daemon + language-specific wrapper	N/A	Yes	Must disable aggregation.

Write to your Kinesis data stream using the KPL

The following sections show sample code in a progression from the most basic producer to fully asynchronous code.

Barebones producer code

The following code is all that is needed to write a minimal working producer. The Amazon Kinesis Producer Library (KPL) user records are processed in the background.

```
// KinesisProducer gets credentials automatically like
// DefaultAWSCredentialsProviderChain.
// It also gets region automatically from the EC2 metadata service.
KinesisProducer kinesis = new KinesisProducer();
// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}
// Do other stuff ...
```

Respond to results synchronously

In the previous example, the code didn't check whether the KPL user records succeeded. The KPL performs any retries needed to account for failures. But if you want to check on the results, you can examine them using the `Future` objects that are returned from `addUserRecord`, as in the following example (previous example shown for context):

```
KinesisProducer kinesis = new KinesisProducer();

// Put some records and save the Futures
List<Future<UserRecordResult>> putFutures = new
 LinkedList<Future<UserRecordResult>>();
for (int i = 0; i < 100; i++) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    putFutures.add(
        kinesis.addUserRecord("myStream", "myPartitionKey", data));
}

// Wait for puts to finish and check the results
for (Future<UserRecordResult> f : putFutures) {
    UserRecordResult result = f.get(); // this does block
    if (result.isSuccessful()) {
        System.out.println("Put record into shard " +
                           result.getShardId());
    } else {
        for (Attempt attempt : result.getAttempts()) {
            // Analyze and respond to the failure
        }
    }
}
```

Respond to results asynchronously

The previous example is calling `get()` on a `Future` object, which blocks runtime. If you don't want to block runtime, you can use an asynchronous callback, as shown in the following example:

```
KinesisProducer kinesis = new KinesisProducer();

FutureCallback<UserRecordResult> myCallback = new FutureCallback<UserRecordResult>() {

    @Override public void onFailure(Throwable t) {
```

```
        /* Analyze and respond to the failure  */
    };
    @Override public void onSuccess(UserRecordResult result) {
        /* Respond to the success */
    };
};

for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    ListenableFuture<UserRecordResult> f = kinesis.addUserRecord("myStream",
 "myPartitionKey", data);
    // If the Future is complete by the time we call addCallback, the callback will be
 invoked immediately.
    Futures.addCallback(f, myCallback);
}
```

Configure the Amazon Kinesis Producer Library

Although the default settings should work well for most use cases, you may want to change some of the default settings to tailor the behavior of the `KinesisProducer` to your needs. An instance of the `KinesisProducerConfiguration` class can be passed to the `KinesisProducer` constructor to do so, for example:

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration()
        .setRecordMaxBufferedTime(3000)
        .setMaxConnections(1)
        .setRequestTimeout(60000)
        .setRegion("us-west-1");

final KinesisProducer kinesisProducer = new KinesisProducer(config);
```

You can also load a configuration from a properties file:

```
KinesisProducerConfiguration config =
 KinesisProducerConfiguration.fromPropertiesFile("default_config.properties");
```

You can substitute any path and file name that the user process has access to. You can additionally call set methods on the `KinesisProducerConfiguration` instance created this way to customize the config.

The properties file should specify parameters using their names in PascalCase. The names match those used in the set methods in the `KinesisProducerConfiguration` class. For example:

```
RecordMaxBufferedTime = 100
MaxConnections = 4
RequestTimeout = 6000
Region = us-west-1
```

For more information about configuration parameter usage rules and value limits, see the sample configuration properties file on GitHub.

Note that after `KinesisProducer` is initialized, changing the `KinesisProducerConfiguration` instance that was used has no further effect. `KinesisProducer` does not currently support dynamic reconfiguration.

Implement consumer de-aggregation

Beginning with release 1.4.0, the KCL supports automatic de-aggregation of KPL user records. Consumer application code written with previous versions of the KCL will compile without any modification after you update the KCL. However, if KPL aggregation is being used on the producer side, there is a subtlety involving checkpointing: all subrecords within an aggregated record have the same sequence number, so additional data has to be stored with the checkpoint if you need to distinguish between subrecords. This additional data is referred to as the *subsequence number*.

Options

- Migrate from previous versions of the KCL
- Use KCL extensions for KPL de-aggregation
- Use GetRecords directly

Migrate from previous versions of the KCL

You are not required to change your existing calls to do checkpointing with aggregation. It is still guaranteed that you can retrieve all records successfully stored in Kinesis Data Streams. The KCL now provides two new checkpoint operations to support particular use cases, described following.

If your existing code was written for the KCL before KPL support, and your checkpoint operation is called without arguments, it is equivalent to checkpointing the sequence number of the last KPL user record in the batch. If your checkpoint operation is called with a sequence number string, it

is equivalent to checkpointing the given sequence number of the batch along with the implicit subsequence number 0 (zero).

Calling the new KCL checkpoint operation `checkpoint()` without any arguments is semantically equivalent to checkpointing the sequence number of the last `Record` call in the batch, along with the implicit subsequence number 0 (zero).

Calling the new KCL checkpoint operation `checkpoint(Record record)` is semantically equivalent to checkpointing the given `Record`'s sequence number along with the implicit subsequence number 0 (zero). If the `Record` call is actually a `UserRecord`, the `UserRecord` sequence number and subsequence number are checkpointed.

Calling the new KCL checkpoint operation `checkpoint(String sequenceNumber, long subSequenceNumber)` explicitly checkpoints the given sequence number along with the given subsequence number.

In any of these cases, after the checkpoint is stored in the Amazon DynamoDB checkpoint table, the KCL can correctly resume retrieving records even when the application crashes and restarts. If more records are contained within the sequence, retrieval occurs starting with the next subsequence number record within the record with the most recently checkpointed sequence number. If the most recent checkpoint included the very last subsequence number of the previous sequence number record, retrieval occurs starting with the record with the next sequence number.

The next section discusses details of sequence and subsequence checkpointing for consumers that must avoid skipping and duplication of records. If skipping (or duplication) of records when stopping and restarting your consumer's record processing is not important, you can run your existing code with no modification.

Use KCL extensions for KPL de-aggregation

KPL de-aggregation can involve subsequence checkpointing. To facilitate using subsequence checkpointing, a `UserRecord` class has been added to the KCL:

```
public class UserRecord extends Record {
    public long getSubSequenceNumber() {
    /* ... */
    }
    @Override
    public int hashCode() {
    /* contract-satisfying implementation */
```

```
    }
    @Override
    public boolean equals(Object obj) {
    /* contract-satisfying implementation */
    }
}
```

This class is now used instead of `Record`. This does not break existing code because it is a subclass of `Record`. The `UserRecord` class represents both actual subrecords and standard, non-aggregated records. Non-aggregated records can be thought of as aggregated records with exactly one subrecord.

In addition, two new operations are added to`IRecordProcessorCheckpointer`:

```
public void checkpoint(Record record);
public void checkpoint(String sequenceNumber, long subSequenceNumber);
```

To begin using subsequence number checkpointing, you can perform the following conversion. Change the following form code:

```
checkpointer.checkpoint(record.getSequenceNumber());
```

New form code:

```
checkpointer.checkpoint(record);
```

We recommend that you use the `checkpoint(Record record)` form for subsequence checkpointing. However, if you are already storing `sequenceNumbers` in strings to use for checkpointing, you should now also store `subSequenceNumber`, as shown in the following example:

```
String sequenceNumber = record.getSequenceNumber();
long subSequenceNumber = ((UserRecord) record).getSubSequenceNumber();  // ... do other
 processing
checkpointer.checkpoint(sequenceNumber, subSequenceNumber);
```

The cast from `Record`to`UserRecord` always succeeds because the implementation always uses `UserRecord`. Unless there is a need to perform arithmetic on the sequence numbers, this approach is not recommended.

While processing KPL user records, the KCL writes the subsequence number into Amazon DynamoDB as an extra field for each row. Previous versions of the KCL used `AFTER_SEQUENCE_NUMBER` to fetch records when resuming checkpoints. The current KCL with KPL support uses `AT_SEQUENCE_NUMBER` instead. When the record at the checkpointed sequence number is retrieved, the checkpointed subsequence number is checked, and subrecords are dropped as appropriate (which may be all of them, if the last subrecord is the one checkpointed). Again, non-aggregated records can be thought of as aggregated records with a single subrecord, so the same algorithm works for both aggregated and non-aggregated records.

Use GetRecords directly

You can also choose not to use the KCL but instead invoke the API operation `GetRecords` directly to retrieve Kinesis Data Streams records. To unpack these retrieved records into your original KPL user records, call one of the following static operations in `UserRecord.java`:

```
public static List<Record> deaggregate(List<Record> records)

public static List<UserRecord> deaggregate(List<UserRecord> records, BigInteger
 startingHashKey, BigInteger endingHashKey)
```

The first operation uses the default value `0` (zero) for `startingHashKey` and the default value `2^128 -1` for `endingHashKey`.

Each of these operations de-aggregates the given list of Kinesis Data Streams records into a list of KPL user records. Any KPL user records whose explicit hash key or partition key falls outside the range of the `startingHashKey` (inclusive) and the `endingHashKey` (inclusive) are discarded from the returned list of records.

Use the KPL with Amazon Data Firehose

If you use the Kinesis Producer Library (KPL) to write data to a Kinesis data stream, you can use aggregation to combine the records that you write to that Kinesis data stream. If you then use that data stream as a source for your Firehose delivery stream, Firehose de-aggregates the records before it delivers them to the destination. If you configure your delivery stream to transform the data, Firehose de-aggregates the records before it delivers them to AWS Lambda. For more information, see Writing to Amazon Firehose Using Kinesis Data Streams.

Use the KPL with the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue Schema Registry. The AWS Glue Schema Registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the KPL and Kinesis Client Library (KCL) libraries in Java.

> **Important**
>
> Currently, Kinesis Data Streams and AWS Glue schema registry integration is only supported for the Kinesis data streams that use KPL producers implemented in Java. Multi-language support is not provided.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the KPL, see the "Interacting with Data Using the KPL/KCL Libraries" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Configure the KPL proxy configuration

For applications that cannot directly connect to the internet, all AWS SDK clients support the use of HTTP or HTTPS proxies. In a typical enterprise environment, all outbound network traffic has to go through proxy servers. If your application uses Kinesis Producer Library (KPL) to collect and send data to AWS in an environment that uses proxy servers, your application will require KPL proxy configuration. KPL is a high level library built on top of the AWS Kinesis SDK. It is split into a native process and a wrapper. The native process performs all of the jobs of processing and sending records, while the wrapper manages the native process and communicates with it. For more information, see Implementing Efficient and Reliable Producers with the Amazon Kinesis Producer Library.

The wrapper is written in Java and the native process is written in C++ with the use of Kinesis SDK. KPL version 0.14.7 and higher now supports proxy configuration in the Java wrapper which can pass all proxy configurations to the native process. For more information, see https://github.com/awslabs/amazon-kinesis-producer/releases/tag/v0.14.7.

You can use the following code to add proxy configurations to your KPL applications.

```
KinesisProducerConfiguration configuration = new KinesisProducerConfiguration();
// Next 4 lines used to configure proxy
configuration.setProxyHost("10.0.0.0"); // required
configuration.setProxyPort(3128); // default port is set to 443
configuration.setProxyUserName("username"); // no default
configuration.setProxyPassword("password"); // no default

KinesisProducer kinesisProducer = new KinesisProducer(configuration);
```

KPL version lifecycle policy

This topic outlines the version lifecycle policy for Amazon Kinesis Producer Library (KPL). AWS regularly provides new releases for KPL versions to support new features and enhancements, bug fixes, security patches, and dependency updates. We recommend that you stay up-to-date with KPL versions to keep up with the latest features, security updates, and underlying dependencies. We **don't** recommend continued use of an unsupported KPL version.

The lifecycle for major KPL versions consists of the following three phases:

- **General availability (GA)** – During this phase, the major version is fully supported. AWS provides regular minor and patch version releases that include support for new features or API updates for Kinesis Data Streams, as well as bug and security fixes.
- **Maintenance mode** – AWS limits patch version releases to address critical bug fixes and security issues only. The major version won't receive updates for new features or APIs of Kinesis Data Streams.
- **End-of-support** – The major version will no longer receive updates or releases. Previously published releases will continue to be available through public package managers and the code will remain on GitHub. Use of a version which has reached end-of-support is done at the user's discretion. We recommend that you upgrade to the latest major version.

Major version	Current phase	Release date	Maintenance mode date	End-of-support date
KPL 0.x	Maintenance mode	2015-06-02	2025-04-17	2026-01-30
KPL 1.x	General availability	2024-12-15	--	--

Develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java

You can develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java. If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in What is Amazon Kinesis Data Streams? and Use the AWS CLI to perform Amazon Kinesis Data Streams operations.

These examples discuss the Kinesis Data Streams API and use the AWS SDK for Java to add (put) data to a stream. However, for most use cases, you should prefer the Kinesis Data Streams KPL library. For more information, see Develop producers using the Amazon Kinesis Producer Library (KPL).

The Java example code in this chapter demonstrates how to perform basic Kinesis Data Streams API operations, and is divided up logically by operation type. These examples do not represent production-ready code, in that they do not check for all possible exceptions, or account for all possible security or performance considerations. Also, you can call the Kinesis Data Streams API using other programming languages. For more information about all available AWS SDKs, see Start Developing with Amazon Web Services.

Each task has prerequisites; for example, you cannot add data to a stream until you have created a stream, which requires you to create a client . For more information, see Create and manage Kinesis data streams.

Topics

- Add data to a stream
- Interact with data using the AWS Glue Schema Registry

Add data to a stream

Once a stream is created, you can add data to it in the form of records. A record is a data structure that contains the data to be processed in the form of a data blob. After you store the data in the record, Kinesis Data Streams does not inspect, interpret, or change the data in any way. Each record also has an associated sequence number and partition key.

There are two different operations in the Kinesis Data Streams API that add data to a stream, `PutRecords` and `PutRecord`. The `PutRecords` operation sends multiple records to your stream per HTTP request, and the singular `PutRecord` operation sends records to your stream one at a time (a separate HTTP request is required for each record). You should prefer using `PutRecords` for most applications because it will achieve higher throughput per data producer. For more information about each of these operations, see the separate subsections below.

Topics

- Add multiple records with PutRecords
- Add a single record with PutRecord

Always keep in mind that, as your source application is adding data to the stream using the Kinesis Data Streams API, there are most likely one or more consumer applications that are simultaneously processing data off the stream. For information about how consumers get data using the Kinesis Data Streams API, see Get data from a stream.

> **Important**
>
> Change the data retention period

Add multiple records with PutRecords

The `PutRecords` operation sends multiple records to Kinesis Data Streams in a single request. By using `PutRecords`, producers can achieve higher throughput when sending data to their Kinesis data stream. Each `PutRecords` request can support up to 500 records. Each record in the request can be as large as 1 MB, up to a limit of 5 MB for the entire request, including partition keys. As with the single `PutRecord` operation described below, `PutRecords` uses sequence numbers and partition keys. However, the `PutRecord` parameter `SequenceNumberForOrdering` is not

included in a PutRecords call. The PutRecords operation attempts to process all records in the natural order of the request.

Each data record has a unique sequence number. The sequence number is assigned by Kinesis Data Streams after you call client.putRecords to add the data records to the stream. Sequence numbers for the same partition key generally increase over time; the longer the time period between PutRecords requests, the larger the sequence numbers become.

> **Note**
>
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

A PutRecords request can include records with different partition keys. The scope of the request is a stream; each request may include any combination of partition keys and records up to the request limits. Requests made with many different partition keys to streams with many different shards are generally faster than requests with a small number of partition keys to a small number of shards. The number of partition keys should be much larger than the number of shards to reduce latency and maximize throughput.

PutRecords example

The following code creates 100 data records with sequential partition keys and puts them in a stream called DataStream.

```
        AmazonKinesisClientBuilder clientBuilder =
AmazonKinesisClientBuilder.standard();

        clientBuilder.setRegion(regionName);
        clientBuilder.setCredentials(credentialsProvider);
        clientBuilder.setClientConfiguration(config);

        AmazonKinesis kinesisClient = clientBuilder.build();

        PutRecordsRequest putRecordsRequest  = new PutRecordsRequest();
        putRecordsRequest.setStreamName(streamName);
        List <PutRecordsRequestEntry> putRecordsRequestEntryList  = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
```

```
            PutRecordsRequestEntry putRecordsRequestEntry  = new
PutRecordsRequestEntry();

putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(i).getBytes()));
            putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d",
i));
            putRecordsRequestEntryList.add(putRecordsRequestEntry);
        }

        putRecordsRequest.setRecords(putRecordsRequestEntryList);
        PutRecordsResult putRecordsResult  =
kinesisClient.putRecords(putRecordsRequest);
        System.out.println("Put Result" + putRecordsResult);
```

The `PutRecords` response includes an array of response `Records`. Each record in the response array directly correlates with a record in the request array using natural ordering, from the top to the bottom of the request and response. The response `Records` array always includes the same number of records as the request array.

Handle failures when using PutRecords

By default, failure of individual records within a request does not stop the processing of subsequent records in a `PutRecords` request. This means that a response `Records` array includes both successfully and unsuccessfully processed records. You must detect unsuccessfully processed records and include them in a subsequent call.

Successful records include `SequenceNumber` and `ShardID` values, and unsuccessful records include `ErrorCode` and `ErrorMessage` values. The `ErrorCode` parameter reflects the type of error and can be one of the following values: `ProvisionedThroughputExceededException` or `InternalFailure`. `ErrorMessage` provides more detailed information about the `ProvisionedThroughputExceededException` exception including the account ID, stream name, and shard ID of the record that was throttled. The example below has three records in a `PutRecords` request. The second record fails and is reflected in the response.

Example PutRecords Request Syntax

```
{
    "Records": [
        {
     "Data": "XzxkYXRhPl8w",
     "PartitionKey": "partitionKey1"
```

```
        },
        {
     "Data": "AbceddeRFfg12asd",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "KFpcd98*7nd1",
     "PartitionKey": "partitionKey3"
        }
    ],
    "StreamName": "myStream"
}
```

Example PutRecords Response Syntax

```
{
    "FailedRecordCount”: 1,
    "Records": [
        {
     "SequenceNumber": "21269319989900637946712965403778482371",
     "ShardId": "shardId-000000000001"

        },
        {
     “ErrorCode":”ProvisionedThroughputExceededException”,
     “ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream
 exampleStreamName under account 111111111111."

        },
        {
     "SequenceNumber": "21269319989999637946712965403778482985",
     "ShardId": "shardId-000000000002"
        }
    ]
}
```

Records that were unsuccessfully processed can be included in subsequent `PutRecords` requests. First, check the `FailedRecordCount` parameter in the `putRecordsResult` to confirm if there are failed records in the request. If so, each `putRecordsEntry` that has an `ErrorCode` that is not `null` should be added to a subsequent request. For an example of this type of handler, refer to the following code.

Example PutRecords failure handler

```
PutRecordsRequest putRecordsRequest = new PutRecordsRequest();
putRecordsRequest.setStreamName(myStreamName);
List<PutRecordsRequestEntry> putRecordsRequestEntryList = new ArrayList<>();
for (int j = 0; j < 100; j++) {
    PutRecordsRequestEntry putRecordsRequestEntry = new PutRecordsRequestEntry();
    putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(j).getBytes()));
    putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d", j));
    putRecordsRequestEntryList.add(putRecordsRequestEntry);
}

putRecordsRequest.setRecords(putRecordsRequestEntryList);
PutRecordsResult putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);

while (putRecordsResult.getFailedRecordCount() > 0) {
    final List<PutRecordsRequestEntry> failedRecordsList = new ArrayList<>();
    final List<PutRecordsResultEntry> putRecordsResultEntryList =
 putRecordsResult.getRecords();
    for (int i = 0; i < putRecordsResultEntryList.size(); i++) {
        final PutRecordsRequestEntry putRecordRequestEntry =
 putRecordsRequestEntryList.get(i);
        final PutRecordsResultEntry putRecordsResultEntry =
 putRecordsResultEntryList.get(i);
        if (putRecordsResultEntry.getErrorCode() != null) {
            failedRecordsList.add(putRecordRequestEntry);
        }
    }
    putRecordsRequestEntryList = failedRecordsList;
    putRecordsRequest.setRecords(putRecordsRequestEntryList);
    putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);
}
```

Add a single record with PutRecord

Each call to `PutRecord` operates on a single record. Prefer the `PutRecords` operation described in Add multiple records with PutRecords unless your application specifically needs to always send single records per request, or some other reason `PutRecords` can't be used.

Each data record has a unique sequence number. The sequence number is assigned by Kinesis Data Streams after you call `client.putRecord` to add the data record to the stream. Sequence

numbers for the same partition key generally increase over time; the longer the time period between `PutRecord` requests, the larger the sequence numbers become.

When puts occur in quick succession, the returned sequence numbers are not guaranteed to increase because the put operations appear essentially as simultaneous to Kinesis Data Streams. To guarantee strictly increasing sequence numbers for the same partition key, use the `SequenceNumberForOrdering` parameter, as shown in the PutRecord example code sample.

Whether or not you use `SequenceNumberForOrdering`, records that Kinesis Data Streams receives through a `GetRecords` call are strictly ordered by sequence number.

> **Note**
>
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

A partition key is used to group data within the stream. A data record is assigned to a shard within the stream based on its partition key. Specifically, Kinesis Data Streams uses the partition key as input to a hash function that maps the partition key (and associated data) to a specific shard.

As a result of this hashing mechanism, all data records with the same partition key map to the same shard within the stream. However, if the number of partition keys exceeds the number of shards, some shards necessarily contain records with different partition keys. From a design standpoint, to ensure that all your shards are well utilized, the number of shards (specified by the `setShardCount` method of `CreateStreamRequest`) should be substantially less than the number of unique partition keys, and the amount of data flowing to a single partition key should be substantially less than the capacity of the shard.

PutRecord example

The following code creates ten data records, distributed across two partition keys, and puts them in a stream called `myStreamName`.

```
for (int j = 0; j < 10; j++)
{
  PutRecordRequest putRecordRequest = new PutRecordRequest();
  putRecordRequest.setStreamName( myStreamName );
```

```
  putRecordRequest.setData(ByteBuffer.wrap( String.format( "testData-%d",
 j ).getBytes() ));
  putRecordRequest.setPartitionKey( String.format( "partitionKey-%d", j/5 ));
  putRecordRequest.setSequenceNumberForOrdering( sequenceNumberOfPreviousRecord );
  PutRecordResult putRecordResult = client.putRecord( putRecordRequest );
  sequenceNumberOfPreviousRecord = putRecordResult.getSequenceNumber();
}
```

The preceding code sample uses `setSequenceNumberForOrdering` to guarantee strictly increasing ordering within each partition key. To use this parameter effectively, set the `SequenceNumberForOrdering` of the current record (record *n*) to the sequence number of the preceding record (record *n-1*). To get the sequence number of a record that has been added to the stream, call `getSequenceNumber` on the result of `putRecord`.

The `SequenceNumberForOrdering` parameter ensures strictly increasing sequence numbers for the same partition key. `SequenceNumberForOrdering` does not provide ordering of records across multiple partition keys.

Interact with data using the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue Schema Registry. The AWS Glue Schema Registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry lets you improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the `PutRecords` and `PutRecord` Kinesis Data Streams APIs available in the AWS Java SDK.

For detailed instructions on how to set up integration of Kinesis Data Streams with schema registry using the PutRecords and PutRecord Kinesis Data Streams APIs, see the "Interacting with Data Using the Kinesis Data Streams APIs" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Write to Amazon Kinesis Data Streams using Kinesis Agent

Kinesis Agent is a stand-alone Java software application that offers an easy way to collect and send data to Kinesis Data Streams. The agent continuously monitors a set of files and sends new data to your stream. The agent handles file rotation, checkpointing, and retry upon failures. It delivers all

of your data in a reliable, timely, and simple manner. It also emits Amazon CloudWatch metrics to help you better monitor and troubleshoot the streaming process.

By default, records are parsed from each file based on the newline (`'\n'`) character. However, the agent can also be configured to parse multi-line records (see Specify the agent configuration settings).

You can install the agent on Linux-based server environments such as web servers, log servers, and database servers. After installing the agent, configure it by specifying the files to monitor and the stream for the data. After the agent is configured, it durably collects data from the files and reliably sends it to the stream.

Topics

- Complete the prerequisites for Kinesis Agent
- Download and install the agent
- Configure and start the agent
- Specify the agent configuration settings
- Monitor multiple file directories and write to multiple streams
- Use the agent to pre-process data
- Use agent CLI commands
- FAQ

Complete the prerequisites for Kinesis Agent

- Your operating system must be either Amazon Linux AMI with version 2015.09 or later, or Red Hat Enterprise Linux version 7 or later.
- If you are using Amazon EC2 to run your agent, launch your EC2 instance.
- Manage your AWS credentials using one of the following methods:
  - Specify an IAM role when you launch your EC2 instance.
  - Specify AWS credentials when you configure the agent (see awsAccessKeyId and awsSecretAccessKey).
  - Edit `/etc/sysconfig/aws-kinesis-agent` to specify your region and AWS access keys.
  - If your EC2 instance is in a different AWS account, create an IAM role to provide access to the Kinesis Data Streams service, and specify that role when you configure the agent (see

assumeRoleARN and assumeRoleExternalId). Use one of the previous methods to specify the AWS credentials of a user in the other account who has permission to assume this role.

- The IAM role or AWS credentials that you specify must have permission to perform the Kinesis Data Streams PutRecords operation for the agent to send data to your stream. If you enable CloudWatch monitoring for the agent, permission to perform the CloudWatch PutMetricData operation is also needed. For more information, see Controlling access to Amazon Kinesis Data Streams resources using IAM, Monitor Kinesis Data Streams Agent health with Amazon CloudWatch, and CloudWatch Access Control.

Download and install the agent

First, connect to your instance. For more information, see Connect to Your Instance in the *Amazon EC2 User Guide*. If you have trouble connecting, see Troubleshooting Connecting to Your Instance in the *Amazon EC2 User Guide*.

To set up the agent using the Amazon Linux AMI

Use the following command to download and install the agent:

```
sudo yum install -y aws-kinesis-agent
```

To set up the agent using Red Hat Enterprise Linux

Use the following command to download and install the agent:

```
sudo yum install -y https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-latest.amzn2.noarch.rpm
```

To set up the agent using GitHub

1. Download the agent from awlabs/amazon-kinesis-agent.
2. Install the agent by navigating to the download directory and running the following command:

   ```
   sudo ./setup --install
   ```

To set up the agent in a Docker container

Kinesis Agent can be run in a container as well via the amazonlinux container base. Use the following Dockerfile and then run `docker build`.

```
FROM amazonlinux

RUN yum install -y aws-kinesis-agent which findutils
COPY agent.json /etc/aws-kinesis/agent.json

CMD ["start-aws-kinesis-agent"]
```

Configure and start the agent

To configure and start the agent

1. Open and edit the configuration file (as superuser if using default file access permissions): `/etc/aws-kinesis/agent.json`

   In this configuration file, specify the files (`"filePattern"`) from which the agent collects data, and the name of the stream (`"kinesisStream"`) to which the agent sends data. Note that the file name is a pattern, and the agent recognizes file rotations. You can rotate files or create new files no more than once per second. The agent uses the file creation timestamp to determine which files to track and tail into your stream; creating new files or rotating files more frequently than once per second does not allow the agent to differentiate properly between them.

   ```
   {
      "flows": [
           {
               "filePattern": "/tmp/app.log*",
               "kinesisStream": "yourkinesisstream"
           }
      ]
   }
   ```

2. Start the agent manually:

   ```
   sudo service aws-kinesis-agent start
   ```

3. (Optional) Configure the agent to start on system startup:

```
sudo chkconfig aws-kinesis-agent on
```

The agent is now running as a system service in the background. It continuously monitors the specified files and sends data to the specified stream. Agent activity is logged in `/var/log/aws-kinesis-agent/aws-kinesis-agent.log`.

Specify the agent configuration settings

The agent supports the two mandatory configuration settings, `filePattern` and `kinesisStream`, plus optional configuration settings for additional features. You can specify both mandatory and optional configuration in `/etc/aws-kinesis/agent.json`.

Whenever you change the configuration file, you must stop and start the agent, using the following commands:

```
sudo service aws-kinesis-agent stop
sudo service aws-kinesis-agent start
```

Alternatively, you could use the following command:

```
sudo service aws-kinesis-agent restart
```

The following are the general configuration settings.

Configuration Setting	Description
`assumeRoleARN`	The ARN of the role to be assumed by the user. For more information, see Delegate Access Across AWS Accounts Using IAM Roles in the *IAM User Guide*.
`assumeRoleExternalId`	An optional identifier that determines who can assume the role. For more information, see How to Use an External ID in the *IAM User Guide*.
`awsAccessKeyId`	AWS access key ID that overrides the default credentials. This setting takes precedence over all other credential providers.

Configuration Setting	Description
awsSecretAccessKey	AWS secret key that overrides the default credentials. This setting takes precedence over all other credential providers.
cloudwatch.emitMetrics	Enables the agent to emit metrics to CloudWatch if set (true). Default: true
cloudwatch.endpoint	The regional endpoint for CloudWatch. Default: monitoring.us-east-1.amazonaws.com
kinesis.endpoint	The regional endpoint for Kinesis Data Streams. Default: kinesis.us-east-1.amazonaws.com

The following are the flow configuration settings.

Configuration Setting	Description
dataProcessingOptions	The list of processing options applied to each parsed record before it is sent to the stream. The processing options are performed in the specified order. For more information, see Use the agent to pre-process data.
kinesisStream	[Required] The name of the stream.
filePattern	[Required] The directory and file pattern that must be matched to be picked up by the agent. For all files matching this pattern, read permission must be granted to aws-kinesis-agent-user . For the directory containing the files, read and execute permissions must be granted to aws-kinesis-agent-user .
initialPosition	The initial position from which the file started to be parsed. Valid values are START_OF_FILE and END_OF_FILE .

Configuration Setting	Description
	Default: `END_OF_FILE`
`maxBufferAgeMillis`	The maximum time, in milliseconds, for which the agent buffers data before sending it to the stream. Value range: 1,000 to 900,000 (1 second to 15 minutes) Default: 60,000 (1 minute)
`maxBufferSizeBytes`	The maximum size, in bytes, for which the agent buffers data before sending it to the stream. Value range: 1 to 4,194,304 (4 MB) Default: 4,194,304 (4 MB)
`maxBufferSizeRecords`	The maximum number of records for which the agent buffers data before sending it to the stream. Value range: 1 to 500 Default: 500
`minTimeBetweenFilePollsMillis`	The time interval, in milliseconds, at which the agent polls and parses the monitored files for new data. Value range: 1 or more Default: 100
`multiLineStartPattern`	The pattern for identifying the start of a record. A record is made of a line that matches the pattern and any following lines that don't match the pattern. The valid values are regular expressions. By default, each new line in the log files is parsed as one record.

Configuration Setting	Description
partitionKeyOption	The method for generating the partition key. Valid values are RANDOM (randomonly generated integer) and DETERMINISTIC (a hash value computed from the data). Default: RANDOM
skipHeaderLines	The number of lines for the agent to skip parsing at the beginning of monitored files. Value range: 0 or more Default: 0 (zero)
truncatedRecordTerminator	The string that the agent uses to truncate a parsed record when the record size exceeds the Kinesis Data Streams record size limit. (1,000 KB) Default: '\n' (newline)

Monitor multiple file directories and write to multiple streams

By specifying multiple flow configuration settings, you can configure the agent to monitor multiple file directories and send data to multiple streams. In the following configuration example, the agent monitors two file directories and sends data to an Kinesis stream and a Firehose delivery stream respectively. Note that you can specify different endpoints for Kinesis Data Streams and Firehose so that your Kinesis stream and Firehose delivery stream don't need to be in the same region.

```
{
    "cloudwatch.emitMetrics": true,
    "kinesis.endpoint": "https://your/kinesis/endpoint",
    "firehose.endpoint": "https://your/firehose/endpoint",
    "flows": [
        {
            "filePattern": "/tmp/app1.log*",
            "kinesisStream": "yourkinesisstream"
```

```
        },
        {
            "filePattern": "/tmp/app2.log*",
            "deliveryStream": "yourfirehosedeliverystream"
        }
    ]
}
```

For more detailed information about using the agent with Firehose, see Writing to Amazon Data Firehose with Kinesis Agent.

Use the agent to pre-process data

The agent can pre-process the records parsed from monitored files before sending them to your stream. You can enable this feature by adding the `dataProcessingOptions` configuration setting to your file flow. One or more processing options can be added and they will be performed in the specified order.

The agent supports the following processing options listed. Because the agent is open-source, you can further develop and extend its processing options. You can download the agent from Kinesis Agent.

Processing Options

`SINGLELINE`

Converts a multi-line record to a single line record by removing newline characters, leading spaces, and trailing spaces.

```
{
    "optionName": "SINGLELINE"
}
```

`CSVTOJSON`

Converts a record from delimiter separated format to JSON format.

```
{
    "optionName": "CSVTOJSON",
    "customFieldNames": [ "field1", "field2", ... ],
```

```
    "delimiter": "yourdelimiter"
}
```

`customFieldNames`

[Required] The field names used as keys in each JSON key value pair. For example, if you specify `["f1", "f2"]`, the record "v1, v2" will be converted to `{"f1":"v1","f2":"v2"}`.

`delimiter`

The string used as the delimiter in the record. The default is a comma (,).

`LOGTOJSON`

Converts a record from a log format to JSON format. The supported log formats are **Apache Common Log**, **Apache Combined Log**, **Apache Error Log**, and **RFC3164 Syslog**.

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "logformat",
    "matchPattern": "yourregexpattern",
    "customFieldNames": [ "field1", "field2", … ]
}
```

`logFormat`

[Required] The log entry format. The following are possible values:

- `COMMONAPACHELOG` — The Apache Common Log format. Each log entry has the following pattern by default: `"%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes}"`.
- `COMBINEDAPACHELOG` — The Apache Combined Log format. Each log entry has the following pattern by default: `"%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes} %{referrer} %{agent}"`.
- `APACHEERRORLOG` — The Apache Error Log format. Each log entry has the following pattern by default: `"[%{timestamp}] [%{module}:%{severity}] [pid %{processid}:tid %{threadid}] [client: %{client}] %{message}"`.
- `SYSLOG` — The RFC3164 Syslog format. Each log entry has the following pattern by default: `"%{timestamp} %{hostname} %{program}[%{processid}]: %{message}"`.

`matchPattern`

The regular expression pattern used to extract values from log entries. This setting is used if your log entry is not in one of the predefined log formats. If this setting is used, you must also specify `customFieldNames`.

`customFieldNames`

The custom field names used as keys in each JSON key value pair. You can use this setting to define field names for values extracted from `matchPattern`, or override the default field names of predefined log formats.

Example: LOGTOJSON Configuration

Here is one example of a `LOGTOJSON` configuration for an Apache Common Log entry converted to JSON format:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG"
}
```

Before conversion:

```
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision
 HTTP/1.1" 200 6291
```

After conversion:

```
{"host":"64.242.88.10","ident":null,"authuser":null,"datetime":"07/
Mar/2004:16:10:02 -0800","request":"GET /mailman/listinfo/hsdivision
 HTTP/1.1","response":"200","bytes":"6291"}
```

Example: LOGTOJSON Configuration With Custom Fields

Here is another example `LOGTOJSON` configuration:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
    "customFieldNames": ["f1", "f2", "f3", "f4", "f5", "f6", "f7"]
```

```
}
```

With this configuration setting, the same Apache Common Log entry from the previous example is converted to JSON format as follows:

```
{"f1":"64.242.88.10","f2":null,"f3":null,"f4":"07/Mar/2004:16:10:02 -0800","f5":"GET /
mailman/listinfo/hsdivision HTTP/1.1","f6":"200","f7":"6291"}
```

Example: Convert Apache Common Log Entry

The following flow configuration converts an Apache Common Log entry to a single line record in JSON format:

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "dataProcessingOptions": [
                {
                    "optionName": "LOGTOJSON",
                    "logFormat": "COMMONAPACHELOG"
                }
            ]
        }
    ]
}
```

Example: Convert Multi-Line Records

The following flow configuration parses multi-line records whose first line starts with "[SEQUENCE=". Each record is first converted to a single line record. Then, values are extracted from the record based on a tab delimiter. Extracted values are mapped to specified `customFieldNames` values to form a single-line record in JSON format.

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "multiLineStartPattern": "\\[SEQUENCE=",
```

```
            "dataProcessingOptions": [
                {
                    "optionName": "SINGLELINE"
                },
                {
                    "optionName": "CSVTOJSON",
                    "customFieldNames": [ "field1", "field2", "field3" ],
                    "delimiter": "\\t"
                }
            ]
        }
    ]
}
```

Example: LOGTOJSON Configuration with Match Pattern

Here is one example of a LOGTOJSON configuration for an Apache Common Log entry converted to JSON format, with the last field (bytes) omitted:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
    "matchPattern": "^([\\d.]+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.
+?)\" (\\d{3})",
    "customFieldNames": ["host", "ident", "authuser", "datetime", "request",
 "response"]
}
```

Before conversion:

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
 200
```

After conversion:

```
{"host":"123.45.67.89","ident":null,"authuser":null,"datetime":"27/Oct/2000:09:27:09
 -0400","request":"GET /java/javaResources.html HTTP/1.0","response":"200"}
```

Use agent CLI commands

Automatically start the agent on system startup:

```
sudo chkconfig aws-kinesis-agent on
```

Check the status of the agent:

```
sudo service aws-kinesis-agent status
```

Stop the agent:

```
sudo service aws-kinesis-agent stop
```

Read the agent's log file from this location:

```
/var/log/aws-kinesis-agent/aws-kinesis-agent.log
```

Uninstall the agent:

```
sudo yum remove aws-kinesis-agent
```

FAQ

Is there a Kinesis Agent for Windows?

Kinesis Agent for Windows is different software than Kinesis Agent for Linux platforms.

Why is Kinesis Agent slowing down and/or `RecordSendErrors` increasing?

This is usually due to throttling from Kinesis. Check the `WriteProvisionedThroughputExceeded` metric for Kinesis Data Streams or the `ThrottledRecords` metric for Firehose Delivery Streams. Any increase from 0 in these metrics indicates that the stream limits need to be increased. For more information, see Kinesis Data Stream limits and Amazon Firehose Delivery Streams.

Once you rule out throttling, see if the Kinesis Agent is configured to tail a large amount of small files. There is a delay when Kinesis Agent tails a new file, so Kinesis Agent should be tailing a small amount of larger files. Try consolidating your log files into larger files.

Why am I getting `java.lang.OutOfMemoryError`exceptions?

Kinesis Agent does not have enough memory to handle its current workload. Try increasing `JAVA_START_HEAP` and `JAVA_MAX_HEAP` in `/usr/bin/start-aws-kinesis-agent` and restarting the agent.

Why am I getting `IllegalStateException : connection pool shut down` exceptions?

Kinesis Agent does not have enough connections to handle its current workload. Try increasing `maxConnections` and `maxSendingThreads` in your general agent configuration settings at `/etc/aws-kinesis/agent.json`. The default value for these fields is 12 times the runtime processors available. See AgentConfiguration.java for more about advanced agent configurations settings.

How can I debug another issue with Kinesis Agent?

`DEBUG` level logs can be enabled in `/etc/aws-kinesis/log4j.xml` .

How should I configure Kinesis Agent?

The smaller the `maxBufferSizeBytes`, the more frequently Kinesis Agent will send data. This can be good as it decreases delivery time of records, but it also increases the requests per second to Kinesis.

Why is Kinesis Agent sending duplicate records?

This occurs due to a misconfiguration in file tailing. Make sure that each `fileFlow's filePattern` is only matching one file. This can also occur if the `logrotate` mode being used is in `copytruncate` mode. Try changing the mode to the default or create mode to avoid duplication. For more information on handling duplicate records, see Handling Duplicate Records.

Write to Kinesis Data Streams using other AWS services

The following AWS services can integrate directly with Amazon Kinesis Data Streams to write data to Kinesis data streams. Review the information for each service that you are interested in and refer to the provided references.

Topics

- Write to Kinesis Data Streams using AWS Amplify

- Write to Kinesis Data Streams using Amazon Aurora
- Write to Kinesis Data Streams using Amazon CloudFront
- Write to Kinesis Data Streams using Amazon CloudWatch Logs
- Write to Kinesis Data Streams using Connect Customer
- Write to Kinesis Data Streams using AWS Database Migration Service
- Write to Kinesis Data Streams using Amazon DynamoDB
- Write to Kinesis Data Streams using Amazon EventBridge
- Write to Kinesis Data Streams using AWS IoT Core
- Write to Kinesis Data Streams using Amazon Relational Database Service
- Write to Kinesis Data Streams usingAmazon Pinpoint
- Write to Kinesis Data Streams using Amazon Quantum Ledger Database (Amazon QLDB)

Write to Kinesis Data Streams using AWS Amplify

You can use Amazon Kinesis Data Streams to stream data from your mobile applications built with AWS Amplify for real-time processing. You can then build real-time dashboards, capture exceptions and generate alerts, drive recommendations, and make other real-time business or operational decisions. You can also send data to other services such as Amazon Simple Storage Service, Amazon DynamoDB, and Amazon Redshift.

For more information, see Using Amazon Kinesis in the *AWS Amplify Developer Center*.

Write to Kinesis Data Streams using Amazon Aurora

You can use Amazon Kinesis Data Streams to monitor activities on your Amazon Aurora DB clusters. Using Database Activity Streams, your Aurora DB cluster pushes activities to an Amazon Kinesis Data Stream in real-time. You can then build applications for compliance management that consume these activities, audit them and generate alerts. You can also use Amazon Amazon Firehose to store the data.

For more information, see Database Activity Streams in the *Amazon Aurora Developer Guide*.

Write to Kinesis Data Streams using Amazon CloudFront

You can use Amazon Kinesis Data Streams with CloudFront real-time logs and get information about requests made to a distribution in real time. You can then build your own Kinesis data stream

consumer, or use Amazon Data Firehose to send the log data to Amazon S3, Amazon Redshift, Amazon OpenSearch Service, or a third-party log processing service.

For more information, see Real-time logs in the *Amazon CloudFront Developer Guide*.

Write to Kinesis Data Streams using Amazon CloudWatch Logs

You can use CloudWatch subscriptions to get access to a real-time feed of log events from Amazon CloudWatch Logs and have it delivered to a Kinesis data stream for processing, analysis, and loading to to other systems.

For more information, see Real-time processing of log data with subscriptions in the *Amazon CloudWatch Logs User Guide*.

Write to Kinesis Data Streams using Connect Customer

You can use Kinesis Data Streams to export contact records and agent events in real-time from your Connect Customer instance. You can also enable data streaming from Connect Customer Customer Profiles to automatically receive updates to a Kinesis data stream about creation of new profiles or changes to existing ones.

You can then build consumer applications to process and analyze the data in real time. For example, using contact records and customer profile data, you can keep your source systems data, such as CRMs and marketing automation tools, up-to-date with the latest information. Using the agents event data, you can create dashboards that display agent information and events, and trigger custom notifications of specific agent activity.

For more information, see data streaming for your instance, set up real-time export, and agent event streams in the *Amazon Connect Administrator Guide*.

Write to Kinesis Data Streams using AWS Database Migration Service

You can use AWS Database Migration Service to migrate data to a Kinesis data stream. You can than build consumer applications that process the data records in real time. You can also easily send data downstream to other services such as Amazon Simple Storage Service, Amazon DynamoDB, and Amazon Redshift

For more information, see Using Kinesis Data Streams in the *AWS Database Migration Service User Guide*.

Write to Kinesis Data Streams using Amazon DynamoDB

You can use Amazon Kinesis Data Streams to capture changes to Amazon DynamoDB. Kinesis Data Streams captures item-level modifications in any DynamoDB table and replicates them to a Kinesis data stream. Your consumer applications can access this stream to view item-level changes in real time and deliver those changes downstream or take action based on the content.

For more information, see how Kinesis Data Streams work with DynamoDB in the *Amazon DynamoDB Developer Guide*.

Write to Kinesis Data Streams using Amazon EventBridge

Using Kinesis Data Streams, you can send AWS API call events in EventBridge to a stream, build consumer applications, and process large amounts of data. You can also use Kinesis Data Streams as a target in EventBridge Pipes and deliver records a stream from one of the available sources after optional filtering and enrichment.

For more information, see Send events to an Amazon Kinesis stream and EventBridge Pipes in the *Amazon EventBridge User Guide*.

Write to Kinesis Data Streams using AWS IoT Core

You can write data in real time from MQTT messages in AWS IoT Core by using AWS IoT Rule actions. You can then build applications that process the data, analyze its contents and generate alerts, and deliver it into analytics applications or other AWS services,

For more information, see Kinesis Data Streams in the *AWS IoT Core Developer Guide*.

Write to Kinesis Data Streams using Amazon Relational Database Service

You can use Amazon Kinesis Data Streams to monitor activities on your Amazon RDS instances. Using Database Activity Streams, Amazon RDS pushes activities to a Kinesis data stream in real-time. You can then build applications for compliance management that consume these activities, audit them and generate alerts. You can also use Amazon Data Firehose to store the data.

For more information, see Database Activity Streams in the *Amazon RDS Developer Guide*.

Write to Kinesis Data Streams usingAmazon Pinpoint

You can set up Amazon Pinpoint to send event data to Amazon Kinesis Data Streams. Amazon Pinpoint can send event data for campaigns, journeys, and transactional email and SMS messages. You can then ingest the data into analytics applications or build your own consumer applications that take actions based on the contents of the events.

For more information, see Streaming Events in the *Amazon Pinpoint Developer Guide*.

Write to Kinesis Data Streams using Amazon Quantum Ledger Database (Amazon QLDB)

You can create a stream in Amazon QLDB that captures every document revision that is committed to your journal and delivers this data to Amazon Kinesis Data Streams in real time. A QLDB stream is a continuous flow of data from your ledger's journal to a Kinesis data stream resource. Then, you can use the Kinesis streaming platform or the Kinesis Client Library to consume your stream, process the data records, and analyze the data contents. A QLDB stream writes your data to Kinesis Data Streams in three types of records: `control`, `block summary`, and `revision details`.

For more information, see Streams in the *Amazon QLDB developer Guide*.

Write to Kinesis Data Streams using third-party integrations

You can write data to Kinesis Data Streams using one of the following third-party options that integrate with Kinesis Data Streams. Select the option that you want to learn more about and find resources and links to relevant documentation.

Topics

- Apache Flink
- Fluentd
- Debezium
- Oracle GoldenGate
- Kafka Connect
- Adobe Experience
- Striim

Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. For more information on writing to Kinesis Data Streams from Apache Flink, see Amazon Kinesis Data Streams Connector.

Fluentd

Fluentd is an open source data collector for unified logging layer. For more information on writing to Kinesis Data Streams from Fluentd. For more information see Stream processing with Kinesis.

Debezium

Debezium is an open source distributed platform for change data capture. For more information on writing to Kinesis Data Streams from Debezium, see Streaming MySQL Data Changes to Amazon Kinesis.

Oracle GoldenGate

Oracle GoldenGate is a software product that allows you to replicate, filter, and transform data from one database to another database. For more information on writing to Kinesis Data Streams from Oracle GoldenGate, see Data replication to Kinesis Data Stream using Oracle GoldenGate.

Kafka Connect

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. For more information on writing data from Apache Kafka to Kinesis Data Streams, see the Kinesis kafka connector.

Adobe Experience

Adobe Experience Platform enables organizations to centralize and standardize customer data from any system. It then applies data science and machine learning to dramatically improve the design and delivery of rich, personalized experiences. For more information on writing data from the Adobe Experience Platform to Kinesis Data Streams. see how to create an Amazon Kinesis connection.

Striim

Striim is a complete, end-to-end, in-memory platform for collecting, filtering, transforming, enriching, aggregating, analyzing, and delivering data in real time. For more information on how to write data to Kinesis Data Streams from Striim, see the Kinesis Writer.

Troubleshoot Amazon Kinesis Data Streams producers

The following topics offer solutions to common issues with Amazon Kinesis Data Streams producers:

- My producer application is writing at a slower rate than expected
- I receive an unauthorized KMS master key permission error
- Troubleshoot other common issues for producers

My producer application is writing at a slower rate than expected

The most common reasons for write throughput being slower than expected are:

- Service limits exceeded
- I want to optimize my producer
- Misuse of flushSync() operations

Service limits exceeded

To find out if service limits are being exceeded, check to see if your producer is throwing throughput exceptions from the service, and validate what API operations are being throttled. Keep in mind that there are different limits based on the call, see Quotas and limits. For example, in addition to the shard-level limits for writes and reads that are most commonly known, there are the following stream-level limits:

- CreateStream
- DeleteStream
- ListStreams
- GetShardIterator
- MergeShards

- DescribeStream
- DescribeStreamSummary

The operations `CreateStream`, `DeleteStream`, `ListStreams`, `GetShardIterator`, and `MergeShards` are limited to 5 calls per second. The `DescribeStream` operation is limited to 10 calls per second. The `DescribeStreamSummary` operation is limited to 20 calls per second.

If these calls aren't the issue, make sure you've selected a partition key that allows you to distribute *put* operations evenly across all shards, and that you don't have a particular partition key that's bumping into the service limits when the rest are not. This requires that you measure peak throughput and take into account the number of shards in your stream. For more information about managing streams, see Create and manage Kinesis data streams.

Tip

Remember to round up to the nearest kilobyte for throughput throttling calculations when using the single-record operation PutRecord, while the multi-record operation PutRecords rounds on the cumulative sum of the records in each call. For example, a `PutRecords` request with 600 records that are 1.1 KB in size will not get throttled.

I want to optimize my producer

Before you begin optimizing your producer, complete the following key tasks. First, identify your desired peak throughput in terms of record size and records per second. Next, rule out stream capacity as the limiting factor (Service limits exceeded). If you've ruled out stream capacity, use the following troubleshooting tips and optimization guidelines for the two common types of producers.

Large Producer

A large producer is usually running from an on-premises server or Amazon EC2 instance. Customers who need higher throughput from a large producer typically care about per-record latency. Strategies for dealing with latency include the following: If the customer can micro-batch/buffer records, use the Amazon Kinesis Producer Library (which has advanced aggregation logic), the multi-record operation PutRecords, or aggregate records into a larger file before using the single-record operation PutRecord. If you are unable to batch/buffer, use multiple threads to write to the

Kinesis Data Streams service at the same time. The AWS SDK for Java and other SDKs include async clients that can do this with very little code.

Small Producer

A small producer is usually a mobile app, IoT device, or web client. If it's a mobile app, we recommend using the `PutRecords` operation or the Kinesis Recorder in the AWS Mobile SDKs. For more information, see AWS Mobile SDK for Android Getting Started Guide and AWS Mobile SDK for iOS Getting Started Guide. Mobile apps must handle intermittent connections inherently and need some sort of batch put, such as `PutRecords`. If you are unable to batch for some reason, see the Large Producer information above. If your producer is a browser, the amount of data being generated is typically very small. However, you are putting the *put* operations on the critical path of the application, which we don't recommend.

Misuse of `flushSync()` operations

Using `flushSync()` incorrectly can significantly impact write performance. The `flushSync()` operation is designed for shutdown scenarios to make sure that all buffered records are sent before the KPL application terminates. If you implemented this operation after every write operation, it can add substantial extra latency, around 500ms per write. Make sure that you have implemented `flushSync()` only for the application shutdown to avoid unnecessary extra delay in write performance.

I receive an unauthorized KMS master key permission error

This error occurs when a producer application writes to an encrypted stream without permissions on the KMS master key. To assign permissions to an application to access a KMS key, see Using Key Policies in AWS KMS and Using IAM Policies with AWS KMS.

Troubleshoot other common issues for producers

- Why is my Kinesis data stream returning a 500 Internal Server Error?
- How do I troubleshoot timeout errors when writing from Flink to Kinesis Data Streams?
- How do I troubleshoot throttling errors in Kinesis Data Streams?
- Why is my Kinesis data stream throttling?
- How can I put data records into a Kinesis data stream using the KPL?

Optimize Kinesis Data Streams producers

You can further optimize your Amazon Kinesis Data Streams producers depending on specific behavior you see. Review the following topics to identify solutions.

Topics

- Customize KPL retries and rate limit behavior
- Apply best practices to KPL aggregation

Customize KPL retries and rate limit behavior

When you add Amazon Kinesis Producer Library (KPL) user records using the KPL `addUserRecord()` operation, a record is given a time stamp and added to a buffer with a deadline set by the `RecordMaxBufferedTime` configuration parameter. This time stamp/deadline combination sets the buffer priority. Records are flushed from the buffer based on the following criteria:

- Buffer priority
- Aggregation configuration
- Collection configuration

The aggregation and collection configuration parameters affecting buffer behavior are as follows:

- `AggregationMaxCount`
- `AggregationMaxSize`
- `CollectionMaxCount`
- `CollectionMaxSize`

Records flushed are then sent to your Kinesis data stream as Amazon Kinesis Data Streams records using a call to the Kinesis Data Streams API operation `PutRecords`. The `PutRecords` operation sends requests to your stream that occasionally exhibit full or partial failures. Records that fail are automatically added back to the KPL buffer. The new deadline is set based on the minimum of these two values:

- Half the current `RecordMaxBufferedTime` configuration

- The record's time-to-live value

This strategy allows retried KPL user records to be included in subsequent Kinesis Data Streams API calls, to improve throughput and reduce complexity while enforcing the Kinesis Data Streams record's time-to-live value. There is no backoff algorithm, making this a relatively aggressive retry strategy. Spamming due to excessive retries is prevented by rate limiting, discussed in the next section.

Rate limiting

The KPL includes a rate limiting feature, which limits per-shard throughput sent from a single producer. Rate limiting is implemented using a token bucket algorithm with separate buckets for both Kinesis Data Streams records and bytes. Each successful write to a Kinesis data stream adds a token (or multiple tokens) to each bucket, up to a certain threshold. This threshold is configurable but by default is set 50 percent higher than the actual shard limit, to allow shard saturation from a single producer.

You can lower this limit to reduce spamming due to excessive retries. However, the best practice is for each producer to retry for maximum throughput aggressively and to handle any resulting throttling determined as excessive by expanding the capacity of the stream and implementing an appropriate partition key strategy.

Apply best practices to KPL aggregation

While the sequence number scheme of the resulting Amazon Kinesis Data Streams records remains the same, aggregation causes the indexing of Amazon Kinesis Producer Library (KPL) user records contained within an aggregated Kinesis Data Streams record to start at 0 (zero); however, as long as you do not rely on sequence numbers to uniquely identify your KPL user records, your code can ignore this, as the aggregation (of your KPL user records into a Kinesis Data Streams record) and subsequent de-aggregation (of a Kinesis Data Streams record into your KPL user records) automatically takes care of this for you. This applies whether your consumer is using the KCL or the AWS SDK. To use this aggregation functionality, you'll need to pull the Java part of the KPL into your build if your consumer is written using the API provided in the AWS SDK.

If you intend to use sequence numbers as unique identifiers for your KPL user records, we recommend that you use the contract-abiding `public int hashCode()` and `public boolean equals(Object obj)` operations provided in `Record` and `UserRecord` to enable the comparison of your KPL user records. Additionally, if you want to examine the subsequence number

of your KPL user record, you can cast it to a `UserRecord` instance and retrieve its subsequence number.

For more information, see Implement consumer de-aggregation.

Read data from Amazon Kinesis Data Streams

A *consumer* is an application that processes all data from a Kinesis data stream. When a consumer uses *enhanced fan-out*, it gets its own 2 MB/sec allotment of read throughput, allowing multiple consumers to read data from the same stream in parallel, without contending for read throughput with other consumers. To use the enhanced fan-out capability of shards, see Develop enhanced fan-out consumers with dedicated throughput.

You can build consumers for Kinesis Data Streams using Kinesis Client Library (KCL) or AWS SDK for Java. You can also develop consumers using other AWS services such as AWS Lambda, Amazon Managed Service for Apache Flink, and Amazon Data Firehose. Kinesis Data Streams supports integrations with other AWS services such as Amazon EMR, Amazon EventBridge, AWS Glue, and Amazon Redshift. It also supports third party integrations including Apache Flink, Adobe Experience Platform, Apache Druid, Apache Spark, Databricks, Confluent Platform, Kinesumer, and Talend.

Topics

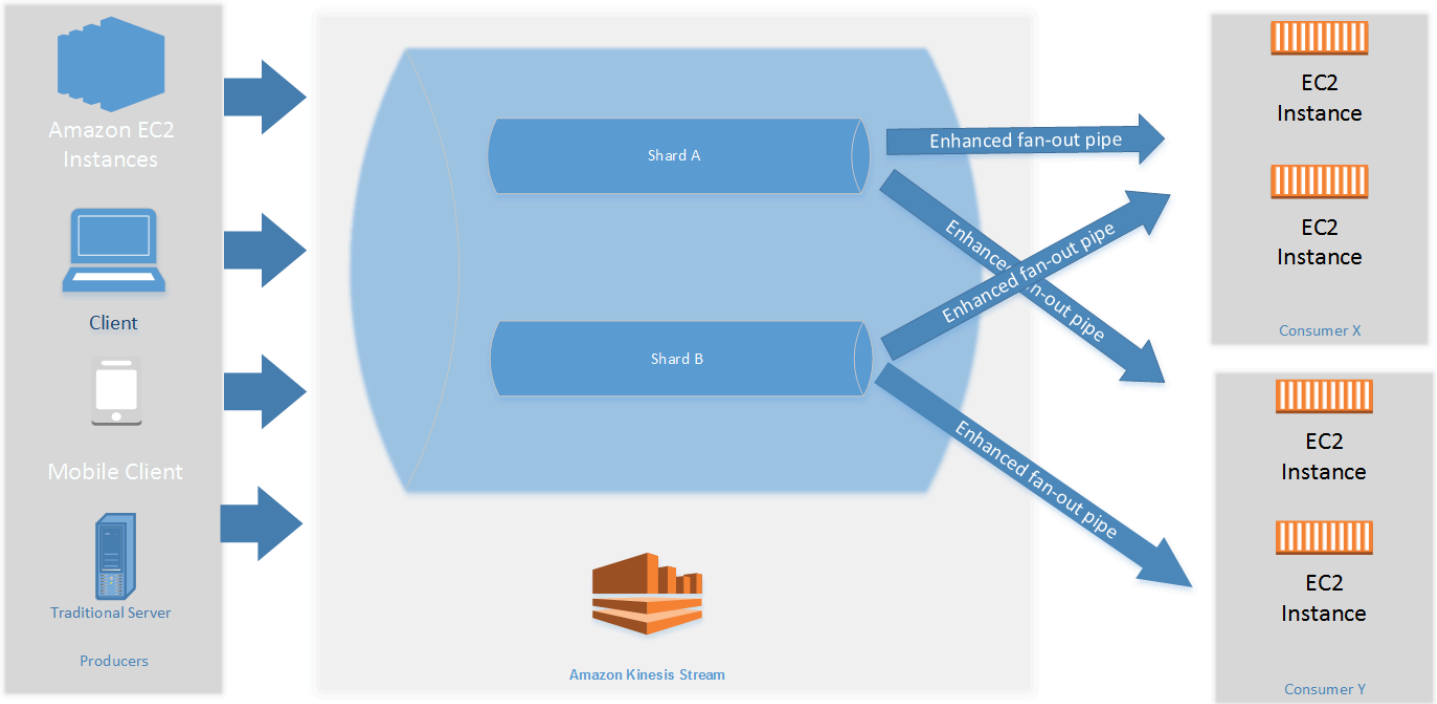
- Develop enhanced fan-out consumers with dedicated throughput
- Use the Data Viewer in the Kinesis console
- Query your data streams in the Kinesis console
- Use Kinesis Client Library
- Develop consumers with the AWS SDK for Java
- Develop consumers using AWS Lambda
- Develop consumers using Amazon Managed Service for Apache Flink
- Develop consumers using Amazon Data Firehose
- Read data from Kinesis Data Streams using other AWS services
- Read from Kinesis Data Streams using third-party integrations
- Troubleshoot Kinesis Data Streams consumers
- Optimize Amazon Kinesis Data Streams consumers

Develop enhanced fan-out consumers with dedicated throughput

In Amazon Kinesis Data Streams, you can build consumers that use a feature called *enhanced fan-out*. This feature lets consumers receive records from a stream with throughput of up to 2 MB of data per second per shard. This throughput is dedicated, which means that consumers that use enhanced fan-out don't have to contend with other consumers that are receiving data from the stream. Kinesis Data Streams pushes data records from the stream to consumers that use enhanced fan-out. Therefore, these consumers don't need to poll for data.

> **Important**
>
> With On-demand Advantage mode, you can register up to 50 consumers per stream to use enhanced fan-out. With On-demand Standard and Provisioned streams, you can register up to 20 consumers per stream to use enhanced fan-out.

The following diagram shows the enhanced fan-out architecture. If you use version 2.0 or later of the Amazon Kinesis Client Library (KCL) to build a consumer, the KCL sets up the consumer to use enhanced fan-out to receive data from all the shards of the stream. If you use the API to build a consumer that uses enhanced fan-out, then you can subscribe to individual shards.



The diagram shows the following:

- A stream with two shards.
- Two consumers that are using enhanced fan-out to receive data from the stream: Consumer X and Consumer Y. Each of the two consumers is subscribed to all of the shards and all of the records of the stream. If you use version 2.0 or later of the KCL to build a consumer, the KCL automatically subscribes that consumer to all the shards of the stream. On the other hand, if you use the API to build a consumer, you can subscribe to individual shards.
- Arrows representing the enhanced fan-out pipes that the consumers use to receive data from the stream. An enhanced fan-out pipe provides up to 2 MB/sec of data per shard, independently of any other pipes or of the total number of consumers.

Topics

- Differences between shared throughput consumer and enhanced fan-out consumer
- Regions supported for up to 50 enhanced fan-out consumers (On-demand Advantage only)
- Manage enhanced fan-out consumers with the AWS CLI or APIs

Differences between shared throughput consumer and enhanced fan-out consumer

The following table compares default shared-throughput consumers to enhanced fan-out consumers. Message propagation delay is defined as the time taken in milliseconds for a payload sent using the payload-dispatching APIs (like `PutRecord` and `PutRecords`) to reach the consumer application through the payload-consuming APIs (like `GetRecords` and `SubscribeToShard`).

This table compares shared-throughput consumers to enhanced fan-out consumers

Characteristics	Shared throughput consumers without enhanced fan-out	Enhanced fan-out consumers
Read throughput	Fixed at a total of 2 MB/sec per shard. If there are multiple consumers reading from the same shard, they all share this throughput.	Scales as consumers register to use enhanced fan-out. Each consumer registered to use enhanced fan-out receives its own read throughpu

Characteristics	Shared throughput consumers without enhanced fan-out	Enhanced fan-out consumers
	The sum of the throughputs they receive from the shard doesn't exceed 2 MB/sec.	t per shard, up to 2 MB/sec, independently of other consumers.
Message propagation delay	An average of around 200 ms if you have one consumer reading from the stream. This average goes up to around 1000 ms if you have five consumers.	Typically an average of 70 ms whether you have one consumer or five consumers.
Cost	Not applicable	There is a data retrieval cost and a consumer-shard hour cost. For more information, see Amazon Kinesis Data Streams Pricing.
Record delivery model	Pull model over HTTP using GetRecords.	Kinesis Data Streams pushes the records to you over HTTP/2 using Subscribe ToShard.

Regions supported for up to 50 enhanced fan-out consumers (On-demand Advantage only)

Support for up to 50 enhanced fan-out consumers in On-demand Advantage mode is available only in the following AWS Regions:

AWS Region	Region Name
eu-north-1	Europe (Stockholm)
me-south-1	Middle East (Bahrain)

AWS Region	Region Name
mx-central-1	Mexico (Central)
ap-south-1	Asia Pacific (Mumbai)
eu-west-3	Europe (Paris)
ap-southeast-3	Asia Pacific (Jakarta)
us-east-2	US East (Ohio)
af-south-1	Africa (Cape Town)
eu-west-1	Europe (Ireland)
me-central-1	Middle East (UAE)
eu-central-1	Europe (Frankfurt)
sa-east-1	South America (São Paulo)
ap-east-1	Asia Pacific (Hong Kong)
ap-east-2	Asia Pacific (Taipei)
ap-south-2	Asia Pacific (Hyderabad)
us-east-1	US East (N. Virginia)
ap-northeast-2	Asia Pacific (Seoul)
ap-northeast-3	Asia Pacific (Osaka)
eu-west-2	Europe (London)
ap-southeast-4	Asia Pacific (Melbourne)
ap-southeast-5	Asia Pacific (Malaysia)
ap-southeast-6	Asia Pacific (New Zealand)

AWS Region	Region Name
ap-southeast-7	Asia Pacific (Thailand)
ap-northeast-1	Asia Pacific (Tokyo)
us-west-2	US West (Oregon)
us-west-1	US West (N. California)
ap-southeast-1	Asia Pacific (Singapore)
ap-southeast-2	Asia Pacific (Sydney)
il-central-1	Israel (Tel Aviv)
ca-central-1	Canada (Central)
ca-west-1	Canada West (Calgary)
eu-south-2	Europe (Spain)
eu-south-1	Europe (Milan)
cn-northwest-1	China (Ningxia)
eu-central-2	Europe (Zurich)
us-gov-east-1	AWS GovCloud (US-East)
us-gov-west-1	AWS GovCloud (US-West)

Manage enhanced fan-out consumers with the AWS CLI or APIs

Consumers that use *enhanced fan-out* in Amazon Kinesis Data Streams can receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. For more information, see Develop enhanced fan-out consumers with dedicated throughput.

You can use AWS CLI or Kinesis Data Streams APIs to register, describe, list, and deregister a consumer that uses enhanced fan-out in Kinesis Data Streams.

Manage consumers using the AWS CLI

You can register, describe, list, and deregister enhanced fan-out consumers using the AWS CLI. For examples, see the following documentation.

register-stream-consumer

Registers a consumer for a Kinesis data stream. You can apply tags while registering the consumer.

describe-stream-consumer

Gets the description of a registered consumer with either consumer ARN or consumer name and stream ARN combination.

list-stream-consumers

Lists the consumers registered to receive data from a stream using enhanced fan-out.

deregister-stream-consumer

Deregister a consumer with either consumer ARN or consumer name and stream ARN combination.

Manage consumers using the Kinesis Data Streams APIs

You can register, describe, list, and deregister enhanced fan-out consumers using the Kinesis Data Streams APIs. For examples, see the following documentation.

RegisterStreamConsumer

Registers a consumer for a Kinesis data stream with tags. You can apply tags while registering the consumer.

DescribeStreamConsumer

Gets the description of a registered consumer with either consumer ARN or consumer name and stream ARN combination.

ListStreamConsumers

Lists the consumers registered to receive data from a stream using enhanced fan-out.

DeregisterStreamConsumer

Deregister a consumer with either consumer ARN or consumer name and stream ARN combination.

Tagging consumers

You can assign your own metadata to streams and enhanced fan-out consumers you create in Kinesis Data Streams in the form of tags. You can use tags to categorize and track costs of your consumers. You can also control access to consumers using tags with attribute-based access control (ABAC). For more information, see Tag your Amazon Kinesis Data Streams resources.

Use the Data Viewer in the Kinesis console

The Data Viewer in the Kinesis Management Console lets you view data records within the specified shard of your data stream without having to develop a consumer application. To use the Data Viewer, follow these steps:

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. Choose the active data stream whose records you want to view with the Data Viewer and then choose the **Data Viewer** tab.
3. In the **Data Viewer** tab for the selected active data stream, choose the shard whose records you want to view, chose the **Starting Position**, and then click **Get records**. You can set the starting position to one of the following values:

   - **At sequence number**: Show records from the position denoted by the sequence number specified in the sequence number field.
   - **After sequence number**: Show records right after the position denoted by the sequence number specified in the sequence number field.
   - **At timestamp**: Show records from the position denoted by the time stamp specified in the timestamp field.
   - **Trim horizon**: Show records at the last untrimmed record in the shard, which is the oldest data record in the shard.
   - **Latest**: Show records just after the most recent record in the shard, so that you always read the most recent data in the shard.

The generated data records that match the specified shard ID and starting position are then displayed in a records table in the console. Maximum 50 records are displayed at a time. To view the next set of records, click the **Next** button.

4. Click any individual record to view that record payload in raw data or JSON format in a separate window.

Note that when you click **Get records** or **Next** buttons in **Data Viewer**, this invokes the **GetRecords** API and this applies towards the **GetRecords** API limit of 5 transactions per second.

Query your data streams in the Kinesis console

The Data Analytics tab in the Kinesis Data Streams Console lets you query your data streams using SQL. To use this capability, follow these steps:

1. Sign in to the AWS Management Console and open the Kinesis console at https://eusc-de-east-1.console.amazonaws-eusc.eu/kinesis.
2. Choose the active data stream that you want to query with SQL and then choose the **Data analytics** tab.
3. In the **Data analytics** tab, you can perform stream inspection and visualization with a Managed Apache Flink Studio notebook. You can perform ad-hoc SQL queries to inspect your data stream and view results in seconds using Apache Zeppelin. In the **Data analytics** tab, choose **I agree** and then choose **Create notebook** to create a notebook.
4. After the notebook is created, choose **Open in Apache Zeppelin**. This will open your notebook in a new tab. A notebook is an interactive interface where you can submit your SQL queries. Choose the note that contains the name of your stream.
5. You will see a note with a sample SELECT query to output the data in the stream already running. This lets you view the schema for your data stream.
6. To try out other queries such as tumbling or sliding windows, choose **View sample queries** in the **Data analytics** tab. Copy the query, modify it to suit your data stream schema, and then run it in a new paragraph in your Zeppelin note.

Use Kinesis Client Library

What is Kinesis Client Library?

Kinesis Client Library (KCL) is a standalone Java software library designed to simplify the process of consuming and processing data from Amazon Kinesis Data Streams. KCL handles many of the complex tasks associated with distributed computing, letting developers focus on implementing their business logic for processing data. It manages activities such as load balancing across multiple workers, responding to worker failures, checkpointing processed records, and responding to changes in the number of shards in the stream.

KCL is frequently updated to incorporate newer versions of underlying libraries, security improvements, and bug fixes. We recommend that you use the latest version of KCL to avoid known issues and benefit from all latest improvements. To find the latest KCL version, see KCL Github.

Important

- We recommend that you use the latest KCL version to avoid known bugs and issues. If you are using KCL 2.6.0 or earlier, upgrade to KCL 2.6.1 or later to avoid a rare condition that can block shard processing when stream capacity changes.
- KCL is a Java library. Support for languages other than Java is provided using a Java-based daemon called MultiLangDaemon. MultiLangDaemon interacts with the KCL application over STDIN and STDOUT. For more information about the MultiLangDaemon on GitHub, see Develop consumers with KCL in non-Java languages.
- Do not use AWS SDK for Java version 2.27.19 to 2.27.23 with KCL 3.x. These versions include an issue that causes an exception error related to KCL's DynamoDB usage. We recommend that you use the AWS SDK for Java version 2.28.0 or later to avoid this issue.

KCL key features and benefits

Following are the key features and related benefits of the KCL:

- **Scalability**: KCL enables applications to scale dynamically by distributing the processing load across multiple workers. You can scale your application in or out, manually or with auto-scaling, without worrying about load redistribution.

- **Load balancing**: KCL automatically balances the processing load across available workers, resulting in an even distribution of work across workers.
- **Checkpointing**: KCL manages checkpointing of processed records, enabling applications to resume processing from their last successfully processed position.
- **Fault tolerance**: KCL provides built-in fault tolerance mechanisms, making sure that data processing continues even if individual workers fail. KCL also provides at-least-once delivery.
- **Handling stream-level changes**: KCL adapts to shard splits and merges that might occur due to changes in data volume. It maintains ordering by making sure that child shards are processed only after their parent shard is completed and checkpointed.
- **Monitoring**: KCL integrates with Amazon CloudWatch for consumer-level monitoring.
- **Multi-language support**: KCL natively supports Java and enables multiple non-Java programming languages through MultiLangDaemon.

KCL concepts

This section explains the core concepts and interactions of Kinesis Client Library (KCL). These concepts are fundamental to developing and managing KCL consumer applications.

- **KCL consumer application** – a custom-built application designed to read and process records from Kinesis data streams using the Kinesis Client Library.
- **Worker** – KCL consumer applications are typically distributed, with one or more workers running simultaneously. KCL coordinates workers to consume data from the stream in a distributed manner and balances the load evenly across multiple workers.
- **Scheduler** – a high-level class that a KCL worker uses to start processing data. Each KCL worker has one scheduler. The scheduler initializes and oversees various tasks, including syncing shard information from Kinesis data streams, tracking shard assignments among workers, and processing data from the stream based on the assigned shards to the worker. Scheduler can take various configurations that affect the scheduler's behavior, such as the name of the stream to process and AWS credentials. Scheduler initiates the delivery of data records from the stream to the record processors.
- **Record processor** – defines the logic for how your KCL consumer application processes the data it receives from the data streams. You must implement your own custom data processing logic in the record processor. A KCL worker instantiates a scheduler. The scheduler then instantiates one record processor for every shard to which it holds a lease. A worker can run multiple record processors.

- **Lease** – defines the assignment between a worker and a shard. KCL consumer applications use leases to distribute data record processing across multiple workers. Each shard is bound to only one worker by a lease at any given time and each worker can hold one or more leases simultaneously. When a worker stops holding a lease due to stopping or failing, KCL assigns another worker to take the lease. To learn more about the lease, see Github documentation: Lease Lifecycle.
- **Lease table** – is a unique Amazon DynamoDB table used to track all leases for the KCL consumer application. Each KCL consumer application creates its own lease table. The lease table is used to maintain state across all workers to coordinate data processing. For more information, see DynamoDB metadata tables and load balancing in KCL.
- **Checkpointing** – is the process of persistently storing the position of the last successfully processed record in a shard. KCL manages checkpointing to make sure that processing can be resumed from the last checkpointed position if a worker fails or the application restarts. Checkpoints are stored in the DynamoDB lease table as part of the metadata of the lease. This allows workers to continue processing from where the previous worker stopped.

DynamoDB metadata tables and load balancing in KCL

KCL manages metadata such as leases and CPU utilization metrics from workers. KCL tracks these metadata using DynamoDB tables. For each Amazon Kinesis Data Streams application, KCL 3.x creates three DynamoDB tables by default to manage the metadata: lease table, worker metrics table, and coordinator state table. Starting with KCL 3.5, you can use the single table format to consolidate all metadata into the lease table. For more information, see Single table format for KCL.

Note

KCL 3.x introduced two new metadata tables: *worker metrics* and *coordinator state* tables. Starting with KCL 3.5, you can consolidate these tables into the lease table using the single table format.

Important

You must add proper permissions for KCL applications to create and manage metadata tables in DynamoDB. For details, see IAM permissions required for KCL consumer applications.

KCL consumer application does not automatically remove these three DynamoDB metadata tables. Make sure that you remove these DynamoDB metadata tables created by KCL consumer application when you decommission your consumer application to prevent unnecessary cost.

Lease table

A lease table is a unique Amazon DynamoDB table used to track the shards being leased and processed by the schedulers of the KCL consumer application. Each KCL consumer application creates its own lease table. KCL uses the name of the consumer application for the name of the lease table by default. You can set a custom table name using configuration. KCL also creates a global secondary index on the lease table with the partition key of leaseOwner for an efficient lease discovery. Global secondary index mirrors the leaseKey attribute from the base lease table. If the lease table for your KCL consumer application does not exist when the application starts up, one of the workers creates the lease table for your application.

You can view the lease table using the Amazon DynamoDB console while the consumer application is running.

Important

- Each KCL consumer application name must be unique to prevent a duplicated lease table name.
- Your account is charged for the costs associated with the DynamoDB table, in addition to the costs associated with Kinesis Data Streams itself.

Each row in the lease table represents a shard that is being processed by the schedulers of your consumer application. Key fields include the following:

- **leaseKey:** For single-stream processing, this is the shard ID. For multi-stream processing with KCL, it's structured as `account-id:StreamName:streamCreationTimestamp:ShardId`. leaseKey is the partition key of the lease table. For more information about multi-stream processing, see Multi-stream processing with KCL.
- **checkpoint:** The most recent checkpoint sequence number for the shard.

- **checkpointSubSequenceNumber:** When using the Kinesis Producer Library's aggregation feature, this is an extension to **checkpoint** that tracks individual user records within the Kinesis record.
- **leaseCounter:** Used for checking if a worker is currently processing the lease actively. leaseCounter increases if the lease ownership is transferred to another worker.
- **leaseOwner:** The current worker that is holding this lease.
- **ownerSwitchesSinceCheckpoint:** How many times this lease has changed workers since the last checkpoint.
- **parentShardId:** ID of this shard's parent. Makes sure that the parent shard is fully processed before processing starts on the child shards, maintaining the correct record processing order.
- **childShardId:** List of child shard IDs resulting from this shard's split or merge. Used to track shard lineage and manage processing order during resharding operations.
- **startingHashKey:** The lower bound of the hash key range for this shard.
- **endingHashKey:** The upper bound of the hash key range for this shard.

If you use multi-stream processing with KCL, you see the following two additional fields in the lease table. For more information, see Multi-stream processing with KCL.

- **shardID:** The ID of the shard.
- **streamName:** The identifier of the data stream in the following format: `account-id:StreamName:streamCreationTimestamp`.

Worker metrics table

Worker metrics table is a unique Amazon DynamoDB table for each KCL application and is used to record CPU utilization metrics from each worker. These metrics will be used by KCL to perform efficient lease assignments to result in balanced resource utilization across workers. KCL uses `KCLApplicationName-WorkerMetricStats` for the name of the worker metrics table by default.

Coordinator state table

A coordinator state table is a unique Amazon DynamoDB table for each KCL application and is used to store internal state information for workers. For example, the coordinator state table stores data regarding the leader election or metadata associated with the in-place migration from KCL 2.x to

KCL 3.x. KCL uses `KCLApplicationName-CoordinatorState` for the name of the coordinator state table by default.

DynamoDB capacity mode for metadata tables created by KCL

By default, the Kinesis Client Library (KCL) creates DynamoDB metadata tables such as lease table, worker metrics table, and coordinator state table using the on-demand capacity mode. This mode automatically scales read and write capacity to accommodate traffic without requiring capacity planning. We strongly recommend you to keep the capacity mode as on-demand mode for more efficient operation of these metadata tables.

If you decide to switch the lease table to provisioned capacity mode, follow these best practices:

- Analyze usage patterns:
  - Monitor your application's read and write patterns and usages (RCU, WCU) using Amazon CloudWatch metrics.
  - Understand peak and average throughput requirements.
- Calculate the required capacity:
  - Estimate read capacity units (RCUs) and write capacity units (WCUs) based on your analysis.
  - Consider factors like the number of shards, checkpoint frequency, and worker count.
- Implement auto scaling:
  - Use DynamoDB auto scaling to automatically adjust provisioned capacity and set appropriate minimum and maximum capacity limits.
  - DynamoDB auto scaling will help to avoid your KCL metadata table from hitting the capacity limit and getting throttled.
- Regular monitoring and optimization:
  - Continuously monitor CloudWatch metrics for `ThrottledRequests`.
  - Adjust capacity as your workload changes over time.

If you experience a `ProvisionedThroughputExceededException` in metadata DynamoDB tables for your KCL consumer application, you must increase the provisioned throughput capacity of the DynamoDB table. If you set a certain level of read capacity units (RCU) and write capacity units (WCU) when you first create your consumer application, it might not be sufficient as your usage grows. For example, if your KCL consumer application does frequent checkpointing or operates on a stream with many shards, you might need more capacity units. For information

about provisioned throughput in DynamoDB, see DynamoDB throughput capacity and updating a table in the Amazon DynamoDB Developer Guide.

How KCL assigns leases to workers and balances the load

KCL continuously gathers and monitors CPU utilization metrics from compute hosts running the workers to ensure even workload distribution. These CPU utilization metrics are stored in the worker metrics table in DynamoDB. If KCL detects that some workers are showing higher CPU utilization rates compared to others, it will reassign leases among workers to lower the load on highly used workers. The goal is to balance the workload more evenly across the consumer application fleet, preventing any single worker from becoming overloaded. As KCL distributes CPU utilization across the consumer application fleet, you can right-size your consumer application fleet capacity by choosing the right number of workers or use auto scaling to efficiently manage the computing capacity to achieve lower cost.

> **Important**
>
> KCL can collect CPU utilization metrics from workers only if certain prerequisites are met. For details, see Prerequisites. If KCL cannot collect CPU utilization metrics from workers, KCL will fall back to using throughput per worker to assign leases and balance the load across workers in the fleet. KCL will monitor the throughput that each worker receives at a given time and reassign leases to make sure that each worker gets a similar total throughput level from its assigned leases.

Develop consumers with KCL

You can use the Kinesis Client Library (KCL) to build consumer applications that process data from your Kinesis data streams.

KCL is available in multiple languages. This topic covers how to develop KCL consumers in Java and non-Java languages.

- To view the Kinesis Client Library Javadoc reference, see the Amazon Kinesis Client Library Javadoc.
- To download KCL for Java from GitHub, see Amazon Kinesis Client Library for Java.
- To locate the KCL for Java on Apache Maven, see the KCL Maven Central Repository.

Topics

- Develop consumers with KCL in Java
- Develop consumers with KCL in non-Java languages

Develop consumers with KCL in Java

Prerequisites

Before you start using KCL 3.x, make sure that you have the following:

- Java Development Kit (JDK) 8 or later
- AWS SDK for Java 2.x
- Maven or Gradle for dependency management

KCL collects CPU utilization metrics such as CPU utilization from the compute host that workers are running on to balance the load to achieve an even resource utilization level across workers. To enable KCL to collect CPU utilization metrics from workers, you must meet the following prerequisites:

Amazon Elastic Compute Cloud(Amazon EC2)

- Your operating system must be Linux OS.
- You must enable IMDSv2 in your EC2 instance.

Amazon Elastic Container Service (Amazon ECS) on Amazon EC2

- Your operating system must be Linux OS.
- You must enable ECS task metadata endpoint version 4.
- Your Amazon ECS container agent version must be 1.39.0 or later.

Amazon ECS on AWS Fargate

- You must enable Fargate task metadata endpoint version 4. If you use Fargate platform version 1.4.0 or later, this is enabled by default.
- Fargate platform version 1.4.0 or later.

Amazon Elastic Kubernetes Service (Amazon EKS) on Amazon EC2

- Your operating system must be Linux OS.

Amazon EKS on AWS Fargate

- Fargate platform 1.3.0 or later.

> **Important**
>
> If KCL cannot collect CPU utilization metrics from workers, KCL will fall back to use throughput per worker to assign leases and balance the load across workers in the fleet. For more information, see How KCL assigns leases to workers and balances the load.

Install and add dependencies

If you're using Maven, add the following dependency to your `pom.xml` file. Make sure you replaced 3.x.x to the latest KCL version.

```
<dependency>
    <groupId>software.amazon.kinesis</groupId>
    <artifactId>amazon-kinesis-client</artifactId>
    <version>3.x.x</version> <!-- Use the latest version -->
</dependency>
```

If you're using Gradle, add the following to your `build.gradle` file. Make sure you replaced 3.x.x to the latest KCL version.

```
implementation 'software.amazon.kinesis:amazon-kinesis-client:3.x.x'
```

You can check for the latest version of the KCL on the Maven Central Repository.

Implement the consumer

A KCL consumer application consists of the following key components:

Key components

- RecordProcessor

- RecordProcessorFactory
- Scheduler
- Main Consumer Application

RecordProcessor

RecordProcessor is the core component where your business logic for processing Kinesis data stream records resides. It defines how your application processes the data it receives from the Kinesis stream.

Key responsibilities:

- Initialize processing for a shard
- Process batches of records from the Kinesis stream
- Shutdown processing for a shard (for example, when the shard splits or merges, or the lease is handed over to another host)
- Handle checkpointing to track progress

The following shows an implementation example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.*;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public class SampleRecordProcessor implements ShardRecordProcessor {
    private static final String SHARD_ID_MDC_KEY = "ShardId";
    private static final Logger log =
 LoggerFactory.getLogger(SampleRecordProcessor.class);
    private String shardId;

    @Override
    public void initialize(InitializationInput initializationInput) {
        shardId = initializationInput.shardId();
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
```

```
            log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Processing {} record(s)", processRecordsInput.records().size());
            processRecordsInput.records().forEach(r ->
                log.info("Processing record pk: {} -- Seq: {}", r.partitionKey(),
r.sequenceNumber())
            );

            // Checkpoint periodically
            processRecordsInput.checkpointer().checkpoint();
        } catch (Throwable t) {
            log.error("Caught throwable while processing records. Aborting.", t);
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Lost lease, so terminating.");
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Reached shard end checkpointing.");
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            log.error("Exception while checkpointing at shard end. Giving up.", e);
```

```
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        MDC.put(SHARD_ID_MDC_KEY, shardId);
        try {
            log.info("Scheduler is shutting down, checkpointing.");
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
        } finally {
            MDC.remove(SHARD_ID_MDC_KEY);
        }
    }
}
```

The following is a detailed explanation of each method used in the example:

initialize(InitializationInput initializationInput)

- Purpose: Set up any necessary resources or state for processing records.
- When it's called: Once, when KCL assigns a shard to this record processor.
- Key points:
  - `initializationInput.shardId()`: The ID of the shard this processor will handle.
  - `initializationInput.extendedSequenceNumber()`: The sequence number to start processing from.

processRecords(ProcessRecordsInput processRecordsInput)

- Purpose: Process the incoming records and optionally checkpoint progress.
- When it's called: Repeatedly, as long as the record processor holds the lease for the shard.
- Key points:
  - `processRecordsInput.records()`: List of records to process.
  - `processRecordsInput.checkpointer()`: Used to checkpoint the progress.
  - Make sure that you handled any exceptions during processing to prevent KCL from failing.

- This method should be idempotent, as the same record may be processed more than once in some scenarios, such as data that has not been checkpointed before unexpected worker crashes or restarts.
- Always flush any buffered data before checkpointing to ensure data consistency.

leaseLost(LeaseLostInput leaseLostInput)

- Purpose: Clean up any resources specific to processing this shard.
- When it's called: When another Scheduler takes over the lease for this shard.
- Key points:
  - Checkpointing is not allowed in this method.

shardEnded(ShardEndedInput shardEndedInput)

- Purpose: Finish processing for this shard and checkpoint.
- When it's called: When the shard splits or merges, indicating all data for this shard has been processed.
- Key points:
  - `shardEndedInput.checkpointer()`: Used to perform the final checkpointing.
  - Checkpointing in this method is mandatory to complete processing.
  - Failing to flush data and checkpoint here may result in data loss or duplicate processing when the shard is reopened.

shutdownRequested(ShutdownRequestedInput shutdownRequestedInput)

- Purpose: Checkpoint and clean up resources when KCL is shutting down.
- When it's called: When KCL is shutting down, for example, when the application is terminating).
- Key points:
  - `shutdownRequestedInput.checkpointer()`: Used to perform checkpointing before shutdown.
  - Make sure you implemented checkpointing in the method so that progress is saved before the application stops.

- Failure to flush data and checkpoint here may result in data loss or reprocessing of records when the application restarts.

> **Important**
>
> KCL 3.x ensures fewer data reprocessing when the lease is handed over from one worker to another worker by checkpointing before the previous worker is shut down. If you don't implement the checkpointing logic in the `shutdownRequested()` method, you won't see this benefit. Make sure that you have implemented a checkpointing logic inside the `shutdownRequested()` method.

RecordProcessorFactory

RecordProcessorFactory is responsible for creating new RecordProcessor instances. KCL uses this factory to create a new RecordProcessor for each shard that the application needs to process.

Key responsibilities:

- Create new RecordProcessor instances on demand
- Make sure that each RecordProcessor is properly initialized

The following is an implementation example:

```
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleRecordProcessorFactory implements ShardRecordProcessorFactory {
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new SampleRecordProcessor();
    }
}
```

In this example, the factory creates a new SampleRecordProcessor each time shardRecordProcessor() is called. You can extend this to include any necessary initialization logic.

Scheduler

Scheduler is a high-level component that coordinates all the activities of the KCL application. It's responsible for the overall orchestration of data processing.

Key responsibilities:

- Manage the lifecycle of RecordProcessors
- Handle lease management for shards
- Coordinate checkpointing
- Balance shard processing load across multiple workers of your application
- Handle graceful shutdown and application termination signals

Scheduler is typically created and started in the Main Application. You can check the implementation example of Scheduler in the following section, Main Consumer Application.

Main Consumer Application

Main Consumer Application ties all the components together. It's responsible for setting up the KCL consumer, creating necessary clients, configuring the Scheduler, and managing the application's lifecycle.

Key responsibilities:

- Set up AWS service clients (Kinesis, DynamoDB, CloudWatch)
- Configure the KCL application
- Create and start the Scheduler
- Handle application shutdown

The following is an implementation example:

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import java.util.UUID;
```

```
public class SampleConsumer {
    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    public SampleConsumer(String streamName, Region region) {
        this.streamName = streamName;
        this.region = region;
        this.kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    public void run() {
        DynamoDbAsyncClient dynamoDbAsyncClient =
 DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();

        ConfigsBuilder configsBuilder = new ConfigsBuilder(
            streamName,
            streamName,
            kinesisClient,
            dynamoDbAsyncClient,
            cloudWatchClient,
            UUID.randomUUID().toString(),
            new SampleRecordProcessorFactory()
        );

        Scheduler scheduler = new Scheduler(
            configsBuilder.checkpointConfig(),
            configsBuilder.coordinatorConfig(),
            configsBuilder.leaseManagementConfig(),
            configsBuilder.lifecycleConfig(),
            configsBuilder.metricsConfig(),
            configsBuilder.processorConfig(),
            configsBuilder.retrievalConfig()
        );

        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();
    }
```

```
    public static void main(String[] args) {
        String streamName = "your-stream-name"; // replace with your stream name
        Region region = Region.US_EAST_1; // replace with your region
        new SampleConsumer(streamName, region).run();
    }
}
```

KCL creates an Enhanced Fan-out (EFO) consumer with dedicated throughput by default. For more information about Enhanced Fan-out, see Develop enhanced fan-out consumers with dedicated throughput. If you have less than 2 consumers or don't need read propagation delays under 200 ms, you must set the following configuration in the scheduler object to use shared-throughput consumers:

```
configsBuilder.retrievalConfig().retrievalSpecificConfig(new PollingConfig(streamName,
 kinesisClient))
```

The following code is an example of creating a scheduler object that uses shared-throughput consumers:

Imports:

```
import software.amazon.kinesis.retrieval.polling.PollingConfig;
```

Code:

```
Scheduler scheduler = new Scheduler(
            configsBuilder.checkpointConfig(),
            configsBuilder.coordinatorConfig(),
            configsBuilder.leaseManagementConfig(),
            configsBuilder.lifecycleConfig(),
            configsBuilder.metricsConfig(),
            configsBuilder.processorConfig(),
            configsBuilder.retrievalConfig().retrievalSpecificConfig(new
 PollingConfig(streamName, kinesisClient))
        );/
```

Develop consumers with KCL in non-Java languages

This section covers the implementation of consumers using Kinesis Client Library (KCL) in Python, Node.js, .NET, and Ruby.

KCL is a Java library. Support for languages other than Java is provided using a multi-language interface called the `MultiLangDaemon`. This daemon is Java-based and runs in the background when you are using a KCL with a language other than Java. Therefore, if you install KCL for non-Java languages and write your consumer app entirely in non-Java languages, you still need Java installed on your system because of the `MultiLangDaemon`. Further, `MultiLangDaemon` has some default settings you might need to customize for your use case (for example, the AWS region that it connects to). For more information about the `MultiLangDaemon` on GitHub, see KCL MultiLangDaemon project.

While the core concepts remain the same across languages, there are some language-specific considerations and implementations. For the core concepts about the KCL consumer development, see Develop consumers with KCL in Java. For more detailed information about how to develop KCL consumers in Python, Node.js, .NET, and Ruby and latest updates, please refer to the following GitHub repositories:

- Python: amazon-kinesis-client-python
- Node.js: amazon-kinesis-client-nodejs
- .NET: amazon-kinesis-client-net
- Ruby: amazon-kinesis-client-ruby

Important

Don't use the following non-Java KCL library versions if you're using JDK 8. These versions contain a dependency (logback) that is incompatible with JDK 8.

- KCL Python 3.0.2 and 2.2.0
- KCL Node.js 2.3.0
- KCL .NET 3.1.0
- KCL Ruby 2.2.0

We recommend that you use versions released either before or after these affected versions when working with JDK 8.

Multi-stream processing with KCL

This section describes the required changes in KCL that allow you to create KCL consumer applications that can process more than one data stream at the same time.

Important

- Multi-stream processing is only supported in KCL 2.3 or later.
- Multi-stream processing is *not* supported for KCL consumers written in non-Java languages that run with `multilangdaemon`.
- Multi-stream processing is *not* supported in any versions of KCL 1.x.

- **MultistreamTracker interface**
  - To build a consumer application that can process multiple streams at the same time, you must implement a new interface called MultistreamTracker. This interface includes the `streamConfigList` method that returns the list of data streams and their configurations to be processed by the KCL consumer application. Notice that the data streams that are being processed can be changed during the consumer application runtime. `streamConfigList` is called periodically by KCL to learn about the changes in data streams to process.
  - The `streamConfigList` populates the StreamConfig list.

```
package software.amazon.kinesis.common;

import lombok.Data;
import lombok.experimental.Accessors;

@Data
@Accessors(fluent = true)
public class StreamConfig {
    private final StreamIdentifier streamIdentifier;
    private final InitialPositionInStreamExtended initialPositionInStreamExtended;
    private String consumerArn;
}
```

  - The `StreamIdentifier` and `InitialPositionInStreamExtended` are required fields, while `consumerArn` is optional. You must provide the `consumerArn` only if you are using KCL to implement an enhanced fan-out consumer application.

- For more information about `StreamIdentifier`, see https://github.com/awslabs/amazon-kinesis-client/blob/v2.5.8/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L129. To create a `StreamIdentifier`, we recommend that you create a multistream instance from the `streamArn` and the `streamCreationEpoch` that is available in KCL 2.5.0 or later. In KCL v2.3 and v2.4, which don't support `streamArm`, create a multistream instance by using the format `account-id:StreamName:streamCreationTimestamp`. This format will be deprecated and no longer supported starting with the next major release.
- MultistreamTracker also includes a strategy for deleting leases of old streams in the lease table (formerStreamsLeasesDeletionStrategy). Notice that the strategy CANNOT be changed during the consumer application runtime. For more information, see https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java.

•

Or you can initialize ConfigsBuilder with `MultiStreamTracker` if you want to implement a KCL consumer application that processes multiple streams at the same time.

```
* Constructor to initialize ConfigsBuilder with MultiStreamTracker
     * @param multiStreamTracker
     * @param applicationName
     * @param kinesisClient
     * @param dynamoDBClient
     * @param cloudWatchClient
     * @param workerIdentifier
     * @param shardRecordProcessorFactory
     */
    public ConfigsBuilder(@NonNull MultiStreamTracker multiStreamTracker, @NonNull
 String applicationName,
            @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
 dynamoDBClient,
            @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
 workerIdentifier,
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.left(multiStreamTracker);
        this.applicationName = applicationName;
```

```
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

- With multi-stream support implemented for your KCL consumer application, each row of the application's lease table now contains the shard ID and the stream name of the multiple data streams that this application processes.
- When multi-stream support for your KCL consumer application is implemented, the leaseKey takes the following structure: `account-id:StreamName:streamCreationTimestamp:ShardId`. For example, `111111111:multiStreamTest-1:12345:shardId-000000000336`.

> **Important**
>
> When your existing KCL consumer application is configured to process only one data stream, the `leaseKey` (which is the partition key for the lease table) is the shard ID. If you reconfigure an existing KCL consumer application to process multiple data streams, it breaks your lease table, because the `leaseKey` structure must be as follows: `account-id:StreamName:StreamCreationTimestamp:ShardId` to support multi-stream.

Use the AWS Glue Schema registry with KCL

You can integrate Kinesis Data Streams with the AWS Glue Schema registry. The AWS Glue Schema registry lets you centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema registry lets you improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through KCL for Java.

Important

- AWS Glue Schema registry integration for Kinesis Data Streams is only supported in KCL 2.3 or later.
- AWS Glue Schema registry integration for Kinesis Data Streams is *not* supported for KCL consumers written in non-Java languages that run with `multilangdaemon`.
- AWS Glue Schema registry integration for Kinesis Data Streams is *not* supported in any versions of KCL 1.x.

For detailed instructions on how to set up integration of Kinesis Data Streamswith AWS Glue Schema registry using KCL, see the "Interacting with Data Using the KPL/KCL Libraries" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

IAM permissions required for KCL consumer applications

You must add the following permissions to the IAM role or user associated with your KCL consumer application.

Security best practices for AWS dictate the use of fine-grained permissions to control access to different resources. AWS Identity and Access Management (IAM) lets you manage users and user permissions in AWS. An IAM policy explicitly lists actions that are allowed and the resources on which the actions are applicable.

The following table shows the minimum IAM permissions generally required for KCL consumer applications:

Minimum IAM permissions for KCL consumer applications

Service	Actions	Resources (ARNs)	Purpose

Amazon Kinesis Data Streams	`DescribeStream`
	
	`DescribeStreamSummary`
	
	`RegisterStreamConsumer`
		Kinesis data stream from which your KCL application will process the data.
		
		`arn:aws:kinesis:re`
			Before attempting to read records, the consumer checks if the data stream exists, if it's active, and if the shards are

Service	Actions	Resources (ARNs)	Purpose
		`gion:acco unt:stream/ StreamName`	contained in the data stream. Registers consumers to a shard.

Amazon Kinesis Data Streams	`GetRecords`
	
	`GetShardI terator`
	
	`ListShards`
		Kinesis data stream from which your KCL application will process the data.
		
		`arn:aws:k inesis:re gion:acco unt:stream/ StreamName`
			Reads records from a shard.

Amazon Kinesis Data Streams	`Subscribe ToShard`
	
	`DescribeS treamConsumer`
		Kinesis data stream from which your KCL application will process the data. Add this action only if you use enhanced fan-out (EFO) consumers.
		
		`arn:aws:k inesis:re gion:acco unt:stream/ StreamName/ consumer/*`
			Subscribes to a shard for enhanced fan-out (EFO) consumers.

Service	Actions	Resources (ARNs)	Purpose

Amazon DynamoDB	`CreateTable`
	`DescribeTable`
	`UpdateTable`
	`Scan`
	`GetItem`
	`PutItem`
	`UpdateItem`
	`DeleteItem`
		Lease table (metadata table in DynamoDB created by KCL.
		`arn:aws:dynamodb:region:account:table/KCLApplicationName`
			These actions are required for KCL to manag the lease table created in DynamoDB.

Service	Actions	Resources (ARNs)	Purpose

Amazon DynamoDB	`CreateTable`
	`DescribeTable`
	`Scan`
	`GetItem`
	`PutItem`
	`UpdateItem`
	`DeleteItem`
	`Condition CheckItem`
		Worker metrics and coordinator state table (metadata tables in DynamoDB) created by KCL.
		`arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats`
		`arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorState`
			These actions are required for KCL to manage the worker metrics and coordinator state metadata tables in DynamoDB. The CoordinatorState table requires the `Condition CheckItem` action to support the conditional writes used during the single table migration.

Amazon DynamoDB	`Query`	Global secondary index on the lease table. `arn:aws:dynamodb:region:account:table/KCLApplicationName/index/*`	This action is required for KCL to read the global secondary index of the lease table created in DynamoDB.

Service	Actions	Resources (ARNs)	Purpose
Amazon CloudWatch	`PutMetricData`	*	Upload metrics to CloudWatch that are useful for monitoring the application. The asterisk (*) is used because there is no spcific resource in CloudWatch on which the `PutMetricData` action is invoked.

Note

Replace "region," "account," "StreamName," and "KCLApplicationName" in the ARNs with your own AWS Region, AWS account number, Kinesis data stream name, and KCL application name respectively. KCL 3.x creates two more metadata tables in DynamoDB. For details about DynamoDB metadata tables created by KCL, see DynamoDB metadata tables and load balancing in KCL. If you use configurations to customize names of the metadata tables created by KCL, use those specified table names instead of KCL application name.

Note

After KCL 3.5 single table migration completes, the legacy worker metrics and coordinator state tables must be manually deleted.

The following is an example policy document for a KCL consumer application.

JSON

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

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream",
                "kinesis:DescribeStreamSummary",
                "kinesis:RegisterStreamConsumer",
                "kinesis:GetRecords",
                "kinesis:GetShardIterator",
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws-eusc:kinesis:eusc-de-
east-1:123456789012:stream/STREAM_NAME"
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamConsumer"
            ],
            "Resource": "arn:aws-eusc:kinesis:eusc-de-
east-1:123456789012:stream/STREAM_NAME/consumer/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:DescribeTable",
                "dynamodb:UpdateTable",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:DeleteItem",
                "dynamodb:Scan"
            ],
            "Resource": [
            "arn:aws-eusc:dynamodb:eusc-de-
east-1:123456789012:table/KCL_APPLICATION_NAME"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
```

```
                "dynamodb:DescribeTable",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:DeleteItem",
                "dynamodb:Scan"
            ],
            "Resource": [
            "arn:aws-eusc:dynamodb:eusc-de-
east-1:123456789012:table/KCL_APPLICATION_NAME-WorkerMetricStats",
    "arn:aws-eusc:dynamodb:eusc-de-
east-1:123456789012:table/KCL_APPLICATION_NAME-CoordinatorState"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query"
            ],
            "Resource": [
            "arn:aws-eusc:dynamodb:eusc-de-
east-1:123456789012:table/KCL_APPLICATION_NAME/index/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": "*"
        }
    ]
}
```

Before you use this example policy, check the following items:

- Replace REGION with your AWS Region (for example, eusc-de-east-1).
- Replace ACCOUNT_ID with your AWS account ID.
- Replace STREAM_NAME with the name of your Kinesis data stream.
- Replace CONSUMER_NAME with the name of your consumer, typically your application name when using KCL.

- Replace KCL_APPLICATION_NAME with the name of your KCL application.

KCL configurations

You can set configuration properties to customize Kinesis Client Library's functionality to meet your specific requirements. The following table describes configuration properties and classes.

> **Important**
>
> In KCL 3.x, the load balancing algorithm aims to achieve even CPU utilization across workers, not an equal number of leases per worker. Setting `maxLeasesForWorker` too low, you might limit KCL's ability to balance the workload effectively. If you use the `maxLeasesForWorker` configuration, consider increasing its value to allow for the best possible load distribution.

This table shows the configuration properties for KCL

Configuration property	Configuration class	Description	Default value
`applicationName`	ConfigsBuilder	The name for this the KCL application. Used as the default for the `tableName` and `consumerName` .	Not applicable
`tableName`	ConfigsBuilder	Allows overridin g the table name used for the Amazon DynamoDB lease table.	Not applicable
`streamName`	ConfigsBuilder	The name of the stream that this application processes records from.	Not applicable

Configuration property	Configuration class	Description	Default value
`workerIdentifier`	ConfigsBuilder	A unique identifier that represents this instantiation of the application processor. This must be unique.	Not applicable
`failoverTimeMillis`	LeaseManagementConfig	The number of milliseconds that must pass before you can consider a lease owner to have failed. For applications that have a large number of shards, this may be set to a higher number to reduce the number of DynamoDB IOPS required for tracking leases.	10,000 (10 seconds)
`shardSyncIntervalMillis`	LeaseManagementConfig	The time between shard sync calls.	60,000 (60 seconds)
`cleanupLeasesUponShardCompletion`	LeaseManagementConfig	When set, leases are removed as soon as the child leases have started processing.	TRUE

Configuration property	Configuration class	Description	Default value
ignoreUnexpectedChildShards	LeaseManagementConfig	When set, child shards that have an open shard are ignored. This is primarily for DynamoDB Streams.	FALSE
maxLeasesForWorker	LeaseManagementConfig	The maximum number of leases a single worker should accept. Setting it too low may cause data loss if workers can't process all shards, and lead to a suboptimal lease assignment among workers. Consider total shard count, number of workers, and worker processing capacity when configuring it.	Unlimited
maxLeaseRenewalThreads	LeaseManagementConfig	Controls the size of the lease renewer thread pool. The more leases that your application could take, the larger this pool should be.	20

Configuration property	Configuration class	Description	Default value
`billingMode`	LeaseManagementConfig	Determines the capacity mode of the lease table created in DynamoDB. There are two options: on-demand mode (PAY_PER_REQUEST) and provisioned mode. We recommend using the default setting of on-demand mode because it automatically scales to accommodate your workload without the need for capacity planning.	PAY_PER_REQUEST (on-demand mode)
`initialLeaseTableReadCapacity`	LeaseManagementConfig	The DynamoDB read capacity that is used if the Kinesis Client Library needs to create a new DynamoDB lease table with provisioned capacity mode. You can ignore this configuration if you are using the default on-demand capacity mode in `billingMode` configuration.	10

Configuration property	Configuration class	Description	Default value
`initialLeaseTableWriteCapacity`	LeaseManagementConfig	The DynamoDB read capacity that is used if the Kinesis Client Library needs to create a new DynamoDB lease table. You can ignore this configuration if you are using the default on-demand capacity mode in `billingMode` configuration.	10
`initialPositionInStreamExtended`	LeaseManagementConfig	The initial position in the stream that the application should start at. This is only used during initial lease creation.	InitialPositionInStream.TRIM_HORIZON
`reBalanceThresholdPercentage`	LeaseManagementConfig	A percentage value that determines when the load balancing algorithm should consider reassigning shards among workers. This is a new configuration introduced in KCL 3.x.	10

Configuration property	Configuration class	Description	Default value
`useAbsoluteReBalanceThreshold`	LeaseManagementConfig	Determines whether `reBalanceThresholdPercentage` is applied as fixed percentage points (`true`) or as a relative percentage of the fleet average (`false`), when the load balancing algorithm uses CPU metrics to rebalance leases. This is a new configuration introduced in KCL 3.x.	FALSE
`dampeningPercentage`	LeaseManagementConfig	A percentage value that is used to dampen the amount of load that will be moved from the overloaded worker in a single rebalance operation. This is a new configuration introduced in KCL 3.x.	60

Configuration property	Configuration class	Description	Default value
`allowThroughputOvershoot`	LeaseManagementConfig	Determines whether additional lease still needs to be taken from the overloaded worker even if it causes total amount of lease throughput taken to exceed the desired throughput amount. This is a new configuration introduced in KCL 3.x.	TRUE
`disableWorkerMetrics`	LeaseManagementConfig	Determines if KCL should ignore resource metrics from workers (such as CPU utilization) when reassigning leases and load balancing. Set this to TRUE if you want to prevent KCL from load balancing based on CPU utilization. This is a new configuration introduced in KCL 3.x.	FALSE

Configuration property	Configuration class	Description	Default value
`maxThroughputPerHostKBps`	LeaseManagementConfig	Amount of the maximum throughput to assign to a worker during the lease assignment. This is a new configuration introduced in KCL 3.x.	Unlimited

Configuration property	Configuration class	Description	Default value
`isGracefulLeaseHandoffEnabled`	LeaseManagementConfig	Controls the behavior of lease handoff between workers. When set to true, KCL will attempt to gracefully transfer leases by allowing the shard's RecordProcessor sufficient time to complete processing before handing off the lease to another worker. This can help ensure data integrity and smooth transitions but may increase handoff time. When set to false, the lease will be handed off immediately without waiting for the RecordProcessor to shut down gracefully. This can lead to faster handoffs but may risk incomplete processing. Note: Checkpointing must be implemented inside the shutdownRequested(	TRUE

Configuration property	Configuration class	Description	Default value
		) method of the RecordProcessor to get benefited from the graceful lease handoff feature. This is a new configuration introduced in KCL 3.x.	

Configuration property	Configuration class	Description	Default value
`gracefulLeaseHandoffTimeoutMillis`	LeaseManagementConfig	Specifies the minimum time (in milliseconds) to wait for the current shard's RecordProcessor to gracefully shut down before forcefully transferring the lease to the next owner. If your processRecords method typically runs longer than the default value, consider increasing this setting. This ensures the RecordProcessor has sufficient time to complete its processing before the lease transfer occurs. This is a new configuration introduced in KCL 3.x.	30,000 (30 seconds)
`maxRecords`	PollingConfig	Allows setting the maximum number of records that Kinesis returns.	10,000

Configuration property	Configuration class	Description	Default value
`retryGetRecordsInSeconds`	PollingConfig	Configures the delay between GetRecords attempts for failures.	None
`maxGetRecordsThreadPool`	PollingConfig	The thread pool size used for GetRecords.	None
`idleTimeBetweenReadsInMillis`	PollingConfig	Determines how long KCL waits between GetRecords calls to poll the data from data streams. The unit is milliseconds.	1,500
`callProcessRecordsEvenForEmptyRecordList`	ProcessorConfig	When set, the record processor is called even when no records were provided from Kinesis.	FALSE
`parentShardPollIntervalMillis`	CoordinatorConfig	How often a record processor should poll to see if the parent shard has been completed. The unit is milliseconds.	10,000 (10 seconds)
`skipShardSyncAtWorkerInitializationIfLeaseExist`	CoordinatorConfig	Disable synchronizing shard data if the lease table contains existing leases.	FALSE
`shardPrioritization`	CoordinatorConfig	Which shard prioritization to use.	NoOpShardPrioritization

Configuration property	Configuration class	Description	Default value
`ClientVersionConfig`	CoordinatorConfig	Determines which KCL version compatibility mode the application will run in. This configuration is only for the migration from previous KCL versions. When migrating to 3.x, you need to set this configuration to `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`. You can remove this configuration when you complete the migration.	CLIENT_VERSION_CONFIG_3X
`taskBackoffTimeMillis`	LifecycleConfig	The time to wait to retry failed KCL tasks. The unit is milliseconds.	500 (0.5 seconds)
`logWarningForTaskAfterMillis`	LifecycleConfig	How long to wait before a warning is logged if a task hasn't completed.	None

Configuration property	Configuration class	Description	Default value
`listShardsBackoffTimeInMillis`	RetrievalConfig	The number of milliseconds to wait between calls to `ListShards` when failures occur. The unit is milliseconds.	1,500 (1.5 seconds)
`maxListShardsRetryAttempts`	RetrievalConfig	The maximum number of times that `ListShards` retries before giving up.	50
`metricsBufferTimeMillis`	MetricsConfig	Specifies the maximum duration (in milliseconds) to buffer metrics before publishing them to CloudWatch.	10,000 (10 seconds)
`metricsMaxQueueSize`	MetricsConfig	Specifies the maximum number of metrics to buffer before publishing to CloudWatch.	10,000
`metricsLevel`	MetricsConfig	Specifies the granularity level of CloudWatch metrics to be enabled and published. Possible values: NONE, SUMMARY, DETAILED.	MetricsLevel.DETAILED

Configuration property	Configuration class	Description	Default value
`metricsEnabledDimensions`	MetricsConfig	Controls allowed dimensions for CloudWatch Metrics.	All dimensions

Discontinued configurations in KCL 3.x

The following configuration properties are discontinued in KCL 3.x:

The table shows discontinued configuration properties for KCL 3.x

Configuration property	Configuration class	Description
`maxLeasesToStealAtOneTime`	LeaseManagementConfig	The maximum number of leases an application should attempt to steal at one time. KCL 3.x will ignore this configuration and reassign leases based on the resource utilization of workers.
`enablePriorityLeaseAssignment`	LeaseManagementConfig	Controls whether workers should prioritize taking very expired leases (leases not renewed for 3x the failover time) and new shard leases, regardless of target lease counts but still respecting max lease limits. KCL 3.x will ignore this configuration and always spread expired leases across workers.

> **Important**
>
> You still must have the discontiuned configuration properties during the migration from previous KCL verisons to KCL 3.x. During the migration, the KCL worker will first start with the KCL 2.x compatible mode and switch to the KCL 3.x functionality mode when it detects that all KCL workers of the application are ready to run KCL 3.x. These discontinued configurations are needed while KCL workers are running the KCL 2.x compatible mode.

KCL version lifecycle policy

This topic outlines the version lifecycle policy for Amazon Kinesis Client Library (KCL). AWS regularly provides new releases for KCL versions to support new features and enhancements, bug fixes, security patches, and dependency updates. We recommend that you stay up-to-date with KCL versions to keep up with the latest features, security updates, and underlying dependencies. We **don't** recommend continued use of an unsupported KCL version.

The lifecycle for major KCL versions consists of the following three phases:

- **General availability (GA)** – During this phase, the major version is fully supported. AWS provides regular minor and patch version releases that include support for new features or API updates for Kinesis Data Streams, as well as bug and security fixes.
- **Maintenance mode** – AWS limits patch version releases to address critical bug fixes and security issues only. The major version won't receive updates for new features or APIs of Kinesis Data Streams.
- **End-of-support** – The major version will no longer receive updates or releases. Previously published releases will continue to be available through public package managers and the code will remain on GitHub. Use of a version which has reached end-of-support is done at the user's discretion. We recommend that you upgrade to the latest major version.

Major version	Current phase	Release date	Maintenance mode date	End-of-support date
KCL 1.x	Maintenance mode	2013-12-19	2025-04-17	2026-01-30

Major version	Current phase	Release date	Maintenance mode date	End-of-support date
KCL 2.x	General availability	2018-08-02	--	--
KCL 3.x	General availability	2024-11-06	--	--

Single table format for KCL

Starting with KCL 3.5, you can consolidate all DynamoDB metadata into a single lease table using the **single table format**. By default, KCL 3.x creates three DynamoDB tables for each application: the lease table, the worker metrics table, and the coordinator state table. The single table format reduces these three tables to one, which helps you avoid DynamoDB account-level table limits.

How single table format works

In single table format, KCL stores worker metrics and coordinator state entries in the lease table alongside lease entries. Each item includes an `entityType` attribute that distinguishes between the different record types.

Worker metrics and coordinator state items use the same primary key structure as the lease table but include a distinct `entityType` value. This attribute allows KCL to identify each item's purpose during table scans.

Each component of KCL filters out the entries it requires for its business logic based on the `entityType` attribute. For example, the Lease Assignment Manager (LAM) filters for lease and worker metrics entries to perform lease assignment.

Configure single table format

How you enable single table format depends on your current KCL version:

- **If you are on KCL 2.x:** Follow the updated migration guide to upgrade to KCL 3.5. Single table format is used by default for new migrations from 2.x to 3.5.
- **If you are on KCL 3.0–3.4:** You must perform a two-phase deployment to migrate to single table format. See the following configuration and migration steps.

For existing KCL 3.x customers, set the `migrateAllEntitiesToLeaseTable` configuration option in `CoordinatorConfig`. This option controls whether KCL stores all metadata entity types in the lease table.

Configuration values for `migrateAllEntitiesToLeaseTable`

Value	Default	Effect
`false`	Yes	KCL uses separate tables for worker metrics and coordinat or state. The application code supports single table format but does not activate it.
`true`	No	KCL begins writing worker metrics and coordinator state data into the lease table. Set this value on the Phase 2 deployment after the `TableMigrationStatus` reaches `DEPLOYED` and you have baked for no regressio ns.

Migrating from KCL 3.x to single table format requires a two-phase deployment:

1. **Phase 1:** Deploy the updated KCL 3.5 code with `migrateAllEntitiesToLeaseTable` set to `false` (the default). This installs the new code that supports single table format but does not activate the migration.
2. **Phase 2:** After all workers run the new code and the `TableMigrationStatus` reaches `DEPLOYED`, and you have verified that there are no regressions, deploy again with `migrateAllEntitiesToLeaseTable` set to `true` to begin the migration.

Migration states

The `TableMigrationStateMachine` manages the transition from multi-table format to single table format. KCL tracks the current migration state in a separate coordinator state entry called

`TableMigration3.5` in the coordinator state table. For the complete list of states, transitions, and descriptions, see KCL single table migration states.

Single table format migration states

State	Description	Transition condition
`INIT`	Initial state. All workers are emitting the minimum support code in worker metric stats. Workers continue to emit worker metrics into the legacy table and read worker metrics and coordinator state from both the legacy and lease tables. Functionally, there is no difference between INIT and DEPLOYED.	All workers emit the minimum support code steadily for the bake time. The application is ready to move to Phase 2 deployment.
`DEPLOYED`	All workers have been deployed with the new code (Phase 1 complete). The application supports single table format but has not activated it.	Phase 2 deployment begins with `migrateAllEntities ToLeaseTable` set to `true`.
`PENDING`	All workers run the new code. KCL migrates data from the worker metrics and coordinator state tables into the lease table.	A 24-hour default bake time elapses after migration completes .
`COMPLETE`	Migration is complete. KCL uses the lease table exclusively for all reads and writes. The old worker metrics and coordinator state tables are no longer used.	Terminal state. No further transitions occur.

For detailed information about each state, transition conditions, and the complete state machine behavior, see KCL single table migration state machine.

> **Note**
>
> The default bake time between the PENDING and COMPLETE states is 24 hours, but you can configure it up to one week. During this period, KCL uses the lease table for all reads and writes but does not delete the old tables. You must manually delete the old worker metrics and coordinator state tables after you confirm the migration is successful. KCL does not delete these tables automatically.

Migration metrics

With KCL, you can monitor the progress and health of the single table migration using CloudWatch metrics. Use these metrics to confirm that workers have adopted the new code and to track the migration as it moves through its states. You can also detect DynamoDB read, write, or delete faults during the migration. The following tables group the metrics by the KCL operation (metric dimension) that emits them and by when each metric is emitted.

The elected leader worker continuously emits the following metrics, regardless of whether a migration is in progress:

Metrics emitted by the leader at all times

Operation	Metric	Unit	Description
`TableMigration`	`StatusOrdinal`	None	Ordinal of the current DynamoDB migration status: `0` = UNKNOWN, `1` = INIT, `2` = DEPLOYED, `3` = PENDING, `4` = COMPLETE.
`WorkerMetrics`	`FleetMinSupportCode`	None	Minimum support code across all lease-owning workers in the fleet.

The elected leader worker emits the following metrics only while a migration is in progress:

Metrics emitted by the leader during migration

Operation	Metric	Unit	Description
TableMigration	Phase1Worker	Count	Number of workers that support operating with a single DynamoDB table but have not yet migrated to using the single table.
TableMigration	Phase2Worker	Count	Number of workers that have migrated to writing to the single table. These workers might still read from multiple tables until the table migration completes.
TableMigration	PrePhase1Worker	Count	Number of workers on a version prior to 3.5 that cannot support operating on a single DynamoDB table.
TableMigration	WriteFault	Count	1 on a DynamoDB write failure, 0 on success.
TableMigration	DeleteFault	Count	1 on a DynamoDB delete failure, 0 on success.
TableMigration	CompletionFault	Count	1 when the transactional write of the COMPLETE status fails, 0 on success.
TableMigrationAsyncMove	Success	Count	1 on a successful transactional move of CoordinatorState entries, 0 on failure.
TableMigrationAsyncMove	Time	Milliseconds	Duration of the async move operation.
TableMigrationAsyncMove	BatchCount	Count	Number of batches that were successfully moved.

All workers emit the following metrics while a migration is in progress:

Metrics emitted by all workers during migration

Operation	Metric	Unit	Description
TableMigration	ReadFault	Count	1 on failure to read the TableMigrationState from DynamoDB, 0 on success.
TableMigration	Time	Milliseconds	Duration of the state machine run.
TableMigrationInitialize	Success	Count	1 on a successful initialize, 0 on failure.
TableMigrationInitialize	Time	Milliseconds	Duration of the initialize operation.

Use these metrics to decide when to advance the migration. When StatusOrdinal is consistently 2 (DEPLOYED) and PrePhase1Worker is 0, all workers support the single table format, and you can move to phase 2 of the table migration deployment. After StatusOrdinal reaches 4 (COMPLETE), you can safely delete the legacy tables.

Rollback considerations

Rollback support depends on the current TableMigrationStatus:

- **During Phase 1** (TableMigrationStatus is DEPLOYED or not yet set): You can safely roll back to the previous version. The new code runs in a backward-compatible mode and no data has been written to non-lease entries in the lease table.
- **During Phase 2** (TableMigrationStatus is DEPLOYED or PENDING): You can roll back to Phase 1. Workers revert to using the legacy tables (multi-table format) for worker metrics and coordinator state. The migration is undone.

- **After COMPLETE state**: Rollback is not supported. KCL only uses the lease table for all entities. Even if the code rolls back to Phase 1, the worker continues using the lease table for all entities and the configuration is ignored.

> **⚠ Warning**
>
> After the migration reaches the COMPLETE state, the application operates exclusively in single table mode. The `migrateAllEntitiesToLeaseTable` configuration is ignored, and KCL does not revert to using separate tables. Make sure the bake time before moving to COMPLETE is sufficient, because after that even a code rollback does not switch back to multiple tables.

Best practices

Follow these best practices when you adopt single table format:

- After Phase 1 deployment, verify that the coordinator state entry `TableMigration3.5` reaches `DEPLOYED` status. Ensure there are no regressions before proceeding to Phase 2.
- Monitor the `TableMigrationStatus` in the `TableMigration3.5` coordinator state entry to track progress through the DEPLOYED, PENDING, and COMPLETE states. The status is stored in the coordinator state table as a separate entry (not in the lease table) until the migration reaches COMPLETE.
- Make sure the bake time before moving to COMPLETE is sufficient, because after that even a code rollback does not switch back to multiple tables. The application functions only in single table mode.
- After the migration reaches COMPLETE, manually delete the old worker metrics and coordinator state tables. KCL does not delete these tables automatically—it only stops using them.
- If you have configured `CoordinatorConfig.coordinatorStateTableConfig` or `LeaseManagementConfig.workerUtilizationAwareAssignmentConfig.workerMetricsTable` you can remove these configurations after migration completes. These configurations are deprecated in KCL 3.5 and later.

Migrate from previous KCL versions

This topic explains how to migrate from previous versions of the Kinesis Client Library (KCL).

What's new in KCL 3.0?

Kinesis Client Library (KCL) 3.0 introduces several major enhancements compared to previous versions:

- It lowers compute costs for consumer applications by automatically redistributing the work from over-utilized workers to under-utilized workers in the consumer application fleet. This new load balancing algorithm ensures the evenly distributed CPU utilization across workers and removes the need to over-provision workers.
- It reduces the DynamoDB cost associated with KCL by optimizing read operations on the lease table.
- It minimizes reprocessing of data when leases are reassigned to another worker by allowing the current worker to complete checkpointing the records that it has processed.
- It uses AWS SDK for Java 2.x for improved performance and security features, fully removing the dependency on AWS SDK for Java 1.x.

For more information, see KCL 3.0 release note.

Topics

- Migrate from KCL 2.x to KCL 3.x
- Roll back to the previous KCL version
- Roll forward to KCL 3.5.x after a rollback from KCL 3.5.x
- Best practices for the lease table with provisioned capacity mode
- Migrate from KCL 3.x to single table format
- Migrating from KCL 1.x to KCL 3.x

Migrate from KCL 2.x to KCL 3.x

This topic provides step-by-step instructions to migrate your consumer from KCL 2.x to KCL 3.x. We recommend migrating to KCL 3.5 or later to use the single table format. KCL 3.x supports in-place migration of KCL 2.x consumers. You can continue consuming the data from your Kinesis data stream while migrating your workers in a rolling manner.

Important

KCL 3.x maintains the same interfaces and methods as KCL 2.x. Therefore you don't have to update your record processing code during the migration. However, you must set the proper configuration and check the required steps for the migration. We highly recommend that you follow the following migration steps for a smooth migration experience.

Important

For new migrations from KCL 2.x to KCL 3.5 or later, single table format is used by default. Your application uses only the lease table for all metadata, eliminating the need for separate worker metrics and coordinator state tables. For more information, see Single table format for KCL.

Step 1: Prerequisites

Before you start using KCL 3.x, make sure that you have the following:

- Java Development Kit (JDK) 8 or later
- AWS SDK for Java 2.x
- Maven or Gradle for dependency management

Important

Do not use AWS SDK for Java version 2.27.19 to 2.27.23 with KCL 3.x. These versions include an issue that causes an exception error related to KCL's DynamoDB usage. We recommend that you use the AWS SDK for Java version 2.28.0 or later to avoid this issue.

Step 2: Add dependencies

If you're using Maven, add the following dependency to your `pom.xml` file. Make sure you replaced 3.x.x to the latest KCL version.

```
<dependency>
```

```
    <groupId>software.amazon.kinesis</groupId>
    <artifactId>amazon-kinesis-client</artifactId>
    <version>3.x.x</version> <!-- Use the latest version -->
</dependency>
```

If you're using Gradle, add the following to your `build.gradle` file. Make sure you replaced 3.x.x to the latest KCL version.

```
implementation 'software.amazon.kinesis:amazon-kinesis-client:3.x.x'
```

You can check for the latest version of the KCL on the Maven Central Repository.

Step 3: Set up the migration-related configuration

To migrate from KCL 2.x to KCL 3.x, you must set the following configuration parameter:

- CoordinatorConfig.clientVersionConfig: This configuration determines which KCL version compatibility mode the application will run in. When migrating from KCL 2.x to 3.x, the migration is performed in phases. First, set this configuration to `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1` and deploy to all workers. Add the following line when creating your scheduler object:

```
configsBuilder.coordinatorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONFI
```

In this phase, your application remains compatible with KCL 2.x and the migration does not start, which lets you safely roll out the KCL 3.x library across your fleet.

After all workers are running with `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`, set this configuration to `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X` to start the migration. To set this configuration, add the following line when creating your scheduler object:

```
configsBuilder.coordinatorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONFI
```

The following is an example of how to set the `CoordinatorConfig.clientVersionConfig` for migrating from KCL 2.x to 3.x. You can adjust other configurations as needed based on your specific requirements:

```
Scheduler scheduler = new Scheduler(
```

```
    configsBuilder.checkpointConfig(),

 configsBuilder.coordinatorConfig().clientVersionConfig(ClientVersionConfig.CLIENT_VERSION_CONF
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
);
```

It's important that all workers in your consumer application use the same load balancing algorithm at a given time because KCL 2.x and 3.x use different load balancing algorithms. Running workers with different load balancing algorithms can cause suboptimal load distribution as the two algorithms operate independently.

This KCL 2.x compatibility setting allows your KCL 3.x application to run in a mode compatible with KCL 2.x and use the load balancing algorithm for KCL 2.x until all workers in your consumer application have been upgraded to KCL 3.x. When the migration is complete, KCL will automatically switch to full KCL 3.x functionality mode and start using a new KCL 3.x load balancing algorithm for all running workers.

Important

If you are not using `ConfigsBuilder` but creating a `LeaseManagementConfig` object to set configurations, you must add one more parameter called `applicationName` in KCL version 3.x or later. For details, see Compilation error with the LeaseManagementConfig constructor. We recommend using `ConfigsBuilder` to set KCL configurations. `ConfigsBuilder` provides a more flexible and maintainable way to configure your KCL application.

Note

For the latest migration-related configuration changes for KCL 3.5, see KCL 3.5 configuration updates.

Step 4: Follow best practices for the shutdownRequested() method implementation

KCL 3.x introduces a feature called *graceful lease handoff* to minimize the reprocessing of data when a lease is handed over to another worker as part of the lease reassignment process. This is achieved by checkpointing the last processed sequence number in the lease table before the lease handoff. To ensure the graceful lease handoff works properly, you must make sure that you invoke the `checkpointer` object within the `shutdownRequested` method in your `RecordProcessor` class. If you're not invoking the `checkpointer` object within the `shutdownRequested` method, you can implement it as illustrated in the following example.

> **⚠ Important**
>
> - The following implementation example is a minimal requirement for the graceful lease handoff. You can extend it to include additional logic related to the checkpointing if needed. If you are performing any asynchronous processing, make sure that all delivered records to the downstream were processed before invoking checkpointing.
> - While graceful lease handoff significantly reduces the likelihood of data reprocessing during lease transfers, it does not entirely eliminate this possibility. To preserve data integrity and consistency, design your downstream consumer applications to be idempotent. This means they should be able to handle potential duplicate record processing without adverse effects on the overall system.

```
/**
 * Invoked when either Scheduler has been requested to gracefully shutdown
 * or lease ownership is being transferred gracefully so the current owner
 * gets one last chance to checkpoint.
 *
 * Checkpoints and logs the data a final time.
 *
 * @param shutdownRequestedInput Provides access to a checkpointer, allowing a record
 processor to checkpoint
 *                               before the shutdown is completed.
 */
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
    try {
       // Ensure that all delivered records are processed
       // and has been successfully flushed to the downstream before calling
       // checkpoint
```

```
        // If you are performing any asynchronous processing or flushing to
        // downstream, you must wait for its completion before invoking
        // the below checkpoint method.
         log.info("Scheduler is shutting down, checkpointing.");
         shutdownRequestedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
         log.error("Exception while checkpointing at requested shutdown. Giving up.",
 e);
    }
}
```

Step 5: Check the KCL 3.x prerequisites for collecting worker metrics

KCL 3.x collects CPU utilization metrics such as CPU utilization from workers to balance the load across workers evenly. Consumer application workers can run on Amazon EC2, Amazon ECS, Amazon EKS, or AWS Fargate. KCL 3.x can collect CPU utilization metrics from workers only when the following prerequisites are met:

Amazon Elastic Compute Cloud(Amazon EC2)

- Your operating system must be Linux OS.
- You must enable IMDSv2 in your EC2 instance.

Amazon Elastic Container Service (Amazon ECS) on Amazon EC2

- Your operating system must be Linux OS.
- You must enable ECS task metadata endpoint version 4.
- Your Amazon ECS container agent version must be 1.39.0 or later.

Amazon ECS on AWS Fargate

- You must enable Fargate task metadata endpoint version 4. If you use Fargate platform version 1.4.0 or later, this is enabled by default.
- Fargate platform version 1.4.0 or later.

Amazon Elastic Kubernetes Service (Amazon EKS) on Amazon EC2

- Your operating system must be Linux OS.

Amazon EKS on AWS Fargate

- Fargate platform 1.3.0 or later.

> **Important**
>
> If KCL 3.x cannot collect CPU utilization metrics from workers because prerequisites are not met, it will rebalance the load the throughput level per lease. This fallback rebalancing mechanism will make sure all workers will get similar total throughput levels from leases assigned to each worker. For more information, see How KCL assigns leases to workers and balances the load.

Step 6: Update IAM permissions for KCL 3.x

You must add the following permissions to the IAM role or policy associated with your KCL 3.x consumer application. This involves updating the existing IAM policy used by the KCL application. For more information, see IAM permissions required for KCL consumer applications.

> **Important**
>
> Your existing KCL applications might not have the following IAM actions and resources added in the IAM policy because they were not needed in KCL 2.x. Make sure that you have added them before running your KCL 3.x application:
>
> - Actions: `UpdateTable`
> - Resources (ARNs): `arn:aws:dynamodb:region:account:table/KCLApplicationName`
> - Actions: `Query`
> - Resources (ARNs): `arn:aws:dynamodb:region:account:table/KCLApplicationName/index/*`
> - Actions: `CreateTable`, `DescribeTable`, `Scan`, `GetItem`, `PutItem`, `UpdateItem`, `DeleteItem`
> - Resources (ARNs): `arn:aws:dynamodb:region:account:table/KCLApplicationName-WorkerMetricStats`, `arn:aws:dynamodb:region:account:table/KCLApplicationName-CoordinatorState`

Replace "region," "account," and "KCLApplicationName" in the ARNs with your own AWS Region, AWS account number, and KCL application name respectively. If you use configurations to customize names of the metadata tables created by KCL, use those specified table names instead of KCL application name.

Step 7: Deploy KCL 3.x code to your workers

After you have set the configuration required for the migration and completed the all previous migration checklists, you can build and deploy your code to your workers.

Note

If you see a compilation error with the `LeaseManagementConfig` constructor, see Compilation error with the LeaseManagementConfig constructor for troubleshooting information.

Step 8: Complete the migration

During the deployment of KCL 3.x code, KCL continues using the lease assignment algorithm from KCL 2.x. When you have successfully deployed KCL 3.x code to all of your workers, KCL automatically detects this and switches to the new lease assignment algorithm based on resource utilization of the workers. For more details about the new lease assignment algorithm, see How KCL assigns leases to workers and balances the load.

During the deployment, you can monitor the migration process with the following metrics emitted to CloudWatch. You can monitor metrics under the `Migration` operation. All metrics are per-KCL-application metrics and set to the `SUMMARY` metric level. If the `Sum` statistic of the `CurrentState:3xWorker` metric matches the total number of workers in your KCL application, it indicates that the migration to KCL 3.x has successfully completed.

Important

It takes at least 10 minutes for KCL to switch to the new leasee assignment algorithm after all workers are ready to run it.

CloudWatch metrics for the KCL migration process

Metrics	Description
`CurrentState:3xWorker`	The number of KCL workers successfully migrated to KCL 3.x and running the new lease assignment algorithm. If the `Sum` count of this metric matches the total number of your workers, it indicates that the migration to KCL 3.x has successfully completed. • Metric level: Summary • Units: Count • Statistics: The most useful statistic is `Sum`
`CurrentState:2xCompatibleWorker`	The number of KCL workers running in KCL 2.x compatible mode during the migration process. A non-zero value for this metric indicates that the migration is still in progress. • Metric level: Summary • Units: Count • Statistics: The most useful statistic is `Sum`
`Fault`	The number of exceptions encountered during the migration process. Most of these exceptions are transient errors, and KCL 3.x will automatically retry to complete the migration. If you observe a persistent `Fault` metric value, review your logs from the migration period for further troubleshooting. If the issue continues, contact Support. • Metric level: Summary • Units: Count • Statistics: The most useful statistic is `Sum`

Metrics	Description
GsiStatusReady	The status of the global secondary index (GSI) creation on the lease table. This metric indicates whether the GSI on the lease table has been created, a prerequisite to run KCL 3.x. The value is 0 or 1, with 1 indicating successful creation. During a rollback state, this metric will not be emitted. After you roll forward again, you can resume monitoring this metric. • Metric level: Summary • Units: Count • Statistics: The most useful statistic is Sum
workerMetricsReady	Status of worker metrics emission from all workers. The metrics indicates whether all workers are emitting metrics like CPU utilization. The value is 0 or 1, with 1 indicating all workers are successfully emitting metrics and ready for the new lease assignment algorithm. During a rollback state, this metric will not be emitted. After you roll forward again, you can resume monitoring this metric. • Metric level: Summary • Units: Count • Statistics: The most useful statistic is Sum

KCL provides rollback capability to the 2.x compatible mode during migration.
After successful migration to KCL 3.x is successful, we recommend that you
remove the `CoordinatorConfig.clientVersionConfig` setting of
`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X` if rollback is no longer needed. Removing this configuration stops the emission of migration-related metrics from the KCL application.

Note

We recommend that you monitor your application's performance and stability for a period during the migration and after completing the migration. If you observe any issues, you can rollback workers to use KCL 2.x compatible functionality using the KCL Migration Tool.

Roll back to the previous KCL version

This topic explains the steps to roll back your KCL 3.5.x consumer to the previous version. The rollback process depends on which migration phase your application is currently in.

Important

The KCL Migration Tool is required when rolling back from Phase 2 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`) to Phase 1 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`). The tool is also required to roll back from Phase 1 to your previous KCL version (KCL 2.x). This applies when your lease table still contains migration-specific entries from an earlier Phase 2 deployment. If your application entered Phase 1 directly and has not started Phase 2, you can roll back to your previous KCL version by redeploying your previous code without running the tool.

Roll back from Phase 1 to the previous KCL version

If your application entered Phase 1 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) directly as part of a migration from KCL 2.x and has not yet started Phase 2, you can roll back to your previous KCL version by redeploying your previous code. Phase 1 is backward compatible with previous KCL versions and does not create any migration-specific entries in the lease table. No migration tool is needed.

To roll back from Phase 1:

1. Redeploy the code with your previous KCL version to all workers.

Note

You cannot roll back directly from Phase 2 to your previous KCL version (KCL 2.x). You must roll back in sequence:

1. Roll back from Phase 2 to Phase 1 (run the tool, then redeploy the Phase 1 code). See Roll back from Phase 2 to Phase 1.
2. Roll back from Phase 1 to KCL 2.x (run the tool, then redeploy your previous code). See Roll back from Phase 1 to KCL 2.x.

You might redeploy the Phase 1 code for either of two reasons: to continue rolling back to KCL 2.x, or because running the tool alone did not mitigate the issue and you want to fully return to Phase 1.

Roll back from Phase 2 to Phase 1

If your application is in Phase 2 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`), you must use the KCL Migration Tool to roll back to Phase 1. This is a two-step process:

1. Run the KCL Migration Tool.
2. Redeploy the code with Phase 1 configuration (optional).

Note

The KCL Migration Tool handles only the Phase 2 to Phase 1 rollback. To reach your previous KCL version (KCL 2.x) from Phase 2, first roll back to Phase 1 using the steps in this section, and then follow Roll back from Phase 1 to KCL 2.x to run the tool in `rollback-to-v2` mode and delete the non-lease entries before you redeploy your previous code.

Step 1: Run the KCL Migration Tool

When you need to roll back from Phase 2 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`) to Phase 1, run the KCL Migration Tool. The tool performs the following tasks:

- It removes the Global Secondary Index (LeaseOwnerToLeaseKeyIndex) on the lease table in DynamoDB. This index is created by KCL 3.5.x but is not needed when you roll back to Phase 1.

- It makes all workers run in a mode compatible with KCL 2.x and start using the load balancing algorithm used in previous KCL versions. If you have issues with the new load balancing algorithm in KCL 3.5.x, this mitigates the issue immediately.

> **Important**
>
> The coordinator state entry (`Migration3.0`) in the lease table must not be deleted during the migration, rollback, and rollforward process.

> **Note**
>
> All workers in your consumer application must use the same load balancing algorithm at a given time. The KCL Migration Tool makes sure that all workers in your KCL 3.5.x consumer application switch to the KCL 2.x compatible mode so that all workers run the same load balancing algorithm during the rolling deployment back to Phase 1.

You can download the KCL Migration Tool in the scripts directory of the KCL GitHub repository. Run the script from any of your workers or any host that has the required permissions to write to and update the lease table. You can refer to IAM permissions required for KCL consumer applications for required IAM permissions to run the script. You must run the script only once per KCL application. Run the KCL Migration Tool with the following command:

```
python3 ./KclMigrationTool.py --region <region> --mode rollback-to-phase1 [--
application_name <applicationName>] [--lease_table_name <leaseTableName>]
```

Parameters

- --region: Replace `<region>` with your AWS Region.
- --application_name: This parameter is required if you're using the default name for your lease table. If you have specified a custom name for the lease table, you can omit this parameter. Replace `<applicationName>` with your actual KCL application name. The tool uses this name to derive the default table name if a custom name is not provided.
- --lease_table_name (optional): This parameter is needed when you have set a custom name for the lease table in your KCL configuration. If you're using the default table name, you can omit

this parameter. Replace `<leaseTableName>` with the custom table name you specified for your lease table.

Step 2: Redeploy the code with Phase 1 configuration (optional)

After running the KCL Migration Tool for a rollback from Phase 2 to Phase 1, you'll see one of these messages:

> **Note**
>
> Redeploying the Phase 1 configuration is optional if you rolled back to mitigate an issue and intend to stay on KCL 3.5.x (you can later roll forward to Phase 2 without redeploying). It is required if you intend to continue rolling back to KCL 2.x, because the `rollback-to-v2` step requires all workers to be running the Phase 1 configuration first.

- **Message 1:** "Rollback completed. Your application was running Phase 2 (2x compatible) functionality. Please rollback to Phase 1 by deploying your KCL 3.5.x application with the Phase 1 configuration."
  - **Required action:** Your workers were running in Phase 2 (2x compatible) mode. Redeploy your KCL 3.5.x application with Phase 1 configuration (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) to your workers.
- **Message 2:** "Rollback completed. Your KCL application was running Phase 2 (3x) functionality and has been rolled back to Phase 2 (2x compatible) mode. If you don't see mitigation after a short period of time, please rollback to Phase 1 by deploying your KCL 3.5.x application with the Phase 1 configuration."
  - **Required action:** Your workers were running in Phase 2 (3x) mode and the KCL Migration Tool rolled them back to Phase 2 (2x compatible) mode. If the issue is resolved, you don't need to redeploy. If the issue persists, redeploy your KCL 3.5.x application with Phase 1 configuration (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) to your workers.

Roll back from Phase 1 to KCL 2.x

This section describes how to roll back from Phase 1 to your previous KCL version (KCL 2.x) when your application has been in Phase 2. This is the second step of the sequence described in Roll back from Phase 1 to the previous KCL version: first roll back from Phase 2 to Phase 1, and then

complete the steps in this section. After all workers are running Phase 1, run the KCL Migration Tool in `rollback-to-v2` mode to clean up the non-lease entries, and then redeploy your previous code.

> **Important**
>
> Rolling back to your previous code without this step will stall the application, because the previous code cannot handle the non-lease entries in the lease table.

> **Important**
>
> Before you run the tool in `rollback-to-v2` mode, confirm that the Phase 2 to Phase 1 rollback is complete: all workers in your fleet have finished the Phase 1 code rollback deployment with `CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1` and no Phase 2 workers remain. The tool proceeds only when `MigrationState.ClientVersion` is already `CLIENT_VERSION_2X`, which indicates that the Phase 2 to Phase 1 rollback has run. Verify this using your CI/CD system.

To roll back from Phase 1 to your previous KCL version (KCL 2.x):

1. Run the KCL Migration Tool in `rollback-to-v2` mode. You can run the script from any of your workers or any host that has the required permissions to write to and update the lease table. Because this mode deletes the non-lease entries from the lease table, the host must have delete permissions (`DeleteItem`) on the lease table. For the required IAM permissions, see IAM permissions required for KCL consumer applications.

   ```
   python3 ./KclMigrationTool.py --region <region> --mode rollback-to-v2 [--
   application_name <applicationName>] [--lease_table_name <leaseTableName>]
   ```

   Parameters

   - --region: Replace `<region>` with your AWS Region.
   - --application_name: This parameter is required if you're using the default name for your lease table. If you have specified a custom name for the lease table, you can omit this parameter. Replace `<applicationName>` with your actual KCL application name. The tool uses this name to derive the default table name if a custom name is not provided.

- (Optional) --lease_table_name: Use this parameter when you have set a custom name for the lease table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace `<leaseTableName>` with the custom table name you specified for your lease table.

2. The tool scans the lease table for non-lease entities and lists the entries that it will delete from the lease table, such as the client version migration entry (`Migration3.0`) and worker metric statistics entries. Review the listed entries.
3. When the tool prompts you, type the following confirmation string exactly to acknowledge that the Phase 1 code rollback is complete:

```
Code rollback to CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1 completed
```

4. When the tool prompts you to confirm the deletion, type `yes`. The tool permanently deletes the listed non-lease entities from the lease table.
5. After the tool reports that the lease table is clean for the KCL 2.x rollback, redeploy the code with your previous KCL version (KCL 2.x) to all workers.

> **Important**
>
> After the tool deletes the non-lease entities, a later roll forward to Phase 2 requires a fresh migration from Phase 1. When you're ready to migrate to KCL 3.5.x again, deploy with the Phase 1 configuration (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`), allow the Phase 1 deployment to bake for your confidence interval, and then deploy with the Phase 2 configuration (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`). No tool is needed for either of these deployments.

Roll forward to KCL 3.5.x after a rollback from KCL 3.5.x

This topic explains the steps to roll forward your consumer back to KCL 3.5.x after a rollback. When you need to roll forward, you must go through a two-step process:

1. Run the KCL Migration Tool.
2. Deploy the Phase 2 code.

Step 1: Run the KCL Migration Tool

Run the KCL Migration Tool with the following command to roll forward to KCL 3.5.x:

```
python3 ./KclMigrationTool.py --region <region> --mode rollforward [--application_name
 <applicationName>] [--lease_table_name <leaseTableName>]
```

Parameters

- --region: Replace `<region>` with your AWS Region.
- --application_name: This parameter is required if you're using the default name for your lease table. If you have specified a custom name for the lease table, you can omit this parameter. Replace `<applicationName>` with your actual KCL application name. The tool uses this name to derive the default table name if a custom name is not provided.
- --lease_table_name (optional): This parameter is needed when you have set a custom name for the lease table in your KCL configuration. If you're using the default table name, you can omit this parameter. Replace `<leaseTableName>` with the custom table name you specified for your lease table.

After you run the migration tool in roll-forward mode, KCL creates the following DynamoDB resource required for KCL 3.5.x:

- A Global Secondary Index on the lease table

> **Note**
>
> KCL 3.5.x does not create separate worker metrics or coordinator state tables during migration. All metadata is stored in the lease table.

Step 2: Deploy the code with Phase 2 configuration

After running the KCL Migration Tool for a roll forward, deploy your code with KCL 3.5.x Phase 2 configuration (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`) to your workers. Follow Step 8: Complete the migration to complete your migration.

Best practices for the lease table with provisioned capacity mode

If the lease table of your KCL application was switched to provisioned capacity mode, KCL 3.x creates a global secondary index on the lease table with the provisioned billing mode and the same read capacity units (RCU) and write capacity units (WCU) as the base lease table. When the global secondary index is created, we recommend that you monitor the actual usage on the global secondary index in the DynamoDB console and adjust the capacity units if needed. For a more detailed guide about switching the capacity mode of DynamoDB metadata tables created by KCL, see DynamoDB capacity mode for metadata tables created by KCL.

> **Note**
>
> By default, KCL creates metadata tables such as the lease table, worker metrics table, and coordinator state table, and the global secondary index on the lease table using the on-demand capacity mode. We recommend that you use the on-demand capacity mode to automatically adjust the capacity based on your usage changes.

Migrate from KCL 3.x to single table format

If you run KCL 3.x with all three DynamoDB metadata tables (lease table, worker metrics table, and coordinator state table), you can migrate to single table format to consolidate all metadata into the lease table.

Prerequisites

- Your application runs KCL 3.x with all three metadata tables.
- Your application is stable and all workers are healthy.
- Your application has finished the KCL 3.x migration and does not configure `CoordinatorConfig.clientVersionConfig`. Do not perform the table migration and the client version migration from 2.x to 3.x at the same time.

To migrate from KCL 3.x to single table format, follow the two-phase deployment process

1. **Phase 1:** Update your project dependency to KCL 3.5 or later. Deploy with `migrateAllEntitiesToLeaseTable` set to `false` (the default). This deployment installs the new code that supports single table format but does not activate the migration.

2. Wait for the deployment to complete on all workers. Monitor the `TableMigration3.5` coordinator state entry until the `TableMigrationStatus` reaches `DEPLOYED`. Verify that there are no regressions and allow sufficient bake time.
3. **Phase 2:** Deploy again with `migrateAllEntitiesToLeaseTable` set to `true`. This deployment begins the migration to single table format.
4. Monitor the `TableMigrationStatus`. The leader worker advances the state through DEPLOYED, PENDING, and COMPLETE.
5. After the status reaches COMPLETE, verify that there are no regressions and allow additional bake time to make sure no rollbacks are necessary.
6. Manually delete the old worker metrics and coordinator state tables. KCL does not delete these tables automatically—it only stops using them. If you have configured `CoordinatorConfig.coordinatorStateTableConfig` or `LeaseManagementConfig.workerUtilizationAwareAssignmentConfig.workerMetricsTabl` you can remove these configurations as they are deprecated.

> **Important**
>
> Do not roll back your application after the migration reaches the COMPLETE state. After COMPLETE, the application operates exclusively in single table mode. Even a code rollback to Phase 1 does not switch back to multiple tables— the configuration is ignored and the application continues using only the lease table.

For details on migration states and rollback options, see Single table format for KCL.

Migrating from KCL 1.x to KCL 3.x

This topic explains the instructions to migrate your consumer from KCL 1.x to KCL 3.x. KCL 1.x uses different classes and interfaces compared to KCL 2.x and KCL 3.x. You must migrate the record processor, record processor factory, and worker classes to the KCL 2.x/3.x compatible format first, and follow the migration steps for KCL 2.x to KCL 3.x migration. You can directly upgrade from KCL 1.x to KCL 3.x.

> **Important**
>
> We recommend that you migrate to KCL 3.5 or later, which uses single table format by default for new migrations. First, migrate your record processor, factory, and worker

classes to the KCL 2.x/3.x compatible format using the steps on this page. Then, follow the migration path from KCL 2.x to KCL 3.5. Single table format is automatically enabled for new migrations from 2.x. For more information, see Single table format for KCL.

- **Step 1: Migrate the record processor**

  Follow the Migrate the record processor section in the Migrate consumers from KCL 1.x to KCL 2.x page.

- **Step 2: Migrate the record processor factory**

  Follow the Migrate the record processor factory section in the Migrate consumers from KCL 1.x to KCL 2.x page.

- **Step 3: Migrate the worker**

  Follow the Migrate the worker section in the Migrate consumers from KCL 1.x to KCL 2.x page.

- **Step 4: Migrate KCL 1.x configuration**

  Follow the Configure the Amazon Kinesis client section in the Migrate consumers from KCL 1.x to KCL 2.x page.

- **Step 5: Check idle time removal and client configuration removals**

  Follow the Idle time removal and Client configuration removals sections in the Migrate consumers from KCL 1.x to KCL 2.x page.

- **Step 6: Follow the step-by-step instructions in the KCL 2.x to KCL 3.x migration guide**

  Follow instructions on the Migrate from KCL 2.x to KCL 3.x page to complete the migration. If you need to roll back to the previous KCL version or roll forward to KCL 3.x after a rollback, refer to Roll back to the previous KCL version and Roll forward to KCL 3.5.x after a rollback from KCL 3.5.x.

Important

Do not use AWS SDK for Java version 2.27.19 to 2.27.23 with KCL 3.x. These versions include an issue that causes an exception error related to KCL's DynamoDB usage. We recommend that you use the AWS SDK for Java version 2.28.0 or later to avoid this issue.

Previous KCL version documentation

The following topics have been archived. To see current Kinesis Client Library documentation, see Use Kinesis Client Library.

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

Retired documentation

- KCL 1.x and 2.x information
- Develop custom consumers with shared throughput
- Migrate consumers from KCL 1.x to KCL 2.x

KCL 1.x and 2.x information

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

One of the methods of developing custom consumer applications that can process data from KDS data streams is to use the Kinesis Client Library (KCL).

Topics

- About KCL (previous versions)

- KCL previous versions
- KCL concepts (previous versions)
- Use a lease table to track the shards processed by the KCL consumer application
- Process multiple data streams with the same KCL 2.x for Java consumer application
- Use the KCL with the AWS Glue Schema Registry

> **Note**
>
> For both KCL 1.x and KCL 2.x, it is recommended that you upgrade to the latest KCL 1.x version or KCL 2.x version, depending on your usage scenario. Both KCL 1.x and KCL 2.x are regularly updated with newer releases that include the latest dependency and security patches, bug fixes, and backward-compatible new features. For more information, see https://github.com/awslabs/amazon-kinesis-client/releases.

About KCL (previous versions)

KCL helps you consume and process data from a Kinesis data stream by taking care of many of the complex tasks associated with distributed computing. These include load balancing across multiple consumer application instances, responding to consumer application instance failures, checkpointing processed records, and reacting to resharding. The KCL takes care of all of these subtasks so that you can focus your efforts on writing your custom record-processing logic.

The KCL is different from the Kinesis Data Streams APIs that are available in the AWS SDKs. The Kinesis Data Streams APIs help you manage many aspects of Kinesis Data Streams, including creating streams, resharding, and putting and getting records. The KCL provides a layer of abstraction around all these subtasks, specifically so that you can focus on your consumer application's custom data processing logic. For information about the Kinesis Data Streams API, see the Amazon Kinesis API Reference.

> **Important**
>
> The KCL is a Java library. Support for languages other than Java is provided using a multi-language interface called the MultiLangDaemon. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. For example, if you install the KCL for Python and write your consumer application entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon

has some default settings that you might need to customize for your use case, for example, the AWS region that it connects to. For more information about the MultiLangDaemon on GitHub, see KCL MultiLangDaemon project.

The KCL acts as an intermediary between your record processing logic and Kinesis Data Streams.

KCL previous versions

Currently, you can use either of the following supported versions of KCL to build your custom consumer applications:

- **KCL 1.x**

  For more information, see Develop KCL 1.x consumers
- **KCL 2.x**

  For more information, see Develop KCL 2.x Consumers

You can use either KCL 1.x or KCL 2.x to build consumer applications that use shared throughput. For more information, see Develop custom consumers with shared throughput using KCL.

To build consumer applications that use dedicated throughput (enhanced fan-out consumers), you can only use KCL 2.x. For more information, see Develop enhanced fan-out consumers with dedicated throughput.

For information about the differences between KCL 1.x and KCL 2.x, and instructions on how to migrate from KCL 1.x to KCL 2.x, see Migrate consumers from KCL 1.x to KCL 2.x.

KCL concepts (previous versions)

- **KCL consumer application** – an application that is custom-built using KCL and designed to read and process records from data streams.
- **Consumer application instance** - KCL consumer applications are typically distributed, with one or more application instances running simultaneously in order to coordinate on failures and dynamically load balance data record processing.
- **Worker** – a high level class that a KCL consumer application instance uses to start processing data.

> **Important**
>
> Each KCL consumer application instance has one worker.

The worker initializes and oversees various tasks, including syncing shard and lease information, tracking shard assignments, and processing data from the shards. A worker provides KCL with the configuration information for the consumer application, such as the name of the data stream whose data records this KCL consumer application is going to process and the AWS credentials that are needed to access this data stream. The worker also kick starts that specific KCL consumer application instance to deliver data records from the data stream to the record processors.

> **Important**
>
> In KCL 1.x this class is called **Worker**. For more information, (these are the Java KCL repositories), see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/Worker.java. In KCL 2.x, this class is called **Scheduler**. Scheduler's purpose in KCL 2.x is identical to Worker's purpose in KCL 1.x. For more information about the Scheduler class in KCL 2.x, see https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/coordinator/Scheduler.java.

- **Lease** – data that defines the binding between a worker and a shard. Distributed KCL consumer applications use leases to partition data record processing across a fleet of workers. At any given time, each shard of data records is bound to a particular worker by a lease identified by the **leaseKey** variable.

  By default, a worker can hold one or more leases (subject to the value of the **maxLeasesForWorker** variable) at the same time.

  > **Important**
  >
  > Every worker will contend to hold all available leases for all available shards in a data stream. But only one worker will successfully hold each lease at any one time.

For example, if you have a consumer application instance A with worker A that is processing a data stream with 4 shards, worker A can hold leases to shards 1, 2, 3, and 4 at the same time. But if you have two consumer application instances: A and B with worker A and worker B, and these instances are processing a data stream with 4 shards, worker A and worker B cannot both hold the lease to shard 1 at the same time. One worker holds the lease to a particular shard until it is ready to stop processing this shard's data records or until it fails. When one worker stops holding the lease, another worker takes up and holds the lease.

For more information, (these are the Java KCL repositories), see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.java for KCL 1.x and https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/Lease.java for KCL 2.x.

- **Lease table** - a unique Amazon DynamoDB table that is used to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application. The lease table must remain in sync (within a worker and across all workers) with the latest shard information from the data stream while the KCL consumer application is running. For more information, see Use a lease table to track the shards processed by the KCL consumer application.
- **Record processor** – the logic that defines how your KCL consumer application processes the data that it gets from the data streams. At runtime, a KCL consumer application instance instantiates a worker, and this worker instantiates one record processor for every shard to which it holds a lease.

Use a lease table to track the shards processed by the KCL consumer application

Topics

- What is a lease table
- Throughput
- How a lease table is synchronized with the shards in a Kinesis data stream

What is a lease table

For each Amazon Kinesis Data Streams application, KCL uses a unique lease table (stored in a Amazon DynamoDB table) to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application.

> **Important**
>
> KCL uses the name of the consumer application to create the name of the lease table that this consumer application uses, therefore each consumer application name must be unique.

You can view the lease table using the Amazon DynamoDB console while the consumer application is running.

If the lease table for your KCL consumer application does not exist when the application starts up, one of the workers creates the lease table for this application.

> **Important**
>
> Your account is charged for the costs associated with the DynamoDB table, in addition to the costs associated with Kinesis Data Streams itself.

Each row in the lease table represents a shard that is being processed by the workers of your consumer application. If your KCL consumer application processes only one data stream, then `leaseKey` which is the hash key for the lease table is the shard ID. If you are Process multiple data streams with the same KCL 2.x for Java consumer application, then the structure of the leaseKey looks like this: `account-id:StreamName:streamCreationTimestamp:ShardId`. For example, `111111111:multiStreamTest-1:12345:shardId-000000000336`.

In addition to the shard ID, each row also includes the following data:

- **checkpoint:** The most recent checkpoint sequence number for the shard. This value is unique across all shards in the data stream.
- **checkpointSubSequenceNumber:** When using the Kinesis Producer Library's aggregation feature, this is an extension to **checkpoint** that tracks individual user records within the Kinesis record.
- **leaseCounter:** Used for lease versioning so that workers can detect that their lease has been taken by another worker.
- **leaseKey:** A unique identifier for a lease. Each lease is particular to a shard in the data stream and is held by one worker at a time.
- **leaseOwner:** The worker that is holding this lease.

- **ownerSwitchesSinceCheckpoint:** How many times this lease has changed workers since the last time a checkpoint was written.
- **parentShardId:** Used to ensure that the parent shard is fully processed before processing starts on the child shards. This ensures that records are processed in the same order they were put into the stream.
- **hashrange:** Used by the `PeriodicShardSyncManager` to run periodic syncs to find missing shards in the lease table and create leases for them if required.

  > **Note**
  >
  > This data is present in the lease table for every shard starting with KCL 1.14 and KCL 2.3. For more information about `PeriodicShardSyncManager` and periodic synchronization between leases and shards, see How a lease table is synchronized with the shards in a Kinesis data stream.

- **childshards:** Used by the `LeaseCleanupManager` to review the child shard's processing status and decide whether the parent shard can be deleted from the lease table.

  > **Note**
  >
  > This data is present in the lease table for every shard starting with KCL 1.14 and KCL 2.3.

- **shardID:** The ID of the shard.

  > **Note**
  >
  > This data is only present in the lease table if you are Process multiple data streams with the same KCL 2.x for Java consumer application. This is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and later.

- **stream name** The identifier of the data stream in the following format: `account-id:StreamName:streamCreationTimestamp`.

> **Note**
>
> This data is only present in the lease table if you are Process multiple data streams with the same KCL 2.x for Java consumer application. This is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and later.

Throughput

If your Amazon Kinesis Data Streams application receives provisioned-throughput exceptions, you should increase the provisioned throughput for the DynamoDB table. The KCL creates the table with a provisioned throughput of 10 reads per second and 10 writes per second, but this might not be sufficient for your application. For example, if your Amazon Kinesis Data Streams application does frequent checkpointing or operates on a stream that is composed of many shards, you might need more throughput.

For information about provisioned throughput in DynamoDB, see Read/Write Capacity Mode and Working with Tables and Data in the *Amazon DynamoDB Developer Guide*.

How a lease table is synchronized with the shards in a Kinesis data stream

Workers in KCL consumer applications use leases to process shards from a given data stream. The information on what worker is leasing what shard at any given time is stored in a lease table. The lease table must remain in sync with the latest shard information from the data stream while the KCL consumer application is running. KCL synchronizes the lease table with the shards information acquired from the Kinesis Data Streams service during the consumer application bootstraping (either when the consumer application is initialized or restarted) and also whenever a shard that is being processed reaches an end (resharding). In other words, the workers or a KCL consumer application are synchronized with the data stream that they are processing during the initial consumer application bootstrap and whenever the consumer application encounters a data stream reshard event.

Topics

- Synchronization in KCL 1.0 - 1.13 and KCL 2.0 - 2.2
- Synchronization in KCL 2.x, starting with KCL 2.3 and later
- Synchronization in KCL 1.x, starting with KCL 1.14 and later

Synchronization in KCL 1.0 - 1.13 and KCL 2.0 - 2.2

In KCL 1.0 - 1.13 and KCL 2.0 - 2.2, during consumer application's bootstraping and also during each data stream reshard event, KCL synchronizes the lease table with the shards information acquired from the Kinesis Data Streams service by invoking the `ListShards` or the `DescribeStream` discovery APIs. In all the KCL versions listed above, each worker of a KCL consumer application completes the following steps to perform the lease/shard synchronization process during the consumer application's bootstrapping and at each stream reshard event:

- Fetches all the shards for data the stream that is being processed
- Fetches all the shard leases from the lease table
- Filters out each open shard that does not have a lease in the lease table
- Iterates over all found open shards and for each open shard with no open parent:
  - Traverses the hierarchy tree through its ancestors path to determine if the shard is a descendant. A shard is considered a descendant, if an ancestor shard is being processed (lease entry for ancestor shard exists in the lease table) or if an ancestor shard should be processed (for example, if the initial position is `TRIM_HORIZON` or `AT_TIMESTAMP`)
  - If the open shard in context is a descendant, KCL checkpoints the shard based on initial position and creates leases for its parents, if required

Synchronization in KCL 2.x, starting with KCL 2.3 and later

Starting with the latest supported versions of KCL 2.x (KCL 2.3) and later, the library now supports the following changes to the synchronization process. These lease/shard synchronization changes significantly reduce the number of API calls made by KCL consumer applications to the Kinesis Data Streams service and optimize the lease management in your KCL consumer application.

- During application's bootstraping, if the lease table is empty, KCL utilizes the `ListShard` API's filtering option (the `ShardFilter` optional request parameter) to retrieve and create leases only for a snapshot of shards open at the time specified by the `ShardFilter` parameter. The `ShardFilter` parameter enables you to filter out the response of the `ListShards` API. The only required property of the `ShardFilter` parameter is `Type`. KCL uses the `Type` filter property and the following of its valid values to identify and return a snapshot of open shards that might require new leases:
  - `AT_TRIM_HORIZON` - the response includes all the shards that were open at `TRIM_HORIZON`.
  - `AT_LATEST` - the response includes only the currently open shards of the data stream.

- `AT_TIMESTAMP` - the response includes all shards whose start timestamp is less than or equal to the given timestamp and end timestamp is greater than or equal to the given timestamp or still open.

  `ShardFilter` is used when creating leases for an empty lease table to initialize leases for a snapshot of shards specified at `RetrievalConfig#initialPositionInStreamExtended`.

  For more information about `ShardFilter`, see https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html.

- Instead of all workers performing the lease/shard synchronization to keep the lease table up to date with the latest shards in the data stream, a single elected worker leader performs the lease/shard synchronization.
- KCL 2.3 uses the `ChildShards` return parameter of the `GetRecords` and the `SubscribeToShard` APIs to perform lease/shard synchronization that happens at `SHARD_END` for closed shards, allowing a KCL worker to only create leases for the child shards of the shard it finished processing. For shared throughout consumer applications, this optimization of the lease/shard synchronization uses the `ChildShards` parameter of the `GetRecords` API. For the dedicated throughput (enhanced fan-out) consumer applications, this optimization of the lease/shard synchronization uses the `ChildShards` parameter of the `SubscribeToShard` API. For more information, see GetRecords, SubscribeToShards, and ChildShard.
- With the above changes, the behavior of KCL is moving from the model of all workers learning about all existing shards to the model of workers learning only about the children shards of the shards that each worker owns. Therefore, in addition to the synchronization that happens during consumer application bootstraping and reshard events, KCL now also performs additional periodic shard/lease scans in order to identify any potential holes in the lease table (in other words, to learn about all new shards) to ensure the complete hash range of the data stream is being processed and create leases for them if required. `PeriodicShardSyncManager` is the component that is responsible for running periodic lease/shard scans.

  For more information about `PeriodicShardSyncManager` in KCL 2.3, see https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/LeaseManagementConfig.java#L201-L213.

  In KCL 2.3, new configuration options are available to configure `PeriodicShardSyncManager` in `LeaseManagementConfig`:

Name	Default value	Description
leasesRecoveryAuditorExecutionFrequencyMillis	120000 (2 minutes)	Frequency (in millis) of the auditor job to scan for partial leases in the lease table. If the auditor detects any hole in the leases for a stream, then it would trigger shard synchronization based on `leasesRecoveryAuditorInconsistencyConfidenceThreshold` .

Name	Default value	Description
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	Confidence threshold for the periodic auditor job to determine if leases for a data stream in the lease table are inconsistent. If the auditor finds same set of inconsistencies consecutively for a data stream for this many times, then it would trigger a shard synchronization.

New CloudWatch metrics are also now emitted to monitor the health of the `PeriodicShardSyncManager`. For more information, see PeriodicShardSyncManager.

- Including an optimization to `HierarchicalShardSyncer` to only create leases for one layer of shards.

Synchronization in KCL 1.x, starting with KCL 1.14 and later

Starting with the latest supported versions of KCL 1.x (KCL 1.14) and later, the library now supports the following changes to the synchronization process. These lease/shard synchronization changes significantly reduce the number of API calls made by KCL consumer applications to the Kinesis Data Streams service and optimize the lease management in your KCL consumer application.

- During application's bootstraping, if the lease table is empty, KCL utilizes the `ListShard` API's filtering option (the `ShardFilter` optional request parameter) to retrieve and create leases

only for a snapshot of shards open at the time specified by the `ShardFilter` parameter. The `ShardFilter` parameter enables you to filter out the response of the `ListShards` API. The only required property of the `ShardFilter` parameter is `Type`. KCL uses the `Type` filter property and the following of its valid values to identify and return a snapshot of open shards that might require new leases:

- `AT_TRIM_HORIZON` - the response includes all the shards that were open at `TRIM_HORIZON`.
- `AT_LATEST` - the response includes only the currently open shards of the data stream.
- `AT_TIMESTAMP` - the response includes all shards whose start timestamp is less than or equal to the given timestamp and end timestamp is greater than or equal to the given timestamp or still open.

`ShardFilter` is used when creating leases for an empty lease table to initialize leases for a snapshot of shards specified at `KinesisClientLibConfiguration#initialPositionInStreamExtended`.

For more information about `ShardFilter`, see https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html.

- Instead of all workers performing the lease/shard synchronization to keep the lease table up to date with the latest shards in the data stream, a single elected worker leader performs the lease/shard synchronization.
- KCL 1.14 uses the `ChildShards` return parameter of the `GetRecords` and the `SubscribeToShard` APIs to perform lease/shard synchronization that happens at `SHARD_END` for closed shards, allowing a KCL worker to only create leases for the child shards of the shard it finished processing. For more information, see GetRecords and ChildShard.
- With the above changes, the behavior of KCL is moving from the model of all workers learning about all existing shards to the model of workers learning only about the children shards of the shards that each worker owns. Therefore, in addition to the synchronization that happens during consumer application bootstraping and reshard events, KCL now also performs additional periodic shard/lease scans in order to identify any potential holes in the lease table (in other words, to learn about all new shards) to ensure the complete hash range of the data stream is being processed and create leases for them if required. `PeriodicShardSyncManager` is the component that is responsible for running periodic lease/shard scans.

  When `KinesisClientLibConfiguration#shardSyncStrategyType` is set to `ShardSyncStrategyType.SHARD_END`, `PeriodicShardSync` `leasesRecoveryAuditorInconsistencyConfidenceThreshold` is

used to determine the threshold for number of consecutive scans containing holes in the lease table after which to enforce a shard synchronization. When `KinesisClientLibConfiguration#shardSyncStrategyType` is set to `ShardSyncStrategyType.PERIODIC`, `leasesRecoveryAuditorInconsistencyConfidenceThreshold` is ignored.

For more information about `PeriodicShardSyncManager` in KCL 1.14, see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/KinesisClientLibConfiguration.java#L987-L999.

In KCL 1.14, new configuration option is available to configure `PeriodicShardSyncManager` in `LeaseManagementConfig`:

Name	Default value	Description
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	Confidence threshold for the periodic auditor job to determine if leases for a data stream in the lease table are inconsistent. If the auditor finds same set of inconsistencies consecutively for a data stream for this many times, then it would trigger a shard synchronization.

New CloudWatch metrics are also now emitted to monitor the health of the `PeriodicShardSyncManager`. For more information, see PeriodicShardSyncManager.

- KCL 1.14 now also supports deferred lease cleanup. Leases are deleted asynchronously by `LeaseCleanupManager` upon reaching `SHARD_END`, when a shard has either expired past the data stream's retention period or been closed as the result of a resharding operation.

  New configuration options are available to configure `LeaseCleanupManager`.

Name	Default value	Description
leaseCleanupIntervalMillis	1 minute	Interval at which to run lease cleanup thread.
completedLeaseCleanupIntervalMillis	5 minutes	Interval at which to check if a lease is completed or not.
garbageLeaseCleanupIntervalMillis	30 minutes	Interval at which to check if a lease is garbage (i.e trimmed past the data stream's retention period) or not.

- Including an optimization to `KinesisShardSyncer` to only create leases for one layer of shards.

Process multiple data streams with the same KCL 2.x for Java consumer application

This section describes the following changes in KCL 2.x for Java that enable you to create KCL consumer applications that can process more than one data stream at the same time.

> **Important**
>
> Multistream processing is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and later.
> Multistream processing is NOT supported for any other languages in which KCL 2.x can be implemented.
> Multistream processing is NOT supported in any versions of KCL 1.x.

- **MultistreamTracker interface**

  To build a consumer application that can process multiple streams at the same time, you must implement a new interface called MultistreamTracker. This interface includes the `streamConfigList` method that returns the list of data streams and their configurations to be processed by the KCL consumer application. Notice that the data streams that are being processed can be changed during the consumer application runtime. `streamConfigList` is called periodically by the KCL to learn about the changes in data streams to process.

  The `streamConfigList` method populates the StreamConfig list.

```
package software.amazon.kinesis.common;

import lombok.Data;
import lombok.experimental.Accessors;

@Data
@Accessors(fluent = true)
public class StreamConfig {
    private final StreamIdentifier streamIdentifier;
    private final InitialPositionInStreamExtended initialPositionInStreamExtended;
    private String consumerArn;
}
```

Note that the `StreamIdentifier` and `InitialPositionInStreamExtended` are required fields, while `consumerArn` is optional. You must provide the `consumerArn` only if you are using KCL 2.x to implement an enhanced fan-out consumer application.

For more information about `StreamIdentifier`, see https://github.com/awslabs/amazon-kinesis-client/blob/v2.5.8/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L129. To create a `StreamIdentifier`, we recommend that you create a multistream instance from the `streamArn` and the `streamCreationEpoch` that is available in v2.5.0 and later. In KCL v2.3 and v2.4, which don't support `streamArm`, create a multistream instance by using the format `account-id:StreamName:streamCreationTimestamp`. This format will be deprecated and no longer supported starting with the next major release.

`MultistreamTracker` also includes a strategy for deleting leases of old streams in the lease table (`formerStreamsLeasesDeletionStrategy`). Notice that the strategy CANNOT be changed during the consumer application runtime. For more information, see https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java

- ConfigsBuilder is a an application-wide class that you can use to specify all of the KCL 2.x configuration settings to be used when building your KCL consumer application. `ConfigsBuilder` class now has support for the `MultistreamTracker` interface. You can initialize ConfigsBuilder either with the name of the one data stream to consume records from:

```
/**
    * Constructor to initialize ConfigsBuilder with StreamName
    * @param streamName
    * @param applicationName
    * @param kinesisClient
    * @param dynamoDBClient
    * @param cloudWatchClient
    * @param workerIdentifier
    * @param shardRecordProcessorFactory
    */
   public ConfigsBuilder(@NonNull String streamName, @NonNull String
applicationName,
           @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
dynamoDBClient,
```

```
            @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
 workerIdentifier,
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.right(streamName);
        this.applicationName = applicationName;
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

Or you can initialize ConfigsBuilder with `MultiStreamTracker` if you want to implement a KCL consumer application that processes multiple streams at the same time.

```
* Constructor to initialize ConfigsBuilder with MultiStreamTracker
     * @param multiStreamTracker
     * @param applicationName
     * @param kinesisClient
     * @param dynamoDBClient
     * @param cloudWatchClient
     * @param workerIdentifier
     * @param shardRecordProcessorFactory
     */
    public ConfigsBuilder(@NonNull MultiStreamTracker multiStreamTracker, @NonNull
 String applicationName,
            @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
 dynamoDBClient,
            @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
 workerIdentifier,
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.left(multiStreamTracker);
        this.applicationName = applicationName;
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

- With multistream support implemented for your KCL consumer application, each row of the application's lease table now contains the shard ID and the stream name of the multiple data streams that this application processes.
- When multistream support for your KCL consumer application is implemented, the leaseKey takes the following structure: `account-id:StreamName:streamCreationTimestamp:ShardId`. For example, `111111111:multiStreamTest-1:12345:shardId-000000000336`.

> **Important**
>
> When your existing KCL consumer application is configured to process only one data stream, the leaseKey (which is the hash key for the lease table) is the shard ID. If you reconfigure this existing KCL consumer application to process multiple data streams, it breaks your lease table, because with multistream support, the leaseKey structure must be as follows: `account-id:StreamName:StreamCreationTimestamp:ShardId`.

Use the KCL with the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue Schema Registry. The AWS Glue Schema Registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS GlueSchema Registry lets you improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the KCL in Java.

> **Important**
>
> Currently, Kinesis Data Streams and AWS Glue Schema Registry integration is only supported for the Kinesis data streams that use KCL 2.3 consumers implemented in Java. Multi-language support is not provided. KCL 1.0 consumers are not supported. KCL 2.x consumers prior to KCL 2.3 are not supported.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the KCL, see the "Interacting with Data Using the KPL/KCL Libraries" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Develop custom consumers with shared throughput

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

If you don't need dedicated throughput when receiving data from Kinesis Data Streams, and if you don't need read propagation delays under 200 ms, you can build consumer applications as described in the following topics. You can use the Kinesis Client Library (KCL) or the AWS SDK for Java.

Topics

- Develop custom consumers with shared throughput using KCL

For information about building consumers that can receive records from Kinesis data streams with dedicated throughput, see Develop enhanced fan-out consumers with dedicated throughput.

Develop custom consumers with shared throughput using KCL

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

One of the methods of developing a custom consumer application with shared throughput is to use the Kinesis Client Library (KCL).

Choose from the following topics for the KCL version you are using.

Topics

- Develop KCL 1.x consumers
- Develop KCL 2.x Consumers

Develop KCL 1.x consumers

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can develop a consumer application for Amazon Kinesis Data Streams using the Kinesis Client Library (KCL).

For more information about KCL, see About KCL (previous versions).

Choose from the following topics depending on the option you want to use.

Contents

- Develop a Kinesis Client Library consumer in Java
- Develop a Kinesis Client Library consumer in Node.js
- Develop a Kinesis Client Library consumer in .NET
- Develop a Kinesis Client Library consumer in Python
- Develop a Kinesis Client Library consumer in Ruby

Develop a Kinesis Client Library consumer in Java

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Java. To view the Javadoc reference, see the AWS Javadoc topic for Class AmazonKinesisClient.

To download the Java KCL from GitHub, go to Kinesis Client Library (Java). To locate the Java KCL on Apache Maven, go to the KCL search results page. To download sample code for a Java KCL consumer application from GitHub, go to the KCL for Java sample project page on GitHub.

The sample application uses Apache Commons Logging. You can change the logging configuration in the static `configure` method defined in the `AmazonKinesisApplicationSample.java` file. For more information about how to use Apache Commons Logging with Log4j and AWS Java applications, see Logging with Log4j in the *AWS SDK for Java Developer Guide*.

You must complete the following tasks when implementing a KCL consumer application in Java:

Tasks

- Implement the IRecordProcessor methods
- Implement a class factory for the IRecordProcessor interface
- Create a worker
- Modify the configuration properties
- Migrate to Version 2 of the record processor interface

Implement the IRecordProcessor methods

The KCL currently supports two versions of the `IRecordProcessor` interface:The original interface is available with the first version of the KCL, and version 2 is available starting with KCL version 1.5.0. Both interfaces are fully supported. Your choice depends on your specific scenario requirements. Refer to your locally built Javadocs or the source code to see all the differences. The following sections outline the minimal implementation for getting started.

IRecordProcessor Versions

- Original Interface (Version 1)
- Updated interface (Version 2)

Original Interface (Version 1)

The original `IRecordProcessor` interface (`package com.amazonaws.services.kinesis.clientlibrary.interfaces`) exposes the following record processor methods that your consumer must implement. The sample provides implementations that you can use as a starting point (see `AmazonKinesisApplicationSampleRecordProcessor.java`).

```
public void initialize(String shardId)
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
 checkpointer)
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

initialize

The KCL calls the `initialize` method when the record processor is instantiated, passing a specific shard ID as a parameter. This record processor processes only this shard and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard may be processed by more than one worker, see Use resharding, scaling, and parallel processing to change the number of shards.

```
public void initialize(String shardId)
```

processRecords

The KCL calls the `processRecords` method, passing a list of data record from the shard specified by the `initialize(shardId)` method. The record processor processes the data in these records according to the semantics of the consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
 checkpointer)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `Record` class exposes the following methods that provide access to the record's data, sequence number, and partition key.

```
record.getData()
record.getSequenceNumber()
record.getPartitionKey()
```

In the sample, the private method `processRecordsWithRetries` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a checkpointer (`IRecordProcessorCheckpointer`) to `processRecords`. The record processor calls the `checkpoint` method on this interface to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL won't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` only after it has processed all the records

in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `processRecords`. A processor could, for example, call `checkpoint` on every third call to `processRecords`. You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

In the sample, the private method `checkpoint` shows how to call `IRecordProcessorCheckpointer.checkpoint` using the appropriate exception handling and retry logic.

The KCL relies on `processRecords` to handle any exceptions that arise from processing the data records. If an exception is thrown from `processRecords`, the KCL skips over the data records that were passed before the exception. That is, these records are not re-sent to the record processor that threw the exception or to any other record processor in the consumer.

shutdown

The KCL calls the `shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown reason is `ZOMBIE`).

```
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `IRecordProcessorCheckpointer` interface to `shutdown`. If the shutdown reason is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Updated interface (Version 2)

The updated `IRecordProcessor` interface (`package com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`) exposes the following record processor methods that your consumer must implement:

```
void initialize(InitializationInput initializationInput)
void processRecords(ProcessRecordsInput processRecordsInput)
void shutdown(ShutdownInput shutdownInput)
```

All of the arguments from the original version of the interface are accessible through get methods on the container objects. For example, to retrieve the list of records in `processRecords()`, you can use `processRecordsInput.getRecords()`.

As of version 2 of this interface (KCL 1.5.0 and later), the following new inputs are available in addition to the inputs provided by the original interface:

starting sequence number

In the `InitializationInput` object passed to the `initialize()` operation, the starting sequence number from which records would be provided to the record processor instance. This is the sequence number that was last checkpointed by the record processor instance previously processing the same shard. This is provided in case your application needs this information.

pending checkpoint sequence number

In the `InitializationInput` object passed to the `initialize()` operation, the pending checkpoint sequence number (if any) that could not be committed before the previous record processor instance stopped.

Implement a class factory for the IRecordProcessor interface

You also need to implement a factory for the class that implements the record processor methods. When your consumer instantiates the worker, it passes a reference to this factory.

The sample implements the factory class in the file `AmazonKinesisApplicationSampleRecordProcessorFactory.java` using the original record processor interface. If you want the class factory to create version 2 record processors, use the package name `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`.

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {
    /**
    * Constructor.
    */
    public SampleRecordProcessorFactory() {
        super();
    }
    /**
    * {@inheritDoc}
    */
```

```
    @Override
    public IRecordProcessor createProcessor() {
        return new SampleRecordProcessor();
    }
}
```

Create a worker

As discussed in Implement the IRecordProcessor methods, there are two versions of the KCL record processor interface to choose from, which affects how you would create a worker. The original record processor interface uses the following code structure to create a worker:

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker(recordProcessorFactory, config);
```

With version 2 of the record processor interface, you can use `Worker.Builder` to create a worker without needing to worry about which constructor to use and the order of the arguments. The updated record processor interface uses the following code structure to create a worker:

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

Modify the configuration properties

The sample provides default values for configuration properties. This configuration data for the worker is then consolidated in a `KinesisClientLibConfiguration` object. This object and a reference to the class factory for `IRecordProcessor` are passed in the call that instantiates the worker. You can override any of these properties with your own values using a Java properties file (see `AmazonKinesisApplicationSample.java`).

Application name

The KCL requires an application name that is unique across your applications, and across Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Set up credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. For example, if you are running your consumer on an EC2 instance, we recommend that you launch the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer running on an EC2 instance.

The sample application first attempts to retrieve IAM credentials from instance metadata:

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

If the sample application cannot obtain credentials from the instance metadata, it attempts to retrieve credentials from a properties file:

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

For more information about instance metadata, see Instance Metadata in the *Amazon EC2 User Guide*.

Use the worker ID for multiple instances

The sample initialization code creates an ID for the worker, `workerId`, using the name of the local computer and appending a globally unique identifier as shown in the following code snippet. This approach supports the scenario of multiple instances of the consumer application running on a single computer.

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +
 UUID.randomUUID();
```

Migrate to Version 2 of the record processor interface

If you want to migrate code that uses the original interface, in addition to the steps described previously, the following steps are required:

1. Change your record processor class to import the version 2 record processor interface:

   ```
   import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
   ```

2. Change the references to inputs to use `get` methods on the container objects. For example, in the `shutdown()` operation, change "`checkpointer`" to "`shutdownInput.getCheckpointer()`".
3. Change your record processor factory class to import the version 2 record processor factory interface:

   ```
   import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
   ```

4. Change the construction of the worker to use `Worker.Builder`. For example:

   ```
   final Worker worker = new Worker.Builder()
       .recordProcessorFactory(recordProcessorFactory)
       .config(config)
       .build();
   ```

Develop a Kinesis Client Library consumer in Node.js

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Node.js.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Node.js and write your consumer app entirely in Node.js, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Node.js KCL from GitHub, go to Kinesis Client Library (Node.js).

Sample Code Downloads

There are two code samples available for KCL in Node.js:

- basic-sample

  Used in the following sections to illustrate the fundamentals of building a KCL consumer application in Node.js.
- click-stream-sample

  Slightly more advanced and uses a real-world scenario, after you have familiarized yourself with the basic sample code. This sample is not discussed here but has a README file with more information.

You must complete the following tasks when implementing a KCL consumer application in Node.js:

Tasks

- Implement the record processor
- Modify the configuration properties

Implement the record processor

The simplest possible consumer using the KCL for Node.js must implement a `recordProcessor` function, which in turn contains the functions `initialize`, `processRecords`, and

shutdown. The sample provides an implementation that you can use as a starting point (see sample_kcl_app.js).

```
function recordProcessor() {
  // return an object that implements initialize, processRecords and shutdown
 functions.}
```

initialize

The KCL calls the initialize function when the record processor starts. This record processor processes only the shard ID passed as initializeInput.shardId, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard might be processed by more than one worker, see Use resharding, scaling, and parallel processing to change the number of shards.

```
initialize: function(initializeInput, completeCallback)
```

processRecords

The KCL calls this function with input that contains a list of data records from the shard specified to the initialize function. The record processor that you implement processes the data in these records according to the semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
processRecords: function(processRecordsInput, completeCallback)
```

In addition to the data itself, the record also contains a sequence number and partition key, which the worker can use when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The record dictionary exposes the following key-value pairs to access the record's data, sequence number, and partition key:

```
record.data
record.sequenceNumber
record.partitionKey
```

Note that the data is Base64-encoded.

In the basic sample, the function `processRecords` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for with a `checkpointer` object passed as `processRecordsInput.checkpointer`. Your record processor calls the `checkpointer.checkpoint` function to inform the KCL how far it has progressed in processing the records in the shard. In the event that the worker fails, the KCL uses this information when you restart the processing of the shard so that it continues from the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass the sequence number to the `checkpoint` function, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` **only** after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `processRecords`. A processor could, for example, call `checkpoint` on every third call, or some event external to your record processor, such as a custom verification/validation service you've implemented.

You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

The basic sample application shows the simplest possible call to the `checkpointer.checkpoint` function. You can add other checkpointing logic you need for your consumer at this point in the function.

shutdown

The KCL calls the `shutdown` function either when processing ends (`shutdownInput.reason` is `TERMINATE`) or the worker is no longer responding (`shutdownInput.reason` is `ZOMBIE`).

```
shutdown: function(shutdownInput, completeCallback)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `shutdownInput.checkpointer` object to `shutdown`. If the shutdown reason is `TERMINATE`, you should make sure that the record processor has finished processing any data records, and then call the `checkpoint` function on this interface.

Modify the configuration properties

The sample provides default values for the configuration properties. You can override any of these properties with your own values (see `sample.properties` in the basic sample).

Application name

The KCL requires an application that this is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Set up credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. The `sample.properties` file must make your credentials available to one of the credentials providers in the default credential providers chain. If you are running your consumer on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

The following example configures KCL to process a Kinesis data stream named `kclnodejssample` using the record processor supplied in `sample_kcl_app.js`:

```
# The Node.js executable script
```

```
executableName = node sample_kcl_app.js
# The name of an Amazon Kinesis stream to process
streamName = kclnodejssample
# Unique KCL application name
applicationName = kclnodejssample
# Use default AWS credentials provider chain
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain
# Read from the beginning of the stream
initialPositionInStream = TRIM_HORIZON
```

Develop a Kinesis Client Library consumer in .NET

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses .NET.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for .NET and write your consumer app entirely in .NET, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the .NET KCL from GitHub, go to Kinesis Client Library (.NET). To download sample code for a .NET KCL consumer application, go to the KCL for .NET sample consumer project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in .NET:

Tasks

- Implement the IRecordProcessor class methods
- Modify the configuration properties

Implement the IRecordProcessor class methods

The consumer must implement the following methods for `IRecordProcessor`. The sample consumer provides implementations that you can use as a starting point (see the `SampleRecordProcessor` class in `SampleConsumer/AmazonKinesisSampleConsumer.cs`).

```
public void Initialize(InitializationInput input)
public void ProcessRecords(ProcessRecordsInput input)
public void Shutdown(ShutdownInput input)
```

Initialize

The KCL calls this method when the record processor is instantiated, passing a specific shard ID in the `input` parameter (`input.ShardId`). This record processor processes only this shard, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard might be processed by more than one worker, see Use resharding, scaling, and parallel processing to change the number of shards.

```
public void Initialize(InitializationInput input)
```

ProcessRecords

The KCL calls this method, passing a list of data records in the `input` parameter (`input.Records`) from the shard specified by the `Initialize` method. The record processor that you implement processes the data in these records according to the semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
public void ProcessRecords(ProcessRecordsInput input)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `Record` class exposes the following to access the record's data, sequence number, and partition key:

```
byte[] Record.Data
string Record.SequenceNumber
string Record.PartitionKey
```

In the sample, the method `ProcessRecordsWithRetries` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a `Checkpointer` object to `ProcessRecords` (`input.Checkpointer`). The record processor calls the `Checkpointer.Checkpoint` method to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `Checkpointer.Checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `Checkpointer.Checkpoint` signifies that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `Checkpointer.Checkpoint` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `Checkpointer.Checkpoint` on each call to `ProcessRecords`. A processor could, for example, call `Checkpointer.Checkpoint` on every third or fourth call. You can optionally specify the exact sequence number of a record as a parameter to `Checkpointer.Checkpoint`. In this case, the KCL assumes that records have been processed only up to that record.

In the sample, the private method `Checkpoint(Checkpointer checkpointer)` shows how to call the `Checkpointer.Checkpoint` method using appropriate exception handling and retry logic.

The KCL for .NET handles exceptions differently from other KCL language libraries in that it does not handle any exceptions that arise from processing the data records. Any uncaught exceptions from user code crashes the program.

Shutdown

The KCL calls the `Shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown `input.Reason` value is `ZOMBIE`).

```
public void Shutdown(ShutdownInput input)
```

Processing ends when the record processor does not receive any further records from the shard, because the shard was split or merged, or the stream was deleted.

The KCL also passes a `Checkpointer` object to `shutdown`. If the shutdown reason is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Modify the configuration properties

The sample consumer provides default values for the configuration properties. You can override any of these properties with your own values (see `SampleConsumer/kcl.properties`).

Application name

The KCL requires an application that this is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Set up credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a

credentials provider. The sample.properties must make your credentials available to one of the credentials providers in the default credential providers chain. If you are running your consumer application on an EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer running on an EC2 instance.

The sample's properties file configures KCL to process a Kinesis data stream called "words" using the record processor supplied in `AmazonKinesisSampleConsumer.cs`.

Develop a Kinesis Client Library consumer in Python

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Python.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Python and write your consumer app entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Python KCL from GitHub, go to Kinesis Client Library (Python). To download sample code for a Python KCL consumer application, go to the KCL for Python sample project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in Python:

Tasks

- Implement the RecordProcessor class methods
- Modify the configuration properties

Implement the RecordProcessor class methods

The `RecordProcess` class must extend the `RecordProcessorBase` to implement the following methods. The sample provides implementations that you can use as a starting point (see `sample_kclpy_app.py`).

```
def initialize(self, shard_id)
def process_records(self, records, checkpointer)
def shutdown(self, checkpointer, reason)
```

initialize

The KCL calls the `initialize` method when the record processor is instantiated, passing a specific shard ID as a parameter. This record processor processes only this shard, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard may be processed by more than one worker, see Use resharding, scaling, and parallel processing to change the number of shards.

```
def initialize(self, shard_id)
```

process_records

The KCL calls this method, passing a list of data record from the shard specified by the `initialize` method. The record processor that you implement processes the data in these records according to the semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
def process_records(self, records, checkpointer)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `record` dictionary exposes the following key-value pairs to access the record's data, sequence number, and partition key:

```
record.get('data')
record.get('sequenceNumber')
record.get('partitionKey')
```

Note that the data is Base64-encoded.

In the sample, the method `process_records` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a `Checkpointer` object to `process_records`. The record processor calls the `checkpoint` method on this object to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `process_records`. A processor could, for example, call `checkpoint` on every third call. You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

In the sample, the private method `checkpoint` shows how to call the `Checkpointer.checkpoint` method using appropriate exception handling and retry logic.

The KCL relies on `process_records` to handle any exceptions that arise from processing the data records. If an exception is thrown from `process_records`, the KCL skips over the data records

that were passed to `process_records` before the exception. That is, these records are not re-sent to the record processor that threw the exception or to any other record processor in the consumer.

shutdown

The KCL calls the `shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown `reason` is `ZOMBIE`).

```
def shutdown(self, checkpointer, reason)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `Checkpointer` object to `shutdown`. If the shutdown `reason` is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Modify the configuration properties

The sample provides default values for the configuration properties. You can override any of these properties with your own values (see `sample.properties`).

Application name

The KCL requires an application name that is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers that are associated with this application name are assumed to be working together on the same stream. These workers can be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Set up credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. The sample.properties must make your credentials available to one of the credentials providers in the default credential providers chain. If you are running your consumer application on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

The sample's properties file configures KCL to process a Kinesis data stream called "words" using the record processor supplied in `sample_kclpy_app.py`.

Develop a Kinesis Client Library consumer in Ruby

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Ruby.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Ruby and write your consumer app entirely in Ruby, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Ruby KCL from GitHub, go to Kinesis Client Library (Ruby). To download sample code for a Ruby KCL consumer application, go to the KCL for Ruby sample project page on GitHub.

For more information about the KCL Ruby support library, see KCL Ruby Gems Documentation.

Develop KCL 2.x Consumers

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

This topic shows you how to use version 2.0 of the Kinesis Client Library (KCL).

For more information about the KCL, see the overview provided in Developing Consumers Using the Kinesis Client Library 1.x.

Choose from the following topics depending on the option you want to use.

Topics

- Develop a Kinesis Client Library consumer in Java
- Develop a Kinesis Client Library consumer in Python
- Develop enhanced fan-out consumers with KCL 2.x

Develop a Kinesis Client Library consumer in Java

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For

information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

The following code shows an example implementation in Java of `ProcessorFactory` and `RecordProcessor`. If you want to take advantage of the enhanced fan-out feature, see Using Consumers with Enhanced Fan-Out .

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */


/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.retrieval.polling.PollingConfig;

/**
 * This class will run a simple app that uses the KCL to read data and uses the AWS SDK
 to publish data.
 * Before running this program you must first create a Kinesis stream through the AWS
 console or AWS SDK.
 */
```

```
public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    /**
     * Invoke the main method with 2 args: the stream name and (optionally) the region.
     * Verifies valid inputs and then starts running the app.
     */
    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
 The Region may be specified as the second argument.");
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    /**
     * Constructor sets streamName and region. It also creates a KinesisClient object
 to send data to Kinesis.
     * This KinesisClient is used to send dummy data so that the consumer has something
 to read; it is also used
     * indirectly by the KCL to handle the consumption of the data.
     */
    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {
```

```
        /**
         * Sends dummy data to Kinesis. Not relevant to consuming the data with the KCL
         */
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        /**
         * Sets up configuration for the KCL, including DynamoDB and CloudWatch
dependencies. The final argument, a
         * ShardRecordProcessorFactory, is where the logic for record processing lives,
and is located in a private
         * class below.
         */
        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        /**
         * The Scheduler (also called Worker in earlier versions of the KCL) is the
entry point to the KCL. This
         * instance is configured with defaults provided by the ConfigsBuilder.
         */
        Scheduler scheduler = new Scheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                configsBuilder.retrievalConfig().retrievalSpecificConfig(new
PollingConfig(streamName, kinesisClient))
        );

        /**
         * Kickoff the Scheduler. Record processing of the stream of dummy data will
continue indefinitely
         * until an exit is triggered.
         */
```

```
        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        /**
         * Allows termination of app by pressing Enter.
         */
        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        /**
         * Stops sending dummy data.
         */
        log.info("Cancelling producer and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();

        /**
         * Stops consuming data. Finishes processing the current batch of data already
received from Kinesis
         * before shutting down.
         */
        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown.  Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    /**
```

```
     * Sends a single record of dummy data to Kinesis.
     */
    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }

    /**
     * The implementation of the ShardRecordProcessor interface is where the heart of
the record processing logic lives.
     * In this example all we do to 'process' is log info about the records.
     */
    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        /**
         * Invoked by the KCL before data records are delivered to the
ShardRecordProcessor instance (via
         * processRecords). In this example we do nothing except some logging.
         *
```

```
         * @param initializationInput Provides information related to initialization.
         */
        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Handles record processing logic. The Amazon Kinesis Client Library will
invoke this method to deliver
         * data records to the application. In this example we simply log our records.
         *
         * @param processRecordsInput Provides the records to be processed as well as
information and capabilities
         *                            related to them (e.g. checkpointing).
         */
        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)",
processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /** Called when the lease tied to this record processor has been lost. Once the
lease has been lost,
         * the record processor can no longer checkpoint.
         *
         * @param leaseLostInput Provides access to functions and data related to the
loss of the lease.
         */
```

```
        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Called when all data on this shard has been processed. Checkpointing must
occur in the method for record
         * processing to be considered complete; an exception will be thrown otherwise.
         *
         * @param shardEndedInput Provides access to a checkpointer method for
completing processing of the shard.
         */
        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Invoked when Scheduler has been requested to shut down (i.e. we decide to
stop running the app by pressing
         * Enter). Checkpoints and logs the data a final time.
         *
         * @param shutdownRequestedInput Provides access to a checkpointer, allowing a
record processor to checkpoint
         *                               before the shutdown is completed.
         */
        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
```

```
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Develop a Kinesis Client Library consumer in Python

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Python.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Python and write your consumer app entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Python KCL from GitHub, go to Kinesis Client Library (Python). To download sample code for a Python KCL consumer application, go to the KCL for Python sample project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in Python:

Tasks

- Implement the RecordProcessor class methods
- Modify the configuration properties

Implement the RecordProcessor class methods

The `RecordProcess` class must extend the `RecordProcessorBase` class to implement the following methods:

```
initialize
process_records
shutdown_requested
```

This sample provides implementations that you can use as a starting point.

```
#!/usr/bin/env python

# Copyright 2014-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Amazon Software License (the "License").
# You may not use this file except in compliance with the License.
# A copy of the License is located at
#
# http://aws.amazon.com/asl/
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.

from __future__ import print_function

import sys
import time

from amazon_kclpy import kcl
from amazon_kclpy.v3 import processor
```

```
class RecordProcessor(processor.RecordProcessorBase):
    """
    A RecordProcessor processes data from a shard in a stream. Its methods will be
 called with this pattern:

    * initialize will be called once
    * process_records will be called zero or more times
    * shutdown will be called if this MultiLangDaemon instance loses the lease to this
 shard, or the shard ends due
        a scaling change.
    """
    def __init__(self):
        self._SLEEP_SECONDS = 5
        self._CHECKPOINT_RETRIES = 5
        self._CHECKPOINT_FREQ_SECONDS = 60
        self._largest_seq = (None, None)
        self._largest_sub_seq = None
        self._last_checkpoint_time = None

    def log(self, message):
        sys.stderr.write(message)

    def initialize(self, initialize_input):
        """
        Called once by a KCLProcess before any calls to process_records

        :param amazon_kclpy.messages.InitializeInput initialize_input: Information
 about the lease that this record
            processor has been assigned.
        """
        self._largest_seq = (None, None)
        self._last_checkpoint_time = time.time()

    def checkpoint(self, checkpointer, sequence_number=None, sub_sequence_number=None):
        """
        Checkpoints with retries on retryable exceptions.

        :param amazon_kclpy.kcl.Checkpointer checkpointer: the checkpointer provided to
 either process_records
            or shutdown
        :param str or None sequence_number: the sequence number to checkpoint at.
        :param int or None sub_sequence_number: the sub sequence number to checkpoint
 at.
        """
```

```
        for n in range(0, self._CHECKPOINT_RETRIES):
            try:
                checkpointer.checkpoint(sequence_number, sub_sequence_number)
                return
            except kcl.CheckpointError as e:
                if 'ShutdownException' == e.value:
                    #
                    # A ShutdownException indicates that this record processor should
be shutdown. This is due to
                    # some failover event, e.g. another MultiLangDaemon has taken the
lease for this shard.
                    #
                    print('Encountered shutdown exception, skipping checkpoint')
                    return
                elif 'ThrottlingException' == e.value:
                    #
                    # A ThrottlingException indicates that one of our dependencies is
is over burdened, e.g. too many
                    # dynamo writes. We will sleep temporarily to let it recover.
                    #
                    if self._CHECKPOINT_RETRIES - 1 == n:
                        sys.stderr.write('Failed to checkpoint after {n} attempts,
giving up.\n'.format(n=n))
                        return
                    else:
                        print('Was throttled while checkpointing, will attempt again in
{s} seconds'
                              .format(s=self._SLEEP_SECONDS))
                elif 'InvalidStateException' == e.value:
                    sys.stderr.write('MultiLangDaemon reported an invalid state while
checkpointing.\n')
                else:  # Some other error
                    sys.stderr.write('Encountered an error while checkpointing, error
was {e}.\n'.format(e=e))
            time.sleep(self._SLEEP_SECONDS)

    def process_record(self, data, partition_key, sequence_number,
sub_sequence_number):
        """
        Called for each record that is passed to process_records.

        :param str data: The blob of data that was contained in the record.
        :param str partition_key: The key associated with this recod.
        :param int sequence_number: The sequence number associated with this record.
```

```
        :param int sub_sequence_number: the sub sequence number associated with this
record.
        """
        ###################################
        # Insert your processing logic here
        ###################################
        self.log("Record (Partition Key: {pk}, Sequence Number: {seq}, Subsequence
Number: {sseq}, Data Size: {ds}"
                 .format(pk=partition_key, seq=sequence_number,
sseq=sub_sequence_number, ds=len(data)))

    def should_update_sequence(self, sequence_number, sub_sequence_number):
        """
        Determines whether a new larger sequence number is available

        :param int sequence_number: the sequence number from the current record
        :param int sub_sequence_number: the sub sequence number from the current record
        :return boolean: true if the largest sequence should be updated, false
otherwise
        """
        return self._largest_seq == (None, None) or sequence_number >
self._largest_seq[0] or \
            (sequence_number == self._largest_seq[0] and sub_sequence_number >
self._largest_seq[1])

    def process_records(self, process_records_input):
        """
        Called by a KCLProcess with a list of records to be processed and a
checkpointer which accepts sequence numbers
        from the records to indicate where in the stream to checkpoint.

        :param amazon_kclpy.messages.ProcessRecordsInput process_records_input: the
records, and metadata about the
            records.
        """
        try:
            for record in process_records_input.records:
                data = record.binary_data
                seq = int(record.sequence_number)
                sub_seq = record.sub_sequence_number
                key = record.partition_key
                self.process_record(data, key, seq, sub_seq)
                if self.should_update_sequence(seq, sub_seq):
                    self._largest_seq = (seq, sub_seq)
```

```
            #
            # Checkpoints every self._CHECKPOINT_FREQ_SECONDS seconds
            #
            if time.time() - self._last_checkpoint_time >
 self._CHECKPOINT_FREQ_SECONDS:
                self.checkpoint(process_records_input.checkpointer,
 str(self._largest_seq[0]), self._largest_seq[1])
                self._last_checkpoint_time = time.time()

        except Exception as e:
            self.log("Encountered an exception while processing records. Exception was
 {e}\n".format(e=e))

    def lease_lost(self, lease_lost_input):
        self.log("Lease has been lost")

    def shard_ended(self, shard_ended_input):
        self.log("Shard has ended checkpointing")
        shard_ended_input.checkpointer.checkpoint()

    def shutdown_requested(self, shutdown_requested_input):
        self.log("Shutdown has been requested, checkpointing.")
        shutdown_requested_input.checkpointer.checkpoint()


if __name__ == "__main__":
    kcl_process = kcl.KCLProcess(RecordProcessor())
    kcl_process.run()
```

Modify the configuration properties

The sample provides default values for the configuration properties, as shown in the following script. You can override any of these properties with your own values.

```
# The script that abides by the multi-language protocol. This script will
# be executed by the MultiLangDaemon, which will communicate with this script
# over STDIN and STDOUT according to the multi-language protocol.
executableName = sample_kclpy_app.py

# The name of an Amazon Kinesis stream to process.
streamName = words
```

```
# Used by the KCL as the name of this application. Will be used as the name
# of an Amazon DynamoDB table which will store the lease and checkpoint
# information for workers with this application name
applicationName = PythonKCLSample

# Users can change the credentials provider the KCL will use to retrieve credentials.
# The DefaultAWSCredentialsProviderChain checks several other providers, which is
# described here:
# http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/auth/
DefaultAWSCredentialsProviderChain.html
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain

# Appended to the user agent of the KCL. Does not impact the functionality of the
# KCL in any other way.
processingLanguage = python/2.7

# Valid options at TRIM_HORIZON or LATEST.
# See http://docs.aws.amazon.com/kinesis/latest/APIReference/
API_GetShardIterator.html#API_GetShardIterator_RequestSyntax
initialPositionInStream = TRIM_HORIZON

# The following properties are also available for configuring the KCL Worker that is
 created
# by the MultiLangDaemon.

# The KCL defaults to us-east-1
#regionName = us-east-1

# Fail over time in milliseconds. A worker which does not renew it's lease within this
 time interval
# will be regarded as having problems and it's shards will be assigned to other
 workers.
# For applications that have a large number of shards, this msy be set to a higher
 number to reduce
# the number of DynamoDB IOPS required for tracking leases
#failoverTimeMillis = 10000

# A worker id that uniquely identifies this worker among all workers using the same
 applicationName
# If this isn't provided a MultiLangDaemon instance will assign a unique workerId to
 itself.
#workerId =
```

```
# Shard sync interval in milliseconds - e.g. wait for this long between shard sync
 tasks.
#shardSyncIntervalMillis = 60000

# Max records to fetch from Kinesis in a single GetRecords call.
#maxRecords = 10000

# Idle time between record reads in milliseconds.
#idleTimeBetweenReadsInMillis = 1000

# Enables applications flush/checkpoint (if they have some data "in progress", but
 don't get new data for while)
#callProcessRecordsEvenForEmptyRecordList = false

# Interval in milliseconds between polling to check for parent shard completion.
# Polling frequently will take up more DynamoDB IOPS (when there are leases for shards
 waiting on
# completion of parent shards).
#parentShardPollIntervalMillis = 10000

# Cleanup leases upon shards completion (don't wait until they expire in Kinesis).
# Keeping leases takes some tracking/resources (e.g. they need to be renewed,
 assigned), so by default we try
# to delete the ones we don't need any longer.
#cleanupLeasesUponShardCompletion = true

# Backoff time in milliseconds for Amazon Kinesis Client Library tasks (in the event of
 failures).
#taskBackoffTimeMillis = 500

# Buffer metrics for at most this long before publishing to CloudWatch.
#metricsBufferTimeMillis = 10000

# Buffer at most this many metrics before publishing to CloudWatch.
#metricsMaxQueueSize = 10000

# KCL will validate client provided sequence numbers with a call to Amazon Kinesis
 before checkpointing for calls
# to RecordProcessorCheckpointer#checkpoint(String) by default.
#validateSequenceNumberBeforeCheckpointing = true

# The maximum number of active threads for the MultiLangDaemon to permit.
# If a value is provided then a FixedThreadPool is used with the maximum
# active threads set to the provided value. If a non-positive integer or no
```

```
# value is provided a CachedThreadPool is used.
#maxActiveThreads = 0
```

Application name

The KCL requires an application name that is unique among your applications and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers that are associated with this application name are assumed to be working together on the same stream. These workers can be distributed across multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. If you run your consumer application on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

Develop enhanced fan-out consumers with KCL 2.x

Important

Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For

information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

Consumers that use *enhanced fan-out* in Amazon Kinesis Data Streams can receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. This type of consumer doesn't have to contend with other consumers that are receiving data from the stream. For more information, see Develop enhanced fan-out consumers with dedicated throughput.

You can use version 2.0 or later of the Kinesis Client Library (KCL) to develop applications that use enhanced fan-out to receive data from streams. The KCL automatically subscribes your application to all the shards of a stream, and ensures that your consumer application can read with a throughput value of 2 MB/sec per shard. If you want to use the KCL without turning on enhanced fan-out, see Developing Consumers Using the Kinesis Client Library 2.0.

Topics

- Develop enhanced fan-out consumers using KCL 2.x in Java

Develop enhanced fan-out consumers using KCL 2.x in Java

Important

Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

You can use version 2.0 or later of the Kinesis Client Library (KCL) to develop applications in Amazon Kinesis Data Streams to receive data from streams using enhanced fan-out. The following code shows an example implementation in Java of `ProcessorFactory` and `RecordProcessor`.

It is recommended that you use `KinesisClientUtil` to create `KinesisAsyncClient` and to configure `maxConcurrency` in `KinesisAsyncClient`.

Important

The Amazon Kinesis Client might see significantly increased latency, unless you configure `KinesisAsyncClient` to have a `maxConcurrency` high enough to allow all leases plus additional usages of `KinesisAsyncClient`.

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */

/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument.
 The Region may be specified as the second argument.");
            System.exit(1);
        }
```

```
        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }

    private void run() {
        ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
        ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

        DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
        CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
        ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

        Scheduler scheduler = new Scheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                configsBuilder.retrievalConfig()
        );
```

```
        Thread schedulerThread = new Thread(scheduler);
        schedulerThread.setDaemon(true);
        schedulerThread.start();

        System.out.println("Press enter to shutdown");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            reader.readLine();
        } catch (IOException ioex) {
            log.error("Caught exception while waiting for confirm. Shutting down.",
ioex);
        }

        log.info("Cancelling producer, and shutting down executor.");
        producerFuture.cancel(true);
        producerExecutor.shutdownNow();

        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown. Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
```

```
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }


    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)",
processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record
pk: {} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
```

```
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Migrate consumers from KCL 1.x to KCL 2.x

> **Important**
>
> Amazon Kinesis Client Library (KCL) versions 1.x and 2.x are outdated. KCL 1.x will reach end-of-support on January 30, 2026. We **strongly recommend** that you migrate your KCL applications using version 1.x to the latest KCL version before January 30, 2026. To find the latest KCL version, see Amazon Kinesis Client Library page on GitHub. For information about the latest KCL versions, see Use Kinesis Client Library. For information about migrating from KCL 1.x to KCL 3.x, see Migrating from KCL 1.x to KCL 3.x.

This topic explains the differences between versions 1.x and 2.x of the Kinesis Client Library (KCL). It also shows you how to migrate your consumer from version 1.x to version 2.x of the KCL. After you migrate your client, it will start processing records from the last checkpointed location.

Version 2.0 of the KCL introduces the following interface changes:

KCL Interface Changes

KCL 1.x Interface	KCL 2.0 Interface
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor`	`software.amazon.kinesis.processor.ShardRecordProcessor`
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory`	`software.amazon.kinesis.processor.ShardRecordProcessorFactory`
`com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware`	Folded into `software.amazon.kinesis.processor.ShardRecordProcessor`

Topics

- Migrate the record processor
- Migrate the record processor factory
- Migrate the worker

- Configure the Amazon Kinesis client
- Idle time removal
- Client configuration removals

Migrate the record processor

The following example shows a record processor implemented for KCL 1.x:

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;

public class TestRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        //
        // Process records, and possibly checkpoint
        //
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
```

```
                shutdownInput.getCheckpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
        try {
            checkpointer.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
}
```

To migrate the record processor class

1. Change the interfaces from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor` and `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificat` to `software.amazon.kinesis.processor.ShardRecordProcessor`, as follows:

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// public class TestRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
public class TestRecordProcessor implements ShardRecordProcessor {
```

2. Update the `import` statements for the `initialize` and `processRecords` methods.

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
```

```
//import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
```

3. Replace the `shutdown` method with the following new methods: `leaseLost`, `shardEnded`, and `shutdownRequested`.

```
//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shardEnded(...)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shutdownRequested(ShutdownReauestedInput)
```

```
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
```

The following is the updated version of the record processor class.

```
package com.amazonaws.kcl;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public class TestRecordProcessor implements ShardRecordProcessor {
    @Override
    public void initialize(InitializationInput initializationInput) {

    }
```

```
    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {

    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

Migrate the record processor factory

The record processor factory is responsible for creating record processors when a lease is acquired. The following is an example of a KCL 1.x factory.

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

```
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class TestRecordProcessorFactory implements IRecordProcessorFactory {
    @Override
    public IRecordProcessor createProcessor() {
        return new TestRecordProcessor();
    }
}
```

To migrate the record processor factory

1. Change the implemented interface from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFa` to `software.amazon.kinesis.processor.ShardRecordProcessorFactory`, as follows.

   ```
   // import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
   import software.amazon.kinesis.processor.ShardRecordProcessor;

   // import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
   import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

   // public class TestRecordProcessorFactory implements IRecordProcessorFactory {
   public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
   ```

2. Change the return signature for `createProcessor`.

   ```
   // public IRecordProcessor createProcessor() {
   public ShardRecordProcessor shardRecordProcessor() {
   ```

The following is an example of the record processor factory in 2.0:

```
package com.amazonaws.kcl;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
```

```
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new TestRecordProcessor();
    }
}
```

Migrate the worker

In version 2.0 of the KCL, a new class, called `Scheduler`, replaces the `Worker` class. The following is an example of a KCL 1.x worker.

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

To migrate the worker

1. Change the `import` statement for the `Worker` class to the import statements for the `Scheduler` and `ConfigsBuilder` classes.

   ```
   // import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
   import software.amazon.kinesis.coordinator.Scheduler;
   import software.amazon.kinesis.common.ConfigsBuilder;
   ```

2. Create the `ConfigsBuilder` and a `Scheduler` as shown in the following example.

   It is recommended that you use `KinesisClientUtil` to create `KinesisAsyncClient` and to configure `maxConcurrency` in `KinesisAsyncClient`.

   > **Important**
   >
   > The Amazon Kinesis Client might see significantly increased latency, unless you configure `KinesisAsyncClient` to have a `maxConcurrency` high enough to allow all leases plus additional usages of `KinesisAsyncClient`.

```
import java.util.UUID;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;

...

Region region = Region.AP_NORTHEAST_2;
KinesisAsyncClient kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(region));
DynamoDbAsyncClient dynamoClient =
 DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();

ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, applicationName,
 kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
 SampleRecordProcessorFactory());

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
    );
```

Configure the Amazon Kinesis client

With the 2.0 release of the Kinesis Client Library, the configuration of the client moved from a single configuration class (`KinesisClientLibConfiguration`) to six configuration classes. The following table describes the migration.

Configuration Fields and Their New Classes

Original Field	New Configuration Class	Description
`applicationName`	`ConfigsBuilder`	The name for this the KCL application. Used as the default for the `tableName` and `consumerName` .
`tableName`	`ConfigsBuilder`	Allows overriding the table name used for the Amazon DynamoDB lease table.
`streamName`	`ConfigsBuilder`	The name of the stream that this application processes records from.
`kinesisEndpoint`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`dynamoDBEndpoint`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`initialPositionInStreamExtended`	`RetrievalConfig`	The location in the shard from which the KCL begins fetching records, starting with the application's initial run.
`kinesisCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`dynamoDBCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`cloudWatchCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`failoverTimeMillis`	LeaseManagementConfig	The number of milliseconds that must pass before you can consider a lease owner to have failed.

Original Field	New Configuration Class	Description
workerIdentifier	ConfigsBuilder	A unique identifier that represents this instantiation of the application processor. This must be unique.
shardSyncIntervalMillis	LeaseManagementConfig	The time between shard sync calls.
maxRecords	PollingConfig	Allows setting the maximum number of records that Kinesis returns.
idleTimeBetweenReadsInMillis	CoordinatorConfig	This option has been removed. See Idle Time Removal.
callProcessRecordsEvenForEmptyRecordList	ProcessorConfig	When set, the record processor is called even when no records were provided from Kinesis.
parentShardPollIntervalMillis	CoordinatorConfig	How often a record processor should poll to see if the parent shard has been completed.
cleanupLeasesUponShardCompletion	LeaseManagementConfig	When set, leases are removed as soon as the child leases have started processing.
ignoreUnexpectedChildShards	LeaseManagementConfig	When set, child shards that have an open shard are ignored. This is primarily for DynamoDB Streams.
kinesisClientConfig	ConfigsBuilder	This option has been removed. See Client Configuration Removals.

Original Field	New Configuration Class	Description
dynamoDBClientConfig	ConfigsBuilder	This option has been removed. See Client Configuration Removals.
cloudWatchClientConfig	ConfigsBuilder	This option has been removed. See Client Configuration Removals.
taskBackoffTimeMillis	LifecycleConfig	The time to wait to retry failed tasks.
metricsBufferTimeMillis	MetricsConfig	Controls CloudWatch metric publishing.
metricsMaxQueueSize	MetricsConfig	Controls CloudWatch metric publishing.
metricsLevel	MetricsConfig	Controls CloudWatch metric publishing.
metricsEnabledDimensions	MetricsConfig	Controls CloudWatch metric publishing.
validateSequenceNumberBeforeCheckpointing	CheckpointConfig	This option has been removed. See Checkpoint Sequence Number Validation.
regionName	ConfigsBuilder	This option has been removed. See Client Configuration Removal.
maxLeasesForWorker	LeaseManagementConfig	The maximum number of leases a single instance of the application should accept.

Original Field	New Configuration Class	Description
maxLeasesToStealAtOneTime	LeaseManagementConfig	The maximum number of leases an application should attempt to steal at one time.
initialLeaseTableReadCapacity	LeaseManagementConfig	The DynamoDB read IOPs that is used if the Kinesis Client Library needs to create a new DynamoDB lease table.
initialLeaseTableWriteCapacity	LeaseManagementConfig	The DynamoDB read IOPs that is used if the Kinesis Client Library needs to create a new DynamoDB lease table.
initialPositionInStreamExtended	LeaseManagementConfig	The initial position in the stream that the application should start at. This is only used during initial lease creation.
skipShardSyncAtWorkerInitializationIfLeasesExist	CoordinatorConfig	Disable synchronizing shard data if the lease table contains existing leases. TODO: KinesisEco-438
shardPrioritization	CoordinatorConfig	Which shard prioritization to use.
shutdownGraceMillis	N/A	This option has been removed. See MultiLang Removals.
timeoutInSeconds	N/A	This option has been removed. See MultiLang Removals.
retryGetRecordsInSeconds	PollingConfig	Configures the delay between GetRecords attempts for failures.

Original Field	New Configuration Class	Description
`maxGetRecordsThreadPool`	`PollingConfig`	The thread pool size used for GetRecords.
`maxLeaseRenewalThreads`	`LeaseManagementConfig`	Controls the size of the lease renewer thread pool. The more leases that your application could take, the larger this pool should be.
`recordsFetcherFactory`	`PollingConfig`	Allows replacing the factory used to create fetchers that retrieve from streams.
`logWarningForTaskAfterMillis`	`LifecycleConfig`	How long to wait before a warning is logged if a task hasn't completed.
`listShardsBackoffTimeInMillis`	`RetrievalConfig`	The number of milliseconds to wait between calls to `ListShards` when failures occur.
`maxListShardsRetryAttempts`	`RetrievalConfig`	The maximum number of times that `ListShards` retries before giving up.

Idle time removal

In the 1.x version of the KCL, the `idleTimeBetweenReadsInMillis` corresponded to two quantities:

- The amount of time between task dispatching checks. You can now configure this time between tasks by setting `CoordinatorConfig#shardConsumerDispatchPollIntervalMillis`.
- The amount of time to sleep when no records were returned from Kinesis Data Streams. In version 2.0, in enhanced fan-out records are pushed from their respective retriever. Activity on the shard consumer only occurs when a pushed request arrives.

Client configuration removals

In version 2.0, the KCL no longer creates clients. It depends on the user to supply a valid client. With this change, all configuration parameters that controlled client creation have been removed. If you need these parameters, you can set them on the clients before providing the clients to `ConfigsBuilder`.

Removed Field	Equivalent Configuration
`kinesisEndpoint`	Configure the SDK `KinesisAsyncClient` with preferred endpoint: `KinesisAsyncClient.builder().endpointOverride(URI.create("https://<kinesis endpoint>")).build()` .
`dynamoDBEndpoint`	Configure the SDK `DynamoDbAsyncClient` with preferred endpoint: `DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://<dynamodb endpoint>")).build()` .
`kinesisClientConfig`	Configure the SDK `KinesisAsyncClient` with the needed configuration: `KinesisAsyncClient.builder().overrideConfiguration(<your configuration>).build()` .
`dynamoDBClientConfig`	Configure the SDK `DynamoDbAsyncClient` with the needed configuration: `DynamoDbAsyncClient.builder().overrideConfiguration(<your configuration>).build()` .
`cloudWatchClientConfig`	Configure the SDK `CloudWatchAsyncClient` with the needed configuration: `CloudWatchAsyncClient.builder().overrideConfiguration(<your configuration>).build()` .
`regionName`	Configure the SDK with the preferred Region. This is the same for all SDK clients. For example, `KinesisAsyncClient.builder().region(Region.US_WEST_2).build()` .

Develop consumers with the AWS SDK for Java

You can develop custom consumers using the Amazon Kinesis Data Streams APIs. This section describes using the Kinesis Data Streams APIs with the AWS SDK for Java.

Important

The recommended method for developing custom Kinesis Data Streams consumers with shared throughout is to use the Kinesis Client Library (KCL). KCL helps you consume and process data from a Kinesis data stream by taking care of many of the complex tasks associated with distributed computing. For more information, see Develop consumers with KCL in Java.

Topics

- Develop shared-throughput consumers with the AWS SDK for Java
- Develop enhanced fan-out consumers with the AWS SDK for Java
- Interact with data using the AWS Glue Schema Registry

Develop shared-throughput consumers with the AWS SDK for Java

One of the methods for developing custom Kinesis Data Streams consumers with shared throughout is to use the Amazon Kinesis Data Streams APIs with the AWS SDK for Java. This section describes using the Kinesis Data Streams APIs with the AWS SDK for Java. You can call the Kinesis Data Streams APIs using other different programming languages. For more information about all available AWS SDKs, see Start Developing with Amazon Web Services.

The Java sample code in this section demonstrates how to perform basic Kinesis Data Streams API operations, and is divided up logically by operation type. These examples don't represent production-ready code. They don't check for all possible exceptions or account for all possible security or performance considerations.

Topics

- Get data from a stream
- Use shard iterators
- Use GetRecords
- Adapt to a reshard

Get data from a stream

The Kinesis Data Streams APIs include the `getShardIterator` and `getRecords` methods that you can invoke to retrieve records from a data stream. This is the pull model, where your code draws data records directly from the shards of the data stream.

> **Important**
>
> We recommend that you use the record processor support provided by KCL to retrieve records from your data streams. This is the push model, where you implement the code that processes the data. The KCL retrieves data records from the data stream and delivers them to your application code. In addition, the KCL provides failover, recovery, and load balancing functionality. For more information, see Developing Custom Consumers with Shared Throughput Using KCL.

However, in some cases you might prefer to use the Kinesis Data Streams APIs. For example, to implement custom tools for monitoring or debugging your data streams.

> **Important**
>
> Kinesis Data Streams supports changes to the data record retention period of your data stream. For more information, see Change the data retention period.

Use shard iterators

You retrieve records from the stream on a per-shard basis. For each shard, and for each batch of records that you retrieve from that shard, you must obtain a *shard iterator*. The shard iterator is used in the `getRecordsRequest` object to specify the shard from which records are to be retrieved. The type associated with the shard iterator determines the point in the shard from which the records should be retrieved (see later in this section for more details). Before you can work with the shard iterator, you must retrieve the shard. For more information, see List shards.

Obtain the initial shard iterator using the `getShardIterator` method. Obtain shard iterators for additional batches of records using the `getNextShardIterator` method of the `getRecordsResult` object returned by the `getRecords` method. A shard iterator is valid for 5 minutes. If you use a shard iterator while it is valid, you get a new one. Each shard iterator remains valid for 5 minutes, even after it is used.

To obtain the initial shard iterator, instantiate `GetShardIteratorRequest` and pass it to the `getShardIterator` method. To configure the request, specify the stream and the shard ID. For information about how to obtain the streams in your AWS account, see List streams. For information about how to obtain the shards in a stream, see List shards.

```
String shardIterator;
GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest();
getShardIteratorRequest.setStreamName(myStreamName);
getShardIteratorRequest.setShardId(shard.getShardId());
getShardIteratorRequest.setShardIteratorType("TRIM_HORIZON");

GetShardIteratorResult getShardIteratorResult =
 client.getShardIterator(getShardIteratorRequest);
shardIterator = getShardIteratorResult.getShardIterator();
```

This sample code specifies `TRIM_HORIZON` as the iterator type when obtaining the initial shard iterator. This iterator type means that records should be returned beginning with the first record added to the shard—rather than beginning with the most recently added record, also known as the *tip*. The following are possible iterator types:

- `AT_SEQUENCE_NUMBER`
- `AFTER_SEQUENCE_NUMBER`
- `AT_TIMESTAMP`
- `TRIM_HORIZON`
- `LATEST`

For more information, see ShardIteratorType.

Some iterator types require that you specify a sequence number in addition to the type; for example:

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

After you obtain a record using `getRecords`, you can get the sequence number for the record by calling the record's `getSequenceNumber` method.

```
record.getSequenceNumber()
```

In addition, the code that adds records to the data stream can get the sequence number for an added record by calling `getSequenceNumber` on the result of `putRecord`.

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

You can use sequence numbers to guarantee strictly increasing ordering of records. For more information, see the code example in PutRecord example.

Use GetRecords

After you obtain the shard iterator, instantiate a `GetRecordsRequest` object. Specify the iterator for the request using the `setShardIterator` method.

Optionally, you can also set the number of records to retrieve using the `setLimit` method. The number of records returned by `getRecords` is always equal to or less than this limit. If you do not specify this limit, `getRecords` returns 10 MB of retrieved records. The sample code below sets this limit to 25 records.

If no records are returned, that means no data records are currently available from this shard at the sequence number referenced by the shard iterator. In this situation, your application should wait for an amount of time that's appropriate for the data sources for the stream. Then try to get data from the shard again using the shard iterator returned by the preceding call to `getRecords`.

Pass the `getRecordsRequest` to the `getRecords` method, and capture the returned value as a `getRecordsResult` object. To get the data records, call the `getRecords` method on the `getRecordsResult` object.

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(25);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

To prepare for another call to `getRecords`, obtain the next shard iterator from `getRecordsResult`.

```
shardIterator = getRecordsResult.getNextShardIterator();
```

For best results, sleep for at least 1 second (1,000 milliseconds) between calls to `getRecords` to avoid exceeding the limit on `getRecords` frequency.

```
try {
  Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

Typically, you should call `getRecords` in a loop, even when you're retrieving a single record in a test scenario. A single call to `getRecords` might return an empty record list, even when the shard contains more records at later sequence numbers. When this occurs, the `NextShardIterator` returned along with the empty record list references a later sequence number in the shard, and successive `getRecords` calls eventually returns the records. The following sample demonstrates the use of a loop.

Example: getRecords

The following code example reflects the `getRecords` tips in this section, including making calls in a loop.

```
// Continuously read data records from a shard
List<Record> records;

while (true) {

  // Create a new getRecordsRequest with an existing shardIterator
  // Set the maximum records to return to 25

  GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
  getRecordsRequest.setShardIterator(shardIterator);
  getRecordsRequest.setLimit(25);

  GetRecordsResult result = client.getRecords(getRecordsRequest);

  // Put the result into record list. The result can be empty.
  records = result.getRecords();
```

```
  try {
    Thread.sleep(1000);
  }
  catch (InterruptedException exception) {
    throw new RuntimeException(exception);
  }

  shardIterator = result.getNextShardIterator();
}
```

If you are using the Kinesis Client Library, it might make multiple calls before returning data. This behavior is by design and does not indicate a problem with the KCL or your data.

Adapt to a reshard

If `getRecordsResult.getNextShardIterator` returns `null`, it indicates that a shard split or merge has occurred that involved this shard. This shard is now in a `CLOSED` state and you have read all available data records from this shard.

In this scenario, you can use `getRecordsResult.childShards` to learn about the new child shards of the shard that is being processed that were created by the split or merge. For more information, see ChildShard.

In the case of a split, the two new shards both have `parentShardId` equal to the shard ID of the shard that you were processing previously. The value of `adjacentParentShardId` for both of these shards is `null`.

In the case of a merge, the single new shard created by the merge has `parentShardId` equal to shard ID of one of the parent shards and `adjacentParentShardId` equal to the shard ID of the other parent shard. Your application has already read all the data from one of these shards. This is the shard for which `getRecordsResult.getNextShardIterator` returned `null`. If the order of the data is important to your application, ensure that it also reads all the data from the other parent shard before reading any new data from the child shard created by the merge.

If you are using multiple processors to retrieve data from the stream (say, one processor per shard), and a shard split or merge occurs, adjust the number of processors up or down to adapt to the change in the number of shards.

For more information about resharding, including a discussion of shards states—such as `CLOSED`—see Reshard a stream.

Develop enhanced fan-out consumers with the AWS SDK for Java

Enhanced fan-out is an Amazon Kinesis Data Streams feature that enables consumers to receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. A consumer that uses enhanced fan-out doesn't have to contend with other consumers that are receiving data from the stream. For more information, see Develop enhanced fan-out consumers with dedicated throughput.

You can use API operations to build a consumer that uses enhanced fan-out in Kinesis Data Streams.

To register a consumer with enhanced fan-out using the Kinesis Data Streams API

1. Call RegisterStreamConsumer to register your application as a consumer that uses enhanced fan-out. Kinesis Data Streams generates an Amazon Resource Name (ARN) for the consumer and returns it in the response.
2. To start listening to a specific shard, pass the consumer ARN in a call to SubscribeToShard. Kinesis Data Streams then starts pushing the records from that shard to you, in the form of events of type SubscribeToShardEvent over an HTTP/2 connection. The connection remains open for up to 5 minutes. Call SubscribeToShard again if you want to continue receiving records from the shard after the `future` that is returned by the call to SubscribeToShard completes normally or exceptionally.

   > **Note**
   >
   > `SubscribeToShard` API also returns the list of the child shards of the current shard when the end of the current shard is reached.

3. To deregister a consumer that is using enhanced fan-out, call DeregisterStreamConsumer.

The following code is an example of how you can subscribe your consumer to a shard, renew the subscription periodically, and handle the events.

```
    import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
    import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
    import
 software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;
```

```
    import java.util.concurrent.CompletableFuture;

    /**
     * See https://github.com/awsdocs/aws-doc-sdk-examples/blob/master/javav2/
example_code/kinesis/src/main/java/com/example/kinesis/KinesisStreamEx.java
     * for complete code and more examples.
     */
    public class SubscribeToShardSimpleImpl {

        private static final String CONSUMER_ARN = "arn:aws:kinesis:us-
east-1:123456789123:stream/foobar/consumer/test-consumer:1525898737";
        private static final String SHARD_ID = "shardId-000000000000";

        public static void main(String[] args) {

            KinesisAsyncClient client = KinesisAsyncClient.create();

            SubscribeToShardRequest request = SubscribeToShardRequest.builder()
                    .consumerARN(CONSUMER_ARN)
                    .shardId(SHARD_ID)
                    .startingPosition(s -> s.type(ShardIteratorType.LATEST)).build();

            // Call SubscribeToShard iteratively to renew the subscription
 periodically.
            while(true) {
                // Wait for the CompletableFuture to complete normally or
 exceptionally.
                callSubscribeToShardWithVisitor(client, request).join();
            }

            // Close the connection before exiting.
            // client.close();
        }


        /**
         * Subscribes to the stream of events by implementing the
 SubscribeToShardResponseHandler.Visitor interface.
         */
        private static CompletableFuture<Void>
 callSubscribeToShardWithVisitor(KinesisAsyncClient client, SubscribeToShardRequest
 request) {
```

```
            SubscribeToShardResponseHandler.Visitor visitor = new
SubscribeToShardResponseHandler.Visitor() {
                @Override
                public void visit(SubscribeToShardEvent event) {
                    System.out.println("Received subscribe to shard event " + event);
                }
            };
            SubscribeToShardResponseHandler responseHandler =
SubscribeToShardResponseHandler
                    .builder()
                    .onError(t -> System.err.println("Error during stream - " +
t.getMessage()))
                    .subscriber(visitor)
                    .build();
            return client.subscribeToShard(request, responseHandler);
        }
    }
```

If `event.ContinuationSequenceNumber` returns `null`, it indicates that a shard split or merge has occurred that involved this shard. This shard is now in a `CLOSED` state, and you have read all available data records from this shard. In this scenario, per example above, you can use `event.childShards` to learn about the new child shards of the shard that is being processed that were created by the split or merge. For more information, see ChildShard.

Interact with data using the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue Schema Registry. The AWS Glue Schema Registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the `GetRecords` Kinesis Data Streams API available in the AWS Java SDK.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the `GetRecords` Kinesis Data Streams APIs, see the "Interacting with Data Using the Kinesis Data Streams APIs" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Develop consumers using AWS Lambda

You can use an AWS Lambda function to process records in a data stream. AWS Lambda is a compute service that lets you run code without provisioning or managing servers. It executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume. There is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service, all with zero administration. It runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. For more information, see Using AWS Lambda with Amazon Kinesis.

For troubleshooting information, see Why is Kinesis Data Streams trigger unable to invoke my Lambda function?

Develop consumers using Amazon Managed Service for Apache Flink

You can use an Amazon Managed Service for Apache Flink application to process and analyze data in a Kinesis stream using SQL, Java, or Scala. Managed Service for Apache Flink applications can enrich data using reference sources, aggregate data over time, or use machine learning to find data anomalies. Then you can write the analysis results to another Kinesis stream, a Firehose delivery stream, or a Lambda function. For more information, see the Managed Service for Apache Flink Developer Guide for SQL Applications or the Managed Service for Apache Flink Developer Guide for Flink Applications.

Develop consumers using Amazon Data Firehose

You can use a Firehose to read and process records from a Kinesis stream. Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, Amazon OpenSearch Service, and Splunk. Firehose also supports any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers, including Datadog, MongoDB, and New Relic. You can also configure Firehose to transform your data records and to convert the record format before delivering your data to its destination. For more information, see Writing to Firehose Using Kinesis Data Streams.

Read data from Kinesis Data Streams using other AWS services

The following AWS services can directly integrate with Amazon Kinesis Data Streams to read data from Kinesis data streams. Review the information for each service that you are interested in and refer to the provided references.

Topics

- Read data from Kinesis Data Streams using Amazon EMR
- Read data from Kinesis Data Streams using Amazon EventBridge Pipes
- Read data from Kinesis Data Streams using AWS Glue
- Read data from Kinesis Data Streams using Amazon Redshift

Read data from Kinesis Data Streams using Amazon EMR

Amazon EMR clusters can read and process Kinesis streams directly, using familiar tools in the Hadoop ecosystem such as Hive, Pig, MapReduce, the Hadoop Streaming API, and Cascading. You can also join real-time data from Kinesis Data Streams with existing data on Amazon S3, Amazon DynamoDB, and HDFS in a running cluster. You can directly load the data from Amazon EMR to Amazon S3 or DynamoDB for post-processing activities.

For more information, see Amazon Kinesis in the *Amazon EMR Release Guide*.

Read data from Kinesis Data Streams using Amazon EventBridge Pipes

Amazon EventBridge Pipes supports Kinesis data streams as a source. Amazon EventBridge Pipes helps you create point-to-point integrations between event producers and consumers with optional transform, filter and enrich steps. You can use EventBridge Pipes to receive records in a Kinesis data stream and optionally filter or enhance these records before sending them to one of the available destinations for processing, including Kinesis Data Streams.

For more information, see Amazon Kinesis stream as a source in the *Amazon EventBridge Release Guide*.

Read data from Kinesis Data Streams using AWS Glue

Using AWS Glue streaming ETL, you can create streaming extract, transform, and load (ETL) jobs that run continuously and consume data from Amazon Kinesis Data Streams. The jobs cleanse and transform the data, and then load the results into Amazon S3 data lakes or JDBC data stores.

For more information, see Streaming ETL jobs in AWS Glue in the *AWS Glue Release Guide*.

Read data from Kinesis Data Streams using Amazon Redshift

Amazon Redshift supports streaming ingestion from Amazon Kinesis Data Streams. The Amazon Redshift streaming ingestion feature provides low-latency, high-speed ingestion of streaming data from Amazon Kinesis Data Streams into an Amazon Redshift materialized view. Amazon Redshift streaming ingestion removes the need to stage data in Amazon S3before ingesting into Amazon Redshift.

For more information, see Streaming ingestion in the *Amazon Redshift Release Guide*.

Read from Kinesis Data Streams using third-party integrations

You can read data from Amazon Kinesis Data Streams data streams using one of the following third-party options that integrate with Kinesis Data Streams. Select the option you want to learn more about and find resources and links to relevant documentation.

Topics

- Apache Flink
- Adobe Experience Platform
- Apache Druid
- Apache Spark
- Databricks
- Kafka Confluent Platform
- Kinesumer
- Talend

Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. For more information on consuming Kinesis Data Streams using Apache Flink, see Amazon Kinesis Data Streams Connector.

Adobe Experience Platform

Adobe Experience Platform enables organizations to centralize and standardize customer data from any system. It then applies data science and machine learning to dramatically improve the design and delivery of rich, personalized experiences. For more information on consuming Kinesis data streams using the Adobe Experience Platform, see Amazon Kinesis connector.

Apache Druid

Druid is a high performance, real-time analytics database that delivers sub-second queries on streaming and batch data at scale and under load. For more information on ingesting Kinesis data streams using Apache Druid, see Amazon Kinesis ingestion.

Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. You can use Apache Spark to build stream processing applications that consume the data in your Kinesis data streams.

To consume Kinesis data streams using Apache Spark Structured Streaming, use the Amazon Kinesis Data Streams connector. This connector supports consumption with Enhanced Fan-Out, which provides your application with dedicated read throughput of up to 2 MB of data per second per shard. For more information, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out).

To consume Kinesis data streams using Spark Streaming, see Spark Streaming + Kinesis Integration.

Databricks

Databricks is a cloud-based platform that provides a collaborative environment for data engineering, data science, and machine learning. For more information on consuming Kinesis data streams using Databricks, see Connect to Amazon Kinesis.

Kafka Confluent Platform

Confluent Platform is built on top of Kafka and provides additional features and functionality that help enterprises build and manage real-time data pipelines and streaming applications. For more

information on consuming Kinesis data streams using the Confluent Platform, see Amazon Kinesis Source Connector for Confluent Platform.

Kinesumer

Kinesumer is a Go client implementing a client-side distributed consumer group client for Kinesis data streams. For more information, see Kinesumer Github repository.

Talend

Talend is a data integration and management software that allows users to collect, transform, and connect data from various sources in a scalable and efficient manner. For more information on consuming Kinesis data streams using Talend, see Connect talend to an Amazon Kinesis stream.

Troubleshoot Kinesis Data Streams consumers

The following topics offer solutions to common issues with Amazon Kinesis Data Streams consumers:

- Compilation error with the LeaseManagementConfig constructor
- Some Kinesis Data Streams records are skipped when using the Kinesis Client Library
- Records belonging to the same shard are processed by different record processors at the same time
- The consumer application is reading at a slower rate than expected
- GetRecords returns an empty records array even when there is data in the stream
- The shard iterator expires unexpectedly
- Consumer record processing is falling behind
- Unauthorized KMS key permission error
- DynamoDbException: The document path provided in the update expression is invalid for update
- Troubleshoot other common issues for consumers

Compilation error with the LeaseManagementConfig constructor

When upgrading to Kinesis Client Library (KCL) version 3.x or later, you may encounter a compilation error related to the `LeaseManagementConfig` constructor. If you are directly creating a `LeaseManagementConfig` object to set configurations instead of using `ConfigsBuilder` in

KCL versions 3.x or later, you might see the following error message while compiling your KCL application code.

```
Cannot resolve constructor 'LeaseManagementConfig(String, DynamoDbAsyncClient,
 KinesisAsyncClient, String)'
```

KCL with versions 3.x or later requires you to add one more parameter, applicationName (type: String), after the tableName parameter.

- *Before*: leaseManagementConfig = new LeaseManagementConfig(tableName, dynamoDBClient, kinesisClient, streamName, workerIdentifier)
- *After*: leaseManagementConfig = new LeaseManagementConfig(tableName, **applicationName**, dynamoDBClient, kinesisClient, streamName, workerIdentifier)

Instead of directly creating a LeaseManagementConfig object, we recommend using `ConfigsBuilder` to set configurations in KCL 3.x and later versions. `ConfigsBuilder` provides a more flexible and maintainable way to configure your KCL application.

The following is an example of using `ConfigsBuilder` to set KCL configurations.

```
ConfigsBuilder configsBuilder = new ConfigsBuilder(
    streamName,
    applicationName,
    kinesisClient,
    dynamoClient,
    cloudWatchClient,
    UUID.randomUUID().toString(),
    new SampleRecordProcessorFactory()
);

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig()
    .failoverTimeMillis(60000), // this is an example
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
);
```

Some Kinesis Data Streams records are skipped when using the Kinesis Client Library

The most common cause of skipped records is an unhandled exception thrown from `processRecords`. The Kinesis Client Library (KCL) relies on your `processRecords` code to handle any exceptions that arise from processing the data records. Any exception thrown from `processRecords` is absorbed by the KCL. To avoid infinite retries on a recurring failure, the KCL does not resend the batch of records processed at the time of the exception. The KCL then calls `processRecords` for the next batch of data records without restarting the record processor. This effectively results in consumer applications observing skipped records. To prevent skipped records, handle all exceptions within `processRecords` appropriately.

Records belonging to the same shard are processed by different record processors at the same time

For any running Kinesis Client Library (KCL) application, a shard only has one owner. However, multiple record processors may temporarily process the same shard. If a worker instance loses network connectivity, the KCL assumes that the unreachable worker is no longer processing records after the failover time expires, and directs other worker instances to take over. For a brief period, new record processors and record processors from the unreachable worker may process data from the same shard.

Set a failover time that is appropriate for your application. For low-latency applications, the 10-second default may represent the maximum time you want to wait. However, in cases where you expect connectivity issues such as making calls across geographical areas where connectivity could be lost more frequently, this number may be too low.

Your application should anticipate and handle this scenario, especially because network connectivity is usually restored to the previously unreachable worker. If a record processor has its shards taken by another record processor, it must handle the following two cases to perform graceful shutdown:

1. After the current call to `processRecords` is completed, the KCL invokes the shutdown method on the record processor with shutdown reason 'ZOMBIE.' Your record processors are expected to clean up any resources as appropriate and then exit.
2. When you attempt to checkpoint from a 'zombie' worker, the KCL throws `ShutdownException`. After receiving this exception, your code is expected to exit the current method cleanly.

For more information, see Handle duplicate records.

The consumer application is reading at a slower rate than expected

The most common reasons for read throughput being slower than expected are as follows:

1. Multiple consumer applications have total reads exceeding the per-shard limits. For more information, see Quotas and limits. In this case, increase the number of shards in the Kinesis data stream.
2. The limit that specifies the maximum number of **GetRecords** per call may have been configured with a low value. If you are using the KCL, you may have configured the worker with a low value for the `maxRecords` property. In general, we recommend using the system defaults for this property.
3. The logic inside your `processRecords` call may be taking longer than expected for a number of possible reasons; the logic may be CPU intensive, I/O blocking, or bottlenecked on synchronization. To test if this is true, test run empty record processors and compare the read throughput. For information about how to keep up with the incoming data, see Use resharding, scaling, and parallel processing to change the number of shards.

If you have only one consumer application, it is always possible to read at least two times faster than the put rate. That's because you can write up to 1,000 records per second for writes, up to a maximum total data write rate of 1 MB per second (including partition keys). Each open shard can support up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second. Note that each read (**GetRecords** call) gets a batch of records. The size of the data returned by **GetRecords** varies depending on the utilization of the shard. The maximum size of data that **GetRecords** can return is 10 MB. If a call returns that limit, subsequent calls made within the next 5 seconds throw a `ProvisionedThroughputExceededException`.

GetRecords returns an empty records array even when there is data in the stream

Consuming, or getting records is a pull model. Developers are expected to call GetRecords in a continuous loop with no back-offs. Every call to **GetRecords** also returns a `ShardIterator` value, which must be used in the next iteration of the loop.

The **GetRecords** operation does not block. Instead, it returns immediately; with either relevant data records or with an empty `Records` element. An empty `Records` element is returned under two conditions:

1. There is no more data currently in the shard.
2. There is no data near the part of the shard pointed to by the `ShardIterator`.

The latter condition is subtle, but is a necessary design tradeoff to avoid unbounded seek time (latency) when retrieving records. Thus, the stream-consuming application should loop and call **GetRecords**, handling empty records as a matter of course.

In a production scenario, the only time the continuous loop should be exited is when the `NextShardIterator` value is `NULL`. When `NextShardIterator` is `NULL`, it means that the current shard has been closed and the `ShardIterator`value would otherwise point past the last record. If the consuming application never calls **SplitShard** or **MergeShards**, the shard remains open and the calls to **GetRecords** never return a `NextShardIterator` value that is `NULL`.

If you use the Kinesis Client Library (KCL), the preceding consumption pattern is abstracted for you. This includes automatic handling of a set of shards that dynamically change. With the KCL, the developer only supplies the logic to process incoming records. This is possible because the library makes continuous calls to **GetRecords** for you.

The shard iterator expires unexpectedly

A new shard iterator is returned by every **GetRecords** request (as `NextShardIterator`), which you then use in the next **GetRecords** request (as `ShardIterator`). Typically, this shard iterator does not expire before you use it. However, you may find that shard iterators expire because you have not called **GetRecords** for more than 5 minutes, or because you've performed a restart of your consumer application.

If the shard iterator expires immediately before you can use it, this might indicate that the DynamoDB table used by Kinesis does not have enough capacity to store the lease data. This situation is more likely to happen if you have a large number of shards. To solve this problem, increase the write capacity assigned to the shard table. For more information, see Use a lease table to track the shards processed by the KCL consumer application.

Consumer record processing is falling behind

For most use cases, consumer applications are reading the latest data from the stream. In certain circumstances, consumer reads may fall behind, which may not be desired. After you identify how far behind your consumers are reading, look at the most common reasons why consumers fall behind.

Start with the `GetRecords.IteratorAgeMilliseconds` metric, which tracks the read position across all shards and consumers in the stream. Note that if an iterator's age passes 50% of the retention period (by default, 24 hours, configurable up to 365 days), there is risk for data loss due to record expiration. A quick stopgap solution is to increase the retention period. This stops the loss of important data while you troubleshoot the issue further. For more information, see Monitor the Amazon Kinesis Data Streams service with Amazon CloudWatch. Next, identify how far behind your consumer application is reading from each shard using a custom CloudWatch metric emitted by the Kinesis Client Library (KCL), `MillisBehindLatest`. For more information, see Monitor the Kinesis Client Library with Amazon CloudWatch.

Here are the most common reasons consumers can fall behind:

- Sudden large increases to `GetRecords.IteratorAgeMilliseconds` or `MillisBehindLatest` usually indicate a transient problem, such as API operation failures to a downstream application. Investigate these sudden increases if either of the metrics consistently display this behavior.
- A gradual increase to these metrics indicates that a consumer is not keeping up with the stream because it is not processing records fast enough. The most common root causes for this behavior are insufficient physical resources or record processing logic that has not scaled with an increase in stream throughput. You can verify this behavior by looking at the other custom CloudWatch metrics that the KCL emits associated with the `processTask` operation, including `RecordProcessor.processRecords.Time`, `Success`, and `RecordsProcessed`.
  - If you see an increase in the `processRecords.Time` metric that correlates with increased throughput, you should analyze your record processing logic to identify why it is not scaling with the increased throughput.
  - If you see an increase to the `processRecords.Time` values that are not correlated with increased throughput, check to see if you are making any blocking calls in the critical path, which are often the cause of slowdowns in record processing. An alternative approach is to increase your parallelism by increasing the number of shards. Finally, confirm that you have

an adequate amount of physical resources (memory, CPU utilization, among others) on the underlying processing nodes during peak demand.

Unauthorized KMS key permission error

This error occurs when a consumer application reads from an encrypted stream without permissions on the AWS KMS key. To assign permissions to an application to access a KMS key, see Using Key Policies in AWS KMS and Using IAM Policies with AWS KMS.

DynamoDbException: The document path provided in the update expression is invalid for update

When using KCL 3.x with AWS SDK for Java versions 2.27.19 through 2.27.23, you may encounter the following DynamoDB exception:

"software.amazon.awssdk.services.dynamodb.model.DynamoDbException: The document path provided in the update expression is invalid for update (Service: DynamoDb, Status Code: 400, Request ID: xxx)"

This error occurs due to a known issue in the AWS SDK for Java that affects the DynamoDB metadata table managed by KCL 3.x. The issue was introduced in version 2.27.19 and impacts all versions up to 2.27.23. The issue has been resolved in the AWS SDK for Java version 2.27.24. For optimal performance and stability, we recommend upgrading to version 2.28.0 or later.

Troubleshoot other common issues for consumers

- Why is Kinesis Data Streams trigger unable to invoke my Lambda function?
- How do I detect and troubleshoot ReadProvisionedThroughputExceeded exceptions in Kinesis Data Streams?
- Why am I experiencing high latency issues with Kinesis Data Streams?
- Why is my Kinesis data stream returning a 500 Internal Server Error?
- How do I troubleshoot a blocked or stuck KCL application for Kinesis Data Streams?
- Can I use different Amazon Kinesis Client Library applications with the same Amazon DynamoDB table?

Optimize Amazon Kinesis Data Streams consumers

You can further optimize your Amazon Kinesis Data Streams consumer based on specific behavior you see.

Review the following topics to identify solutions.

Topics

- Improve low-latency processing
- Process serialized data using AWS Lambda with the Amazon Kinesis Producer Library
- Use resharding, scaling, and parallel processing to change the number of shards
- Handle duplicate records
- Handle startup, shutdown, and throttling

Improve low-latency processing

Propagation delay is defined as the end-to-end latency from the moment a record is written to the stream until it is read by a consumer application. This delay varies depending upon a number of factors, but it is primarily affected by the polling interval of consumer applications.

For most applications, we recommend polling each shard one time per second per application. This enables you to have multiple consumer applications processing a stream concurrently without hitting Amazon Kinesis Data Streams limits of 5 `GetRecords` calls per second. Additionally, processing larger batches of data tends to be more efficient at reducing network and other downstream latencies in your application.

The KCL defaults are set to follow the best practice of polling every 1 second. This default results in average propagation delays that are typically below 1 second.

Kinesis Data Streams records are available to be read immediately after they are written. There are some use cases that need to take advantage of this and require consuming data from the stream as soon as it is available. You can significantly reduce the propagation delay by overriding the KCL default settings to poll more frequently, as shown in the following examples.

Java KCL configuration code:

```
kinesisClientLibConfiguration = new
        KinesisClientLibConfiguration(applicationName,
```

```
        streamName,
        credentialsProvider,

 workerId).withInitialPositionInStream(initialPositionInStream).withIdleTimeBetweenReadsInMilli
```

Property file setting for Python and Ruby KCL:

```
idleTimeBetweenReadsInMillis = 250
```

> **Note**
>
> Because Kinesis Data Streams has a limit of 5 `GetRecords` calls per second, per shard, setting the `idleTimeBetweenReadsInMillis` property lower than 200ms may result in your application observing the `ProvisionedThroughputExceededException` exception. Too many of these exceptions can result in exponential back-offs and thereby cause significant unexpected latencies in processing. If you set this property to be at or just above 200 ms and have more than one processing application, you will experience similar throttling.

Process serialized data using AWS Lambda with the Amazon Kinesis Producer Library

The Amazon Kinesis Producer Library (KPL) aggregates small user-formatted records into larger records up to 1 MB to make better use of Amazon Kinesis Data Streams throughput. While the KCL for Java supports deaggregating these records, you need to use a special module to deaggregate records when using AWS Lambda as the consumer of your streams. You can obtain the necessary project code and instructions from GitHub at Amazon Kinesis Producer Library Deaggregation Modules for AWS Lambda. The components in this project give you the ability to process KPL serialized data within AWS Lambda, in Java, Node.js and Python. These components can also be used as part of a multi-lang KCL application.

Use resharding, scaling, and parallel processing to change the number of shards

Resharding enables you to increase or decrease the number of shards in a stream in order to adapt to changes in the rate of data flowing through the stream. Resharding is typically performed by

an administrative application that monitors shard data-handling metrics. Although the KCL itself doesn't initiate resharding operations, it is designed to adapt to changes in the number of shards that result from resharding.

As noted in Use a lease table to track the shards processed by the KCL consumer application, the KCL tracks the shards in the stream using an Amazon DynamoDB table. When new shards are created as a result of resharding, the KCL discovers the new shards and populates new rows in the table. The workers automatically discover the new shards and create processors to handle the data from them. The KCL also distributes the shards in the stream across all the available workers and record processors.

The KCL ensures that any data that existed in shards prior to the resharding is processed first. After that data has been processed, data from the new shards is sent to record processors. In this way, the KCL preserves the order in which data records were added to the stream for a particular partition key.

Example: Resharding, scaling, and parallel processing

The following example illustrates how the KCL helps you handle scaling and resharding:

- For example, if your application is running on one EC2 instance, and is processing one Kinesis data stream that has four shards. This one instance has one KCL worker and four record processors (one record processor for every shard). These four record processors run in parallel within the same process.
- Next, if you scale the application to use another instance, you have two instances processing one stream that has four shards. When the KCL worker starts up on the second instance, it load-balances with the first instance, so that each instance now processes two shards.
- If you then decide to split the four shards into five shards. The KCL again coordinates the processing across instances: one instance processes three shards, and the other processes two shards. A similar coordination occurs when you merge shards.

Typically, when you use the KCL, you should ensure that the number of instances does not exceed the number of shards (except for failure standby purposes). Each shard is processed by exactly one KCL worker and has exactly one corresponding record processor, so you never need multiple instances to process one shard. However, one worker can process any number of shards, so it's fine if the number of shards exceeds the number of instances.

To scale up processing in your application, you should test a combination of these approaches:

- Increasing the instance size (because all record processors run in parallel within a process)
- Increasing the number of instances up to the maximum number of open shards (because shards can be processed independently)
- Increasing the number of shards (which increases the level of parallelism)

Note that you can use Auto Scaling to automatically scale your instances based on appropriate metrics. For more information, see the Amazon EC2 Auto Scaling User Guide.

When resharding increases the number of shards in the stream, the corresponding increase in the number of record processors increases the load on the EC2 instances that are hosting them. If the instances are part of an Auto Scaling group, and the load increases sufficiently, the Auto Scaling group adds more instances to handle the increased load. You should configure your instances to launch your Amazon Kinesis Data Streams application at startup, so that additional workers and record processors become active on the new instance right away.

For more information about resharding, see Reshard a stream.

Handle duplicate records

There are two primary reasons why records may be delivered more than one time to your Amazon Kinesis Data Streams application: producer retries and consumer retries. Your application must anticipate and appropriately handle processing individual records multiple times.

Producer retries

Consider a producer that experiences a network-related timeout after it makes a call to `PutRecord`, but before it can receive an acknowledgement from Amazon Kinesis Data Streams. The producer cannot be sure if the record was delivered to Kinesis Data Streams. Assuming that every record is important to the application, the producer would have been written to retry the call with the same data. If both `PutRecord` calls on that same data were successfully committed to Kinesis Data Streams, then there will be two Kinesis Data Streams records. Although the two records have identical data, they also have unique sequence numbers. Applications that need strict guarantees should embed a primary key within the record to remove duplicates later when processing. Note that the number of duplicates due to producer retries is usually low compared to the number of duplicates due to consumer retries.

> **Note**
>
> If you use the AWS SDK `PutRecord`, learn about SDK Retry behavior in the *AWS SDKs and Tools user guide*.

Consumer retries

Consumer (data processing application) retries happen when record processors restart. Record processors for the same shard restart in the following cases:

1. A worker terminates unexpectedly
2. Worker instances are added or removed
3. Shards are merged or split
4. The application is deployed

In all these cases, the shards-to-worker-to-record-processor mapping is continuously updated to load balance processing. Shard processors that were migrated to other instances restart processing records from the last checkpoint. This results in duplicated record processing as shown in the example below. For more information about load-balancing, see Use resharding, scaling, and parallel processing to change the number of shards.

Example: Consumer retries resulting in redelivered records

In this example, you have an application that continuously reads records from a stream, aggregates records into a local file, and uploads the file to Amazon S3. For simplicity, assume there is only 1 shard and 1 worker processing the shard. Consider the following example sequence of events, assuming that the last checkpoint was at record number 10000:

1. A worker reads the next batch of records from the shard, records 10001 to 20000.
2. The worker then passes the batch of records to the associated record processor.
3. The record processor aggregates the data, creates an Amazon S3 file, and uploads the file to Amazon S3 successfully.
4. Worker terminates unexpectedly before a new checkpoint can occur.
5. Application, worker, and record processor restart.
6. Worker now begins reading from the last successful checkpoint, in this case 10001.

Thus, records 10001-20000 are consumed more than one time.

Being resilient to consumer retries

Even though records may be processed more than one time, your application may want to present the side effects as if records were processed only one time (idempotent processing). Solutions to this problem vary in complexity and accuracy. If the destination of the final data can handle duplicates well, we recommend relying on the final destination to achieve idempotent processing. For example, with Opensearch you can use a combination of versioning and unique IDs to prevent duplicated processing.

In the example application in the previous section, it continuously reads records from a stream, aggregates records into a local file, and uploads the file to Amazon S3. As illustrated, records 10001 -20000 are consumed more than one time resulting in multiple Amazon S3 files with the same data. One way to mitigate duplicates from this example is to ensure that step 3 uses the following scheme:

1. Record Processor uses a fixed number of records per Amazon S3 file, such as 5000.
2. The file name uses this schema: Amazon S3 prefix, shard-id, and `First-Sequence-Num`. In this case, it could be something like `sample-shard000001-10001`.
3. After you upload the Amazon S3 file, checkpoint by specifying `Last-Sequence-Num`. In this case, you would checkpoint at record number 15000.

With this scheme, even if records are processed more than one time, the resulting Amazon S3 file has the same name and has the same data. The retries only result in writing the same data to the same file more than one time.

In the case of a reshard operation, the number of records left in the shard may be less than your desired fixed number needed. In this case, your `shutdown()` method has to flush the file to Amazon S3 and checkpoint on the last sequence number. The above scheme is compatible with reshard operations as well.

Handle startup, shutdown, and throttling

Here are some additional considerations to incorporate into the design of your Amazon Kinesis Data Streams application.

Topics

- Start up data producers and data consumers

- Shut down an Amazon Kinesis Data Streams application
- Read throttling

Start up data producers and data consumers

By default, the KCL begins reading records from the tip of the stream, which is the most recently added record. In this configuration, if a data-producing application adds records to the stream before any receiving record processors are running, the records are not read by the record processors after they start up.

To change the behavior of the record processors so that it always reads data from the beginning of the stream, set the following value in the properties file for your Amazon Kinesis Data Streams application:

```
initialPositionInStream = TRIM_HORIZON
```

By default, Amazon Kinesis Data Streams stores all data for 24 hours. It also supports extended retention of up to 7 days and the long-term retention of up to 365 days. This time frame is called the *retention period*. Setting the starting position to the `TRIM_HORIZON` will start the record processor with the oldest data in the stream, as defined by the retention period. Even with the `TRIM_HORIZON` setting, if a record processor were to start after a greater time has passed than the retention period, then some of the records in the stream will no longer be available. For this reason, you should always have consumer applications reading from the stream and use the CloudWatch metric `GetRecords.IteratorAgeMilliseconds` to monitor that applications are keeping up with incoming data.

In some scenarios, it may be fine for record processors to miss the first few records in the stream. For example, you might run some initial records through the stream to test that the stream is working end-to-end as expected. After doing this initial verification, you would then start your workers and begin to put production data into the stream.

For more information about the `TRIM_HORIZON` setting, see Use shard iterators.

Shut down an Amazon Kinesis Data Streams application

When your Amazon Kinesis Data Streams application has completed its intended task, you should shut it down by terminating the EC2 instances on which it is running. You can terminate the instances using the AWS Management Console or the AWS CLI.

After shutting down your Amazon Kinesis Data Streams application, you should delete the Amazon DynamoDB table that the KCL used to track the application's state.

Read throttling

The throughput of a stream is provisioned at the shard level. Each shard has a read throughput of up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second. If an application (or a group of applications operating on the same stream) attempts to get data from a shard at a faster rate, Kinesis Data Streams throttles the corresponding Get operations.

In an Amazon Kinesis Data Streams application, if a record processor is processing data faster than the limit — such as in the case of a failover — throttling occurs. Because KCL manages the interactions between the application and Kinesis Data Streams, throttling exceptions occur in the KCL code rather than in the application code. However, because the KCL logs these exceptions, you see them in the logs.

If you find that your application is throttled consistently, you should consider increasing the number of shards for the stream.

Streaming tables and S3 delivery for Amazon Kinesis Data Streams

With Amazon Kinesis Data Streams, you can deliver streaming data from your Kinesis data streams to two destination types: streaming tables on Apache Iceberg (Amazon S3 Tables), or general purpose Amazon S3 buckets. You don't need to manage any infrastructure, and you can start delivering within minutes. Amazon Kinesis Data Streams reads from your stream, buffers and aggregates records, and delivers them to your configured destination. Delivery is fully managed – there are no connectors, consumer applications, or compute resources to provision.

These capabilities are supported for streams running in On-Demand Advantage or On-Demand Standard capacity mode. You configure delivery from your stream and specify a destination. Amazon Kinesis Data Streams then handles scaling, retries, and delivery reliability automatically. Delivery does not consume your stream's read throughput and has no impact on existing consumers.

Topics

- How data delivery works
- Delivery destinations
- Data flow
- Key capabilities
- Requirements
- Data delivery concepts
- Streaming tables
- Amazon S3 general purpose delivery
- IAM permissions for data delivery
- Security for data delivery
- Monitoring data delivery
- Best practices for data delivery
- Troubleshooting data delivery
- Delivery quotas and limits

How data delivery works

Data delivery connects your Kinesis data stream to a delivery destination through a managed pipeline:

1. You publish data to a Kinesis data stream in On-Demand mode.
2. You call `CreateChannel` on the stream and specify a destination (streaming tables on Apache Iceberg, or a general purpose Amazon S3 bucket).
3. The delivery reads from the stream, buffers records, and aggregates them into optimally sized files.
4. The delivery writes the files to your configured destination within the data freshness window you specify.

Delivery destinations

Data delivery supports two destination types:

Streaming tables on Apache Iceberg

Streaming tables continuously delivers your Kinesis data stream into Apache Iceberg tables stored in Amazon S3 Tables. As data arrives, it is automatically converted to optimized Apache Parquet format with intelligent inline compaction that eliminates the small files problem and reduces downstream query costs. Within minutes of being published to your stream, data becomes queryable through Amazon Athena, Amazon EMR, Amazon Managed Service for Apache Flink, or any engine that supports Apache Iceberg.

General purpose Amazon S3 buckets

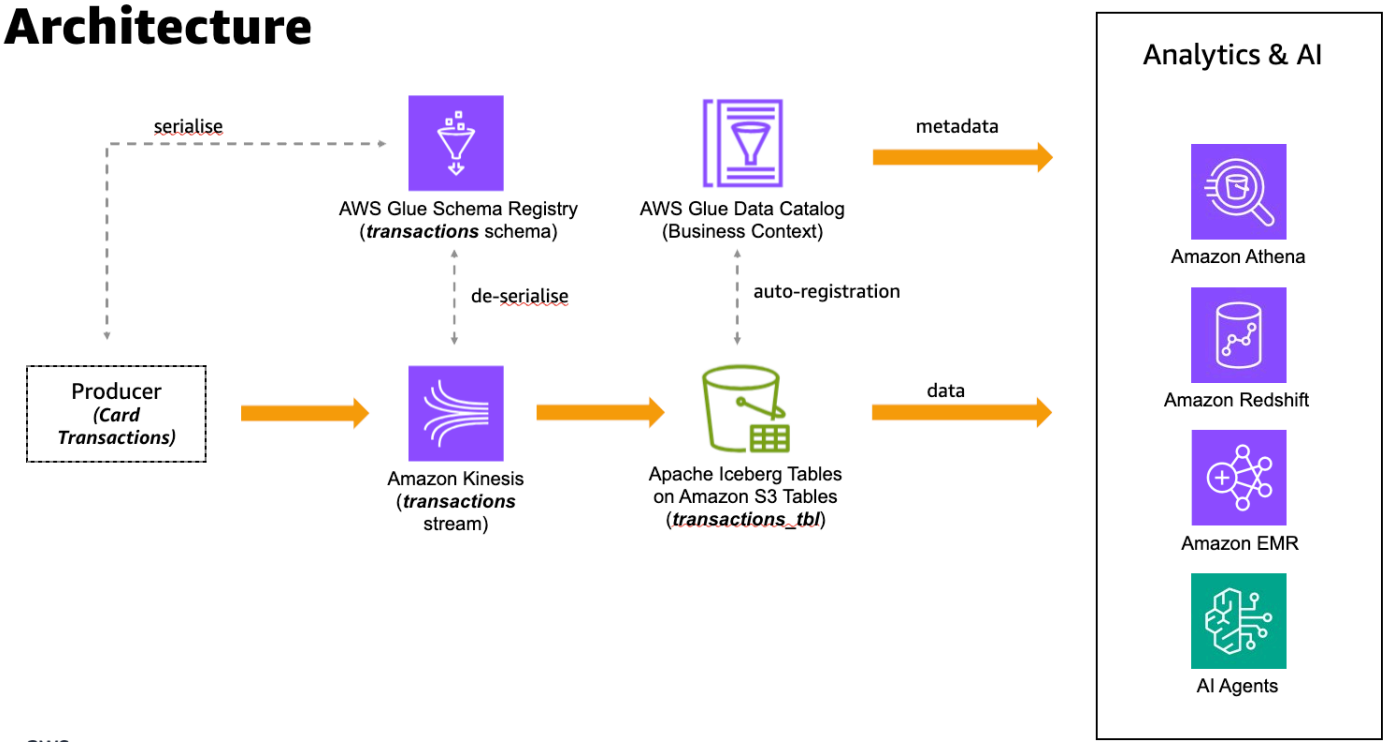
Amazon S3 delivery writes streaming data from your Kinesis data stream directly to an S3 bucket. Records are delivered in their original source format with no transformation applied. Multiple records are buffered and batched into optimally sized objects, with configurable compression and an S3 key structure you define through the output key template. This is ideal for use cases such as raw log archival, event replay, and downstream batch processing, where you need durable, low-cost storage of your streaming data without the overhead of managing a delivery pipeline.

Data flow

The following diagram shows an end-to-end data flow for delivery to streaming tables on Apache Iceberg. The diagram uses a card-transactions use case as an example. A producer serializes records against a schema in AWS Glue Schema Registry and writes them to a Kinesis data stream. Amazon Kinesis Data Streams delivers the records to Apache Iceberg tables on Amazon S3 Tables. If you enable analytics integration in S3 Tables, the table metadata is also registered in the AWS Glue Data Catalog; this does not happen by default. The delivered data and its metadata are then available to analytics and AI engines such as Amazon Athena, Amazon Redshift, and Amazon EMR.

Delivery to general purpose Amazon S3 buckets follows a similar flow, with two differences. A producer writes records to a Kinesis data stream, and Amazon Kinesis Data Streams delivers them to your S3 bucket. Records are delivered in their original source format with no conversion, so AWS Glue Schema Registry is not required and no table metadata is registered in the AWS Glue Data Catalog. Amazon Kinesis Data Streams buffers and batches records into optimally sized objects and writes them using the S3 key structure you define through the output key template. The delivered objects are then available for downstream batch processing and analytics.

Architecture



Key capabilities

Data delivery provides the following key capabilities:

- **Serverless auto-scaling** – Scales automatically with your stream throughput, up to the stream's throughput capacity. No compute resources to provision.
- **Exactly-once delivery per shard** – Records from a shard are delivered to the destination exactly once, with no duplicates or omissions within the shard.
- **Near real-time delivery** – Configurable data freshness from 5 to 15 minutes (300 to 900 seconds).
- **Automatic Parquet conversion** – For streaming tables on Apache Iceberg, converts streaming records into optimized Apache Parquet format for efficient analytics queries.
- **Inline compaction** – Aggregates records into optimally sized files for analytics query performance.
- **Encryption** – Supports customer managed AWS KMS keys for server-side encryption at the destination. The AWS managed key (the `aws/kinesis` alias) is not supported for destination encryption.
- **Dead-letter queue** – Failure metadata for records that cannot be delivered – including stream ARN, shard ID, sequence number, and error context – is written to an S3-based dead-letter queue.
- **CloudWatch metrics and logs** – Monitor bytes delivered, record counts, and data freshness through Amazon CloudWatch metrics. Enable delivery logging to Amazon CloudWatch Logs to capture delivery batch details, failures, and error context for troubleshooting.
- **No impact on other consumers** – Does not consume enhanced fan-out slots or shared throughput.

Requirements

Data delivery has the following requirements:

- Your Kinesis data stream must use On-Demand Standard or On-Demand Advantage capacity mode.
- You must create an IAM service execution role that grants the delivery permissions to write to your destination.
- Your destination bucket or table bucket must be in the same Region as your Kinesis data stream. Data delivery does not support cross-Region delivery for either destination type.

- For streaming tables on Apache Iceberg, cross-account delivery is not supported. The source stream, the destination S3 table bucket, and AWS Glue Schema Registry must all be in the same AWS account and the same Region.
- For general purpose Amazon S3 buckets, cross-account delivery is supported for the destination bucket only. The channel and its source stream must be in the same AWS account; only the destination bucket can be in a different account.
- For streaming tables on Apache Iceberg, you must configure a dead-letter queue in Amazon S3.

Data delivery concepts

Use this topic to review key concepts for streaming tables and Amazon S3 delivery in Amazon Kinesis Data Streams.

Data freshness

Data freshness defines the maximum time (in seconds) that Kinesis Data Streams buffers incoming records before delivering them to the destination. You can configure data freshness between 300 and 900 seconds (5 to 15 minutes). The default value is 300 seconds.

- **Lower values** – Faster delivery to your destination with smaller batch sizes.
- **Higher values** – More efficient batching, which produces fewer, larger files at the destination.

Input formats for streaming tables

When delivering to streaming tables on Apache Iceberg, data delivery supports the following input record formats. Both formats require AWS Glue Schema Registry.

- **JSON** – plain JSON records. You must provide a `GSRSchemaARN` (the of a AWS Glue Schema Registry schema) that references a schema registered in AWS Glue Schema Registry.
- **GSR_JSON** (AWS Glue Schema Registry JSON) – JSON records with the schema ID embedded in each record. The schema is resolved automatically from AWS Glue Schema Registry.

Input formats for general purpose Amazon S3 buckets

When delivering to general purpose Amazon S3 buckets, data delivery supports the following input record formats. No schema registry is required.

- **JSON** – plain JSON records.
- **STRING** – UTF-8 string records.
- **BYTE_ARRAY** – raw binary records.

> **Note**
>
> The `GSR_JSON` format is not supported for general purpose Amazon S3 buckets. It is available only for delivery to streaming tables on Apache Iceberg.

Schema enforcement

For streaming tables on Apache Iceberg, every record is validated against the schema registered in AWS Glue Schema Registry. Records that do not conform to the schema are not delivered to your streaming table. Failure metadata is written to the dead-letter queue for troubleshooting. Data delivery does not support schema evolution. This includes adding new fields, removing existing fields, renaming fields, or changing field data types.

No backfill

Your destination receives only records produced after you create the streaming table or configure the S3 delivery. Existing records in the stream are not backfilled.

New table per delivery

Each streaming table delivery creates its own Iceberg table at the destination. Delivery to existing tables is not supported. You specify the table bucket ARN, namespace, and table name when calling `CreateChannel`.

Dead-letter queue

A dead-letter queue captures information about records that could not be delivered successfully.

- **Streaming tables on Apache Iceberg** – a dead-letter queue is required. You must specify an S3 bucket and prefix for failed records.
- **General purpose Amazon S3 buckets** – a dead-letter queue is optional. If not specified, it defaults to the same destination bucket with an error prefix.

The dead-letter queue contains record identifiers and error context, not full record payloads.

Inline compaction

Data delivery performs inline compaction by aggregating records across shards into optimally sized files. This reduces the number of small files at the destination and improves query performance.

Delivery states

A delivery transitions through the following states during its lifecycle:

- **CREATING** – the delivery is being provisioned.
- **ACTIVE** – the delivery is delivering records.
- **UPDATING** – a configuration change is in progress.
- **DELETING** – the delivery is being removed.
- **FAILED** – the delivery encountered an unrecoverable error.

Append-only delivery

Delivery is append-only. Updates and upserts are not supported. Each record from the stream is written as a new row at the destination.

Streaming tables

Streaming tables continuously delivers records from a Amazon Kinesis Data Streams stream into streaming tables on Apache Iceberg backed by Amazon S3 Tables. As data arrives, it is automatically converted to optimized Apache Parquet format with inline compaction, and becomes queryable through engines such as Amazon Athena, Amazon EMR, and Amazon Managed Service for Apache Flink. This capability requires a schema in AWS Glue Schema Registry and a dead-letter queue.

Topics

- How streaming table delivery works
- Getting started with streaming tables
- Manage streaming table deliveries
- Iceberg behaviors for streaming table

How streaming table delivery works

Use this topic to learn how a streaming table delivery sends records from your Amazon Kinesis Data Streams stream to streaming tables on Apache Iceberg backed by S3 table buckets, including record conversion, destination table creation, partitioning, data freshness, the dead-letter queue, and encryption.

How delivery works

A streaming table delivery sends records to a streaming table on Apache Iceberg in the following steps:

1. **Read** – the delivery reads records from all shards in the Kinesis Data Streams stream.
2. **Buffer** – the delivery buffers records until the data freshness interval is reached.
3. **Validate and convert** – the delivery validates each record against the schema in AWS Glue Schema Registry and converts it to the Apache Iceberg table schema. Records that fail validation go to the dead-letter queue.
4. **Compact and write** – the delivery converts records to optimized Apache Parquet files with inline compaction and writes them to the destination Iceberg table.
5. **Commit** – the delivery commits the new files to the Iceberg table so that the data becomes queryable.

Record conversion

Streaming table delivery requires a schema registered in AWS Glue Schema Registry. The delivery uses the schema to convert incoming records to the Apache Iceberg table schema. It supports the following input record formats:

- **JSON** – plain JSON records. You provide a `GSRSchemaARN` that references a schema registered in AWS Glue Schema Registry.
- **GSR_JSON** – JSON records with the schema ID embedded in each record by the AWS Glue Schema Registry serializer. The schema is resolved automatically from AWS Glue Schema Registry.

For the full AWS Glue Schema Registry to Iceberg type mapping and field-handling rules, see the section called “Iceberg behaviors for streaming table”.

Destination table

When you create a streaming table delivery, Amazon Kinesis Data Streams creates the destination Iceberg table in the S3 table bucket you specify. You provide the following:

- **Table bucket ARN** – the ARN of the S3 table bucket where the table is created.
- **Namespace** – the namespace for the table.
- **Table name** – the name of the table to create. Each delivery creates its own table. You cannot deliver to an existing table.
- **Partition column** – a `timestamptz` column used to partition the table by hour. For partitioning requirements, see the section called “Iceberg behaviors for streaming table”.

Data freshness

Data freshness defines the maximum buffering time before records are delivered. You can configure this value between 300 and 900 seconds (5 to 15 minutes). The default is 300 seconds. Lower values provide faster delivery, while higher values produce fewer, larger files at the destination.

Dead-letter queue

A dead-letter queue is required for streaming table delivery. You specify an Amazon S3 bucket that receives information about records that fail validation. You provide the bucket ARN, the expected bucket owner, and an optional error output prefix. The dead-letter queue contains record identifiers and error context, not full record payloads.

Encryption

Streaming table delivery encrypts delivered data at rest in Amazon S3. By default, data is encrypted with Amazon S3 managed keys (SSE-S3). You can instead use a customer managed AWS KMS key (SSE-KMS).

Important

You cannot use an AWS managed key (the `aws/kinesis` alias) for destination encryption. You must use a customer managed AWS KMS key. In addition, if your source Kinesis data stream is encrypted with an AWS managed key, you cannot create a delivery. For source stream encryption requirements, see the section called “Source stream encryption”. For

the AWS KMS permissions the service execution role needs, see the section called "IAM permissions for data delivery".

Cross-account and cross-Region delivery

Streaming table delivery does not support cross-account or cross-Region delivery. The source stream, the destination S3 table bucket, and the AWS Glue Schema Registry must all be in the same AWS account and the same Region.

Getting started with streaming tables

This tutorial walks you through creating your first streaming table delivery to send records from an Amazon Kinesis Data Streams stream to a streaming table on Apache Iceberg, verifying it, and cleaning up.

Prerequisites

Before you begin, ensure the following prerequisites are met:

- A Kinesis Data Streams stream configured with On-Demand Standard or On-Demand Advantage capacity mode.
- An IAM service execution role that grants Kinesis Data Streams permission to deliver records to your destination. See the section called "IAM permissions for data delivery".
- A schema registered in AWS Glue Schema Registry.
- An S3 table bucket for the destination Iceberg table.
- An Amazon S3 bucket for the dead-letter queue.
- The AWS CLI installed and configured, if you use the AWS CLI.

Step 1: Create the delivery

Create a streaming table delivery with the AWS Management Console or the AWS CLI. For the full procedure and all configuration options, see the section called "Create".

Step 2: Verify the delivery

Verify that the delivery reached the ACTIVE state. For the procedure, see the section called "Describe".

Step 3: Monitor the delivery

After the delivery is active, monitor its health using Amazon CloudWatch metrics published in the `AWS/Kinesis` namespace with the dimensions `ChannelName` and `StreamName`. Streaming table delivery metrics are prefixed with `DeliveryToIceberg`. For the full list of metrics, see the section called “Monitoring data delivery”.

Step 4: Clean up

To avoid ongoing charges, delete the resources you created when you no longer need them. An active delivery continues to process records and incurs delivery charges until you delete it, and any data already written to your destination continues to incur Amazon S3 storage charges.

1. Delete the delivery. See the section called “Delete”.
2. If you no longer need the service execution role – for example, if it is not used by any other delivery – delete it. Detach any customer managed policy, delete any inline policy, and then delete the role using the IAM `detach-role-policy`, `delete-role-policy`, and `delete-role` commands. For details, see Deleting roles or instance profiles in the *IAM User Guide*.
3. If you no longer need the destination table bucket or the dead-letter queue bucket, delete them in Amazon S3 to stop incurring storage charges.

Manage streaming table deliveries

After you understand the prerequisites and IAM requirements, you can create and manage streaming table deliveries with the AWS Management Console or the AWS CLI. The following topics describe each operation. In the API and AWS CLI, a delivery is represented by a channel resource, so the operations use channel terminology (for example, `CreateChannel` and `DescribeChannel`).

Topics

- Create a streaming table delivery
- Describe a streaming table delivery
- Update a streaming table delivery
- Delete a streaming table delivery
- List streaming table deliveries

Create a streaming table delivery

Create a delivery that continuously sends records from an Amazon Kinesis Data Streams stream to a streaming table on Apache Iceberg backed by an S3 table bucket. You can create a delivery with the AWS Management Console or the AWS CLI. When you create a delivery, Amazon Kinesis Data Streams creates the destination Iceberg table in the table bucket you specify.

> **Note**
>
> Before you create a delivery, complete the prerequisites in the section called "Getting started", including the IAM service execution role described in the section called "IAM permissions for data delivery". A streaming table delivery requires a schema in AWS Glue Schema Registry and a dead-letter queue.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation pane, choose **Streaming tables**, and then choose **Create streaming table**.
3. **Step 1: Source configuration.** Under **Delivery details**, enter a **Delivery name**. Acceptable characters are uppercase and lowercase letters, numbers, underscores, hyphens, and periods. For **Source**, choose the On-Demand stream to deliver from. Only On-Demand streams are displayed. Choose **Next**.
4. **Step 2: Destination configuration.** Under **Record conversion configuration**, choose a **Record conversion format**:

   - **JSON** – JSON records that are validated against a schema in AWS Glue Schema Registry. Provide the schema ARN under **Glue Schema registry**.
   - **Glue Schema registry JSON** – Convert records using a AWS Glue Schema Registry JSON schema.

   For **Glue Schema registry**, specify the schema ARN or choose **Browse** to select one.
5. Under **Destination table**, specify where Amazon Kinesis Data Streams creates the Iceberg table:

   - **S3 table bucket ARN** – the ARN of the S3 table bucket. Choose **Browse S3** to select one.

- **Namespace** – the namespace for the table (lowercase letters, numbers, and underscores only).
- **Table name** – the name of the table to create.
- **Column name** – the name of a date-time field from your schema used to partition the table by hour. The field must be a string field with the date-time format in the schema.
- **Compression type** – choose **ZSTD**, **Snappy**, or **None**.
- **Data freshness** – the maximum buffering time in seconds before records are delivered (minimum 300, maximum 900).

Choose **Next**.

6. **Step 3: Additional configuration.** Configure the following, then choose **Next**:

- **Encryption configuration** – by default, data is encrypted with default Amazon S3 encryption. To use a customer managed key, select **Customize encryption settings (advanced)**.
- **Service access** – for **IAM role ARN**, choose or create the IAM role that grants access to the destination. Choose **Create IAM role** to create one.
- **Dead letter queue** – for **S3 bucket URI**, choose the Amazon S3 bucket that stores undeliverable records. Optionally set an **S3 bucket error output prefix**.
- **Log delivery** – optionally select **Deliver to Amazon CloudWatch Logs** to capture delivery diagnostics.
- **Tags** – optionally add key-value tags to identify and organize your delivery.

7. **Step 4: Review and create.** Review your configuration, and then choose **Create streaming table**. The delivery transitions from CREATING to ACTIVE after provisioning is complete.

Using the AWS CLI

Use the `create-channel` command to create a streaming table delivery. This configuration requires a schema from AWS Glue Schema Registry and a dead-letter queue:

```
aws kinesis create-channel \
    --channel-name "my-iceberg-delivery" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
    --stream-configuration-list '[
        {
```

```
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": {
                "RecordFormatType": "JSON",
                "GSRSchemaARN": "arn:aws:glue:us-east-1:111122223333:schema/my-
registry/my-schema"
            }
        }
    ]' \
    --s3-tables-destination-configuration '{
        "DataFreshnessInSeconds": 300,
        "DeadLetterQueueS3Configuration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-dlq-bucket",
            "ExpectedBucketOwner": "111122223333",
            "ErrorOutputPrefix": "dlq/"
        },
        "S3TablesConfigurationList": [
            {
                "TableBucketARN": "arn:aws:s3tables:us-east-1:111122223333:bucket/amzn-
s3-demo-table-bucket",
                "Namespace": "my_namespace",
                "TableName": "my_table",
                "CompressionType": "ZSTD",
                "PartitionSpec": {
                    "PartitionFields": [
                        {
                            "SourceName": "timestamp",
                            "Transform": "TIME_HOUR"
                        }
                    ]
                }
            }
        ]
    }'
```

The command returns the channel ARN (`ChannelARN`). Use it with `describe-channel` to verify the delivery reaches the ACTIVE state. See the section called “Describe”.

API reference – see `CreateChannel` in the *Amazon Kinesis Data Streams API Reference*.

Describe a streaming table delivery

Retrieve the configuration and current status of a streaming table delivery, including its state (`ChannelStatus`) and status reason (`ChannelStatusReason`). Use this to verify that a delivery reached the ACTIVE state after creation, or to diagnose a delivery in the FAILED state.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation pane, choose **Streaming tables**. From the list of deliveries, choose the delivery that you want to view.
3. The delivery details page shows the delivery status, source stream, destination table, data freshness, service access role, and dead-letter queue configuration. Use the **Configurations** tab to view source and destination settings, and the **Logs and tags** tab to view log delivery and tags.

Using the AWS CLI

Use the `describe-channel` command and pass the channel ARN (`ChannelARN`) returned by `create-channel`:

```
aws kinesis describe-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

The response includes the delivery's `ChannelStatus` (CREATING, ACTIVE, UPDATING, DELETING, or FAILED), the source stream configuration, the destination table configuration, and the data freshness setting. A delivery is ready to receive records when `ChannelStatus` is ACTIVE.

API reference – see `DescribeChannel` in the *Amazon Kinesis Data Streams API Reference*.

Update a streaming table delivery

Update an existing streaming table delivery. On an existing delivery, you can change only the data freshness interval (`DataFreshnessInSeconds`) and the CloudWatch Logs configuration. All other properties are immutable after creation. To change any other setting, delete the delivery and create a new one.

Using the AWS Management Console

You edit a delivery from its details page. Choose the tab for the settings you want to change, and then choose **Edit**.

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis, choose **Streaming tables**, and then choose the delivery you want to edit.
2. To change the data freshness setting, on the **Configurations** tab, choose **Edit**. Update the editable settings, and then save your changes.
3. To change log delivery or tags, on the **Logs and tags** tab, choose **Edit**. Update log delivery or tags, and then save your changes.

> **Note**
>
> You can also edit or delete a delivery from the list page by selecting the delivery and using the **Actions** menu.

Using the AWS CLI

Use the `update-channel` command to change the data freshness interval. The following command updates a delivery to a data freshness interval of 600 seconds:

```
aws kinesis update-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111" \
    --s3-tables-destination-configuration '{
        "DataFreshnessInSeconds": 600
    }'
```

The delivery transitions to UPDATING and then back to ACTIVE after the change is applied.

API reference – see `UpdateChannel` in the *Amazon Kinesis Data Streams API Reference*.

Delete a streaming table delivery

Delete a streaming table delivery when you no longer need it. Deleting a delivery stops delivery from the source stream to the destination. Data already delivered to the Iceberg table is not removed.

Important

Deleting a delivery is irreversible. Undelivered records in the stream are not delivered. Data already delivered to the destination remains intact and accessible.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis, choose **Streaming tables**, and then choose the delivery you want to delete.
2. On the delivery details page, choose **Delete**.
3. In the confirmation dialog box, confirm the deletion. The delivery is removed from the list when the deletion completes.

Using the AWS CLI

Use the `delete-channel` command:

```
aws kinesis delete-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
```

To avoid ongoing charges, also delete the IAM role you created for the delivery and, if you no longer need them, the destination table bucket and dead-letter queue bucket. For a full cleanup walkthrough, see the section called “Getting started”.

Note

A stream cannot be deleted while it has active deliveries. To delete the stream, first delete all deliveries attached to it. To find them, use `list-channels` with a stream filter. See the section called “List”.

API reference – see `DeleteChannel` in the *Amazon Kinesis Data Streams API Reference*.

List streaming table deliveries

List the deliveries in your account, optionally filtered to the deliveries attached to a specific stream. Use this to find deliveries before deleting a stream, or to audit the deliveries configured in a AWS Region.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation pane, choose **Streaming tables** to view the list of deliveries, with columns such as name, status, source stream, and destination table.

Using the AWS CLI

Use the `list-channels` command to list deliveries in the current AWS Region:

```
aws kinesis list-channels
```

To list only the deliveries attached to a specific stream, provide a stream filter:

```
aws kinesis list-channels \
    --stream-filter StreamARN="arn:aws:kinesis:us-east-1:111122223333:stream/my-stream"
```

The response returns up to 100 deliveries per page. Use the pagination token in the response to retrieve additional results.

API reference – see `ListChannels` in the *Amazon Kinesis Data Streams API Reference*.

Iceberg behaviors for streaming table

Use this topic to learn how streaming table maps data from AWS Glue Schema Registry schemas to Apache Iceberg tables, including type mapping, field handling, table maintenance, and format specifications.

AWS Glue Schema Registry to Iceberg type mapping

Streaming table maps JSON Schema types from AWS Glue Schema Registry to Apache Iceberg types as shown in the following table.

JSON Schema type	Iceberg type	Notes
`string` (plain)	`string`	Default string mapping.
`string` with format `date-time`	`timestamptz`	Timestamp with time zone.
`string` with format `date`	`date`	Calendar date.
`string` with format `time`	`time`	Time of day.
`string` with format `uuid`	`uuid`	Universally unique identifier.
`string` with encoding `byte` or `base64`	`binary`	Binary data encoded as base64.
`integer` (32-bit or less)	`int`	Values with maximum or exclusive Maximum <= 2^31.
`integer` (greater than 32-bit)	`long`	Values that exceed 32-bit range.
`number` with `multipleOf`	`decimal`	Fixed-point decimal with precision derived from multipleOf.
`number` (plain)	`double`	IEEE 754 double-precision floating point.
`boolean`	`boolean`	True or false.
`object` with named properties	`struct`	Named fields map to struct fields.
`object` with `additionalProperties`	`map`	Key-value map with string keys.
`array`	`list`	Ordered collection of elements.
`enum`	`string`	Enum values stored as strings.

Required columns

Fields listed in the JSON Schema `required` array become required (non-nullable) columns in the Iceberg table. If a required field is missing from a record, the record is sent to the dead-letter queue.

Partition key column

Your table must include a `timestamptz` column that can be used for time-based partitioning by hour (the `TIME_HOUR` transform). In the AWS Glue Schema Registry schema, this is a `string` field with the `date-time` format, which maps to the Iceberg `timestamptz` type.

The source column referenced by the partition is automatically marked as required, regardless of whether it appears in the schema's `required` array. Records missing the partition key value are sent to the dead-letter queue.

You can enable an Amazon S3 Tables record-expiration job based on the table's partition column. For more information, see the section called “Record expiration”.

Field handling

Streaming table handles mismatches between incoming records and the Iceberg table schema as follows:

- **Extra fields** – Fields present in the record but not defined in the schema are dropped and not written to the Iceberg table.
- **Missing optional fields** – Optional fields not present in the record are written as `null` in the Iceberg table.
- **Missing required fields** – If a required field is missing from a record, the entire record is sent to the dead-letter queue.

Nesting limit

Streaming table supports a maximum nesting depth of 16 levels for complex types (structs, maps, and lists). Schemas that exceed 16 levels of nesting are rejected at delivery creation time.

Managed table properties

Streaming table manages Iceberg table properties automatically. These properties are set when the table is created and should not be modified externally. Streaming table uses these properties to control write behavior, compaction, and metadata management.

Because the table is managed by Amazon Kinesis Data Streams, you should not update its schema, modify its table properties, or write to or delete its data directly. You can query the table with AWS analytics services and any compatible Apache Iceberg query engine. For more information about managed table buckets, see Using AWS managed table buckets in the *Amazon S3 User Guide*.

S3 Tables maintenance

When delivering to streaming tables on Apache Iceberg backed by S3 Tables, the following maintenance operations are handled automatically:

- **Compaction** – Small data files are periodically compacted into larger files to improve query performance and reduce metadata overhead.
- **Snapshot expiration** – Expired snapshots are cleaned up to reclaim metadata storage and reduce table metadata size.
- **Unreferenced file cleanup** – Orphaned data files that are no longer referenced by any snapshot are removed to reclaim storage.

Record expiration

You can configure a record expiration period for your Iceberg table. When enabled, Amazon S3 Tables automatically removes records older than the specified retention period during maintenance operations, based on the table's `timestamptz` partition column. For more information, see Managing Amazon S3 Tables record expiration in the *Amazon S3 User Guide*.

Format specifications

Streaming table uses the following format specifications for Iceberg tables:

- **Iceberg format version** – v2
- **Iceberg spec version** – 1.9.0
- **File format** – Apache Parquet

Amazon S3 general purpose delivery

Amazon S3 general purpose delivery writes streaming data from a Amazon Kinesis Data Streams stream directly to a general purpose Amazon S3 bucket. Records are delivered in their original source format with no transformation applied, batched into optimally sized objects with configurable compression and an S3 key structure you define. This is ideal for use cases such as raw log archival, event replay, and downstream batch processing.

Topics

- How Amazon S3 delivery works
- Getting started with S3 general purpose delivery
- Manage Amazon S3 general purpose deliveries
- S3 output key template for Amazon S3 delivery

How Amazon S3 delivery works

Use this topic to learn how an Amazon S3 delivery sends records from an Amazon Kinesis Data Streams stream to general purpose Amazon Simple Storage Service buckets, including storage configuration, compression, storage class, record formats, and encryption.

How delivery works

An Amazon S3 delivery sends records to general purpose Amazon S3 buckets in the following steps:

1. **Read** – the delivery reads records from all shards in the Kinesis Data Streams stream.
2. **Buffer** – records are buffered until the data freshness interval is reached.
3. **Validate** – when the record format is `STRING`, each record is validated as a valid UTF-8 string; when the record format is `JSON`, each record is validated as a valid JSON payload. Records that fail validation are sent to the dead-letter queue.
4. **Compress and format** – records are formatted according to the configured output format and compressed using the specified compression algorithm.
5. **Deliver** – the resulting objects are written to the destination Amazon S3 bucket using the configured key template.
6. **Dead-letter queue** – records that fail validation are written to the dead-letter queue with error context.

Storage configuration

When you create an Amazon S3 delivery, you specify the following storage configuration:

- **BucketARN** – the ARN of the destination Amazon S3 bucket.
- **ExpectedBucketOwner** – the 12-digit AWS account ID of the expected owner of the destination bucket. This value is required and helps prevent delivery to an unintended bucket if ownership changes.
- **CompressionType** – the compression type applied to delivered objects. This value is required; specify `NONE` to deliver objects without compression. See the section called "Compression options".
- **StorageClass** – the Amazon S3 storage class for delivered objects (optional). See the section called "Storage class".
- **OutputKeyTemplate** – the template used to construct the Amazon S3 object key (optional). See the section called "S3 output key template for Amazon S3 delivery".

Compression options

You must specify a compression type for delivery to general purpose Amazon S3 buckets. To deliver objects without compression, choose `NONE`. Choose one of the following compression types:

- **GZIP** – standard gzip compression. Good balance of compression ratio and speed.
- **ZSTD** – Zstandard compression. Higher compression ratio with faster decompression.
- **NONE** – no compression applied. Records are delivered uncompressed.

Storage class

You can optionally specify the Amazon S3 storage class for delivered objects. If not specified, the default storage class is STANDARD. Supported storage classes:

- **STANDARD** – general purpose storage for frequently accessed data.
- **INTELLIGENT_TIERING** – automatically moves data between access tiers based on usage patterns.
- **GLACIER_IR** – low-cost storage for data that requires immediate retrieval.

For more information about Amazon S3 storage classes, see Understanding and managing storage classes in the *Amazon S3 User Guide*.

Output key template

You can configure the output key template to control the Amazon S3 object key structure for delivered files. The template supports dynamic placeholders for date, time, and delivery metadata to organize your data in the destination bucket. For the default template, supported variables, and examples, see the section called "S3 output key template for Amazon S3 delivery".

Data freshness

Data freshness defines the maximum buffering time before records are delivered. You can configure this value between 300 and 900 seconds (5 to 15 minutes). The default is 300 seconds. Lower values provide faster delivery, while higher values produce fewer, larger objects.

Dead-letter queue

A dead-letter queue is optional for delivery to general purpose Amazon S3 buckets. If you do not specify a dead-letter queue configuration, the delivery defaults to writing error information to the same destination bucket with an error prefix.

If you specify a dead-letter queue, you provide the following:

- **BucketARN** – the ARN of the Amazon S3 bucket that receives information about records that could not be delivered.
- **ExpectedBucketOwner** – the 12-digit AWS account ID of the expected owner of the dead-letter queue bucket (required when you specify a dead-letter queue).
- **ErrorOutputPrefix** – an optional prefix for organizing the failure metadata objects within the bucket.

Record formats

Amazon S3 deliveries support the following record formats for delivery to general purpose Amazon S3 buckets:

- **JSON** – plain JSON records.
- **STRING** – UTF-8 string records.

- **BYTE_ARRAY** – raw binary records.

> **Note**
>
> The `GSR_JSON` record format is not supported for delivery to general purpose Amazon S3 buckets. It is supported only for delivery to streaming tables on Apache Iceberg, which require AWS Glue Schema Registry.

Cross-account delivery

Amazon S3 deliveries support cross-account delivery to general purpose Amazon S3 buckets. The source stream and destination bucket must be in the same AWS Region. Configure the destination bucket policy to grant the delivery IAM role permission to write objects.

Encryption

Amazon S3 deliveries support server-side encryption with customer-managed AWS KMS keys for objects delivered to general purpose Amazon S3 buckets. You can specify a customer-managed AWS KMS key to encrypt delivered objects.

> **Important**
>
> You cannot use the `aws/kinesis` AWS KMS alias for destination encryption. You must use a customer-managed AWS KMS key.

If your source Kinesis data stream is encrypted, additional considerations apply. A stream encrypted with an AWS managed key (the `aws/kinesis` alias) is not supported, and the service execution role requires AWS KMS permissions to decrypt source records. For source stream encryption requirements, see the section called “Source stream encryption”. For the AWS KMS permissions the service execution role needs, see the section called “IAM permissions for data delivery”.

Getting started with S3 general purpose delivery

This tutorial walks you through creating your first Amazon S3 delivery to send records from an Amazon Kinesis Data Streams stream to a general purpose Amazon S3 bucket, verifying it, and cleaning up.

Prerequisites

Before you begin, ensure the following prerequisites are met:

- A Kinesis Data Streams stream configured with On-Demand Standard or On-Demand Advantage capacity mode.
- An IAM service execution role that grants Kinesis Data Streams permission to deliver records to your destination. See the section called "IAM permissions for data delivery".
- A general purpose Amazon S3 bucket for the destination.
- The AWS CLI installed and configured, if you use the AWS CLI.

Step 1: Create the delivery

Create an Amazon S3 delivery with the AWS Management Console or the AWS CLI. For the full procedure and all configuration options, see the section called "Create".

Step 2: Verify the delivery

Verify that the delivery reached the ACTIVE state. For the procedure, see the section called "Describe".

Step 3: Monitor the delivery

After the delivery is active, monitor its health using Amazon CloudWatch metrics published in the `AWS/Kinesis` namespace with the dimensions `ChannelName` and `StreamName`. Amazon S3 delivery metrics are prefixed with `DeliveryToS3`. For the full list of metrics, see the section called "Monitoring data delivery".

Step 4: Clean up

To avoid ongoing charges, delete the resources you created when you no longer need them. An active delivery continues to process records and incurs delivery charges until you delete it, and any data already written to your destination continues to incur Amazon S3 storage charges.

1. Delete the delivery. See the section called "Delete".
2. If you no longer need the service execution role – for example, if it is not used by any other delivery – delete it. Detach any customer managed policy, delete any inline policy, and then

delete the role using the IAM `detach-role-policy`, `delete-role-policy`, and `delete-role` commands. For details, see Deleting roles or instance profiles in the *IAM User Guide*.

3. If you no longer need the destination bucket or the dead-letter queue bucket, delete them in Amazon S3 to stop incurring storage charges.

Manage Amazon S3 general purpose deliveries

After you understand the prerequisites and IAM requirements, you can create and manage Amazon S3 deliveries with the AWS Management Console or the AWS CLI. The following topics describe each operation. In the API and AWS CLI, a delivery is represented by a channel resource, so the operations use channel terminology (for example, `CreateChannel` and `DescribeChannel`).

Topics

- Create an Amazon S3 delivery
- Describe an Amazon S3 delivery
- Update an Amazon S3 delivery
- Delete an Amazon S3 delivery
- List Amazon S3 deliveries

Create an Amazon S3 delivery

Create a delivery that sends records from an Amazon Kinesis Data Streams stream to a general purpose Amazon S3 bucket. You can create a delivery with the AWS Management Console or the AWS CLI. Records are delivered in their source format, batched into optimally sized objects, with configurable compression and an S3 key structure you define.

> **Note**
>
> Before you create a delivery, complete the prerequisites in the section called "Getting started", including the IAM service execution role described in the section called "IAM permissions for data delivery".

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.

2. In the navigation pane, choose **S3 general purpose delivery**, and then choose **Create delivery**.
3. **Step 1: Source configuration.** Under **Delivery details**, enter a **Delivery name**. For **Source**, choose the On-Demand stream to deliver from. Only On-Demand streams are displayed. Choose **Next**.
4. **Step 2: Destination configuration.** Choose a **Record format** (**JSON**, **STRING**, or **BYTE_ARRAY**), and then specify the destination:

   - **S3 bucket ARN** – the ARN of the destination general purpose Amazon S3 bucket.
   - **Expected bucket owner** – the 12-digit AWS account ID of the expected bucket owner.
   - **Compression type** – choose **GZIP**, **ZSTD**, or **None**.
   - **Storage class** – optionally choose an Amazon S3 storage class.
   - **Output key template** – optionally define the Amazon S3 object key structure. For details, see the section called “S3 output key template for Amazon S3 delivery”.
   - **Data freshness** – the maximum buffering time in seconds before records are delivered (minimum 300, maximum 900).

   Choose **Next**.
5. **Step 3: Additional configuration.** Configure encryption, service access (the IAM role), an optional dead-letter queue, log delivery, and tags. Choose **Next**.
6. **Step 4: Review and create.** Review your configuration, and then choose **Create delivery**. The delivery transitions from CREATING to ACTIVE after provisioning is complete.

Using the AWS CLI

Use the `create-channel` command to create an Amazon S3 delivery that sends records to a general purpose Amazon S3 bucket with JSON format and GZIP compression:

```
aws kinesis create-channel \
    --channel-name "my-s3-delivery" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
    --stream-configuration-list '[
        {
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": {
                "RecordFormatType": "JSON"
            }
        }
```

```
    ]' \
    --s3-destination-configuration '{
        "DataFreshnessInSeconds": 300,
        "StorageConfiguration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-destination-bucket",
            "ExpectedBucketOwner": "111122223333",
            "CompressionType": "GZIP",
            "StorageClass": "STANDARD"
        }
    }'
```

The command returns the channel ARN (`ChannelARN`). Use it with `describe-channel` to verify the delivery reaches the ACTIVE state. See the section called “Describe”. To customize the Amazon S3 object key structure, see the section called “S3 output key template for Amazon S3 delivery”.

API reference – see `CreateChannel` in the *Amazon Kinesis Data Streams API Reference*.

Describe an Amazon S3 delivery

Retrieve the configuration and current status of an Amazon S3 delivery, including its state (`ChannelStatus`) and status reason (`ChannelStatusReason`). Use this to verify that a delivery reached the ACTIVE state after creation, or to diagnose a delivery in the FAILED state.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation pane, choose **S3 general purpose delivery**. From the list of deliveries, choose the delivery that you want to view.
3. The delivery details page shows the delivery status, source stream, destination bucket, compression, storage class, output key template, data freshness, service access role, and dead-letter queue configuration. Use the **Configurations** tab to view source and destination settings, and the **Logs and tags** tab to view log delivery and tags.

Using the AWS CLI

Use the `describe-channel` command and pass the channel ARN (`ChannelARN`) returned by `create-channel`:

```
aws kinesis describe-channel \
```

```
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

The response includes the delivery's `ChannelStatus` (CREATING, ACTIVE, UPDATING, DELETING, or FAILED), the source stream configuration, the destination bucket configuration, and the data freshness setting. A delivery is ready to receive records when `ChannelStatus` is ACTIVE.

API reference – see `DescribeChannel` in the *Amazon Kinesis Data Streams API Reference*.

Update an Amazon S3 delivery

Update an existing Amazon S3 delivery. On an existing delivery, you can change only the data freshness interval (`DataFreshnessInSeconds`) and the CloudWatch Logs configuration. All other properties are immutable after creation. To change any other setting, delete the delivery and create a new one.

Using the AWS Management Console

You edit a delivery from its details page. Choose the tab for the settings you want to change, and then choose **Edit**.

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis, choose **S3 general purpose delivery**, and then choose the delivery you want to edit.
2. To change the data freshness setting, on the **Configurations** tab, choose **Edit**. Update the editable settings, and then save your changes.
3. To change log delivery or tags, on the **Logs and tags** tab, choose **Edit**. Update log delivery or tags, and then save your changes.

> **Note**
>
> You can also edit or delete a delivery from the list page by selecting the delivery and using the **Actions** menu.

Using the AWS CLI

Use the `update-channel` command to change the data freshness interval. The following command updates a delivery to a data freshness interval of 600 seconds:

```
aws kinesis update-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111" \
    --s3-destination-configuration '{
        "DataFreshnessInSeconds": 600
    }'
```

The delivery transitions to UPDATING and then back to ACTIVE after the change is applied.

API reference – see `UpdateChannel` in the *Amazon Kinesis Data Streams API Reference*.

Delete an Amazon S3 delivery

Delete an Amazon S3 delivery when you no longer need it. Deleting a delivery stops delivery from the source stream to the destination. Data already delivered to the Amazon S3 bucket is not removed.

> **Important**
>
> Deleting a delivery is irreversible. Undelivered records in the stream are not delivered. Data already delivered to the destination remains intact and accessible.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis, choose **S3 general purpose delivery**, and then choose the delivery you want to delete.
2. On the delivery details page, choose **Delete**.
3. In the confirmation dialog box, confirm the deletion. The delivery is removed from the list when the deletion completes.

Using the AWS CLI

Use the `delete-channel` command:

```
aws kinesis delete-channel \
    --channel-arn "arn:aws:kinesis:us-east-1:111122223333:channel/a1b2c3d4-5678-90ab-
cdef-EXAMPLE11111"
```

To avoid ongoing charges, also delete the IAM role you created for the delivery and, if you no longer need them, the destination bucket and dead-letter queue bucket. For a full cleanup walkthrough, see the section called "Getting started".

> **Note**
>
> A stream cannot be deleted while it has active deliveries. To delete the stream, first delete all deliveries attached to it. To find them, use `list-channels` with a stream filter. See the section called "List".

API reference – see `DeleteChannel` in the *Amazon Kinesis Data Streams API Reference*.

List Amazon S3 deliveries

List the deliveries in your account, optionally filtered to the deliveries attached to a specific stream. Use this to find deliveries before deleting a stream, or to audit the deliveries configured in a AWS Region.

Using the AWS Management Console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation pane, choose **S3 general purpose delivery** to view the list of deliveries, with columns such as name, status, source stream, and destination bucket.

Using the AWS CLI

Use the `list-channels` command to list deliveries in the current AWS Region:

```
aws kinesis list-channels
```

To list only the deliveries attached to a specific stream, provide a stream filter:

```
aws kinesis list-channels \
    --stream-filter StreamARN="arn:aws:kinesis:us-east-1:111122223333:stream/my-stream"
```

The response returns up to 100 deliveries per page. Use the pagination token in the response to retrieve additional results.

API reference – see `ListChannels` in the *Amazon Kinesis Data Streams API Reference*.

S3 output key template for Amazon S3 delivery

When delivering to general purpose Amazon S3 buckets, you can configure an S3 output key template to control the object key structure of delivered files. The key template is optional – if not specified, Amazon S3 delivery uses the default template.

Default template

If you do not specify a custom key template, Amazon S3 delivery uses the following default:

```
kinesis-channel/!{channel-name}/!{channel-id}/!{yyyy}/!{MM}/!{dd}/!{HH}/!{channel-name}-!{channel-id}-!{yyyy}-!{MM}-!{dd}-!{HH}-!{mm}!{extension}
```

The portion before the last forward slash (/) forms the object key prefix (the folder structure), and the last segment forms the object name. The default template organizes delivered objects by channel and by hour, and repeats the same values in the object name so that each object is self-describing. Amazon Kinesis Data Streams automatically appends a unique suffix to every object key, so object keys are unique even when the template does not include a time component fine enough to distinguish them.

Template variables

The following variables are available for use in your key template:

S3 key template variables

Variable	Description	Example value
`!{channel-name}`	The name of the delivery.	`my-channel`
`!{channel-id}`	The unique identifier of the delivery.	`abc123def456`
`!{stream-name}`	The name of the source Kinesis Data Streams stream.	`my-stream`
`!{yyyy}`	Four-digit year (UTC).	`2026`
`!{yy}`	Two-digit year (UTC).	`26`
`!{MM}`	Two-digit month (UTC), zero-padded.	`07`

Variable	Description	Example value
`!{dd}`	Two-digit day of month (UTC), zero-padded.	`20`
`!{HH}`	Two-digit hour (UTC, 24-hour), zero-padded.	`14`
`!{mm}`	Two-digit minute (UTC), zero-padded.	`30`
`!{extension}`	A file extension derived from the configured compression type (for example, `.gz` for GZIP or `.zst` for ZSTD). If present, it must appear at the end of the template.	`.gz`
`!{extension:.`*literal*`}`	A literal file extension that you specify. Must start with a period and contain only lowercase letters and periods. If present, it must appear at the end of the template.	`!{extension:.json}`

Template rules

Key templates must conform to the following rules:

- After all variables are expanded, the resulting object key must not exceed the Amazon S3 object key limit of 1,024 characters. Because Amazon Kinesis Data Streams appends a unique suffix of up to 38 characters, the expanded template itself must not exceed 986 characters.
- Must not contain path traversal: no consecutive periods (`..`), no period-only path segment (`.`), and must not start with a forward slash (`/`).
- Must not contain consecutive slashes (`//`).
- Every variable placeholder must be closed (each `!{` must have a matching `}`) and must use a supported variable name.
- Allowed literal characters are alphanumeric characters and the following: `! - _ ' . * ( ) / =`. The `=` character lets you express Hive-style partitioning, such as `year=!{yyyy}/month=!{MM}/`.

- The template can contain at most one extension placeholder (`!{extension}` or `!{extension:.literal}`), and if present it must appear at the end of the template.
- When compression (GZIP or ZSTD) is enabled, the template must include an extension placeholder.

> **Note**
>
> You do not need to include a uniqueness variable in the template. Amazon Kinesis Data Streams automatically appends a unique suffix to each delivered object key.

Valid and invalid examples

Key template examples

Template	Valid	Reason
`!{channel-name}/!{yyyy}/!{MM}/!{dd}/!{HH}/records!{extension}`	Yes	Uses supported variables and ends with a single extension placeholder.
`data/stream=!{stream-name}/!{yyyy}-!{MM}-!{dd}/!{HH}!{mm}!{extension:.json}`	Yes	Hive-style partitioning with a literal extension at the end.
`!{channel-name}/!{channel-id}`	Yes	Minimal template. Kinesis Data Streams appends a unique suffix automatically.
`/data/!{channel-name}!{extension}`	No	Starts with a forward slash.
`data/../!{channel-name}!{extension}`	No	Contains path traversal (`..`).
`data//!{channel-name}!{extension}`	No	Contains consecutive slashes (`//`).

Template	Valid	Reason
`!{channel-name}!{extension}/!{yyyy}`	No	The extension placeholder is not at the end of the template.
`data/!{partition-id}!{extension}`	No	`!{partition-id}` is not a supported variable.
`data/!{channel-name}` (with GZIP or ZSTD compression)	No	Compression is enabled but the template has no extension placeholder.
`data/!{channel-name}/!{yyyy}` (missing closing brace)	No	Unclosed variable placeholder.

IAM permissions for data delivery

Use this topic to learn about the IAM permissions required to create and manage delivery resources in Amazon Kinesis Data Streams, as well as the service execution role that data delivery assumes to deliver data to your destination.

Lifecycle management permissions

To create, update, delete, describe, and list delivery resources, the calling IAM principal must have the following permissions.

Action	Resource	Description
`kinesis:CreateChannel`	Stream ARN (`arn:aws:kinesis:` *`region`*`:`*`account-id`*`:stream/`*`stream-name`*)	Create a delivery on a Kinesis Data Streams stream.
`kinesis:AssociateStreamsWithChannel`	Stream ARN (`arn:aws:kinesis:` *`region`*`:`*`account-id`*`:stream/`*`stream-name`*)	Associate a stream with a delivery. This is a virtual action that the calling principal must be granted to create a delivery that reads from the specified stream. It

Action	Resource	Description
		is authorized in addition to `kinesis:CreateChannel` .
`kinesis:UpdateChannel`	Channel ARN (`arn:aws:kinesis:` *region*`:`*account-id*`:channel/` *channel-id*)	Update an existing delivery configuration.
`kinesis:DeleteChannel`	Channel ARN	Delete an existing delivery.
`kinesis:DescribeChannel`	Channel ARN	Retrieve details of a delivery.
`kinesis:ListChannels`	Account level (`arn:aws:kinesis:` *region*`:`*account-id*`:stream/*`)	List all deliveries in the account.

Service execution role

Data delivery assumes an IAM service execution role to deliver data to your destination. You must create this role and attach both a trust policy and a permission policy.

Trust policy

The trust policy allows the Kinesis Data Streams service to assume the role. Include condition keys to prevent the confused deputy problem.

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

```
                    "aws:SourceAccount": "<account-id>"
                },
                "ArnLike": {
                    "aws:SourceArn": "arn:aws:kinesis:<region>:<account-id>:channel/*"
                }
            }
        }
    ]
}
```

Permission policy for streaming tables on Apache Iceberg

When delivering to streaming tables on Apache Iceberg, the service execution role requires the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3TablesAccess",
            "Effect": "Allow",
            "Action": [
                "s3tables:GetTable",
                "s3tables:GetTableBucket",
                "s3tables:GetTableMetadataLocation",
                "s3tables:UpdateTableMetadataLocation",
                "s3tables:CreateTable",
                "s3tables:CreateNamespace",
                "s3tables:PutTableData",
                "s3tables:GetTableData",
                "s3tables:TagResource",
                "s3tables:PutTableRecordExpirationConfiguration",
                "s3tables:PutTableEncryption"
            ],
            "Resource": [
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>",
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>/
table/*"
            ]
        },
        {
            "Sid": "AllowCreateTableWithTag",
            "Effect": "Allow",
```

```
            "Action": "s3tables:CreateTable",
            "Resource": [
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>/
table/*",
                "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-name>"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/TableName": "<table-name>"
                }
            }
        },
        {
            "Sid": "AllowPutTableDataWithTag",
            "Effect": "Allow",
            "Action": "s3tables:PutTableData",
            "Resource": "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-
name>/table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/TableName": "<table-name>"
                }
            }
        },
        {
            "Sid": "DLQBucketAccess",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<dlq-bucket-name>",
                "arn:aws:s3:::<dlq-bucket-name>/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "<dlq-account-id>"
                }
            }
        },
        {
            "Sid": "GlueSchemaRegistryAccess",
```

```
            "Effect": "Allow",
            "Action": [
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "arn:aws:glue:<region>:<account-id>:registry/<registry-name>",
                "arn:aws:glue:<region>:<account-id>:schema/<registry-name>/<schema-
name>"
            ]
        },
        {
            "Sid": "KMSForCreateTimeValidation",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForS3TablesEncryption",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3.<region>.amazonaws.com"
                },
                "StringLike": {
                    "kms:EncryptionContext:aws:s3:arn": [
                        "arn:aws:s3tables:<region>:<account-id>:bucket/<table-bucket-
name>/*",
                        "arn:aws:s3:::<dlq-bucket-name>/*"
                    ]
                }
            }
        },
        {
            "Sid": "KMSForCreateTable",
            "Effect": "Allow",
            "Action": ["kms:DescribeKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
```

```
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3tables.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForDecryptSourceStreamRecords",
            "Effect": "Allow",
            "Action": "kms:Decrypt",
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<stream-key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                },
                "StringEquals": {
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:<region>:<account-id>:stream/<stream-name>"
                }
            }
        }
    ]
}
```

- `s3tables:PutTableEncryption` is required only when you encrypt the destination table with a customer managed AWS KMS key. Without it, `CreateTable` succeeds but table encryption fails and the table is never created.
- `s3tables:TagResource` is required because the service tags the tables it creates.
- The `AllowCreateTableWithTag` and `AllowPutTableDataWithTag` statements are optional. Include them only if you want tag-based access control instead of scoping solely by table ARN. They restrict `s3tables:CreateTable` and `s3tables:PutTableData` to resources carrying a matching `TableName` tag. Omit them if you scope by ARN.
- Only `glue:GetSchemaVersion` is required for the Glue Schema Registry. It is required for both the `JSON` and `GSR_JSON` input formats.
- The `KMSForCreateTimeValidation` statement lets Amazon Kinesis Data Streams validate your AWS KMS key when you create the delivery. It is required only when your destination table or source stream uses a customer managed AWS KMS key.

- The `KMSForS3TablesEncryption` statement grants the AWS KMS operations used to encrypt delivered table data at rest. It is required only when your table bucket uses a customer managed key.
- The `KMSForCreateTable` statement lets Amazon Kinesis Data Streams read the key configuration when it creates the destination table. It is required only when the table bucket uses a customer managed key.
- The `KMSForDecryptSourceStreamRecords` statement lets the role decrypt records read from the source stream. It is required only when your Kinesis Data Streams stream is encrypted with a customer managed key.

Permission policy for general purpose Amazon S3 buckets

When delivering to general purpose Amazon S3 buckets, the service execution role requires the following permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DeliveryBucketList",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<delivery-bucket-name>",
                "arn:aws:s3:::<delivery-bucket-name>/*"
            ]
        },
        {
            "Sid": "DeliveryBucketWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:CreateMultipartUpload",
                "s3:UploadPart",
                "s3:CompleteMultipartUpload",
                "s3:ListMultipartUploads",
                "s3:ListMultipartUploadParts"
```

```
            ],
            "Resource": [
                "arn:aws:s3:::<delivery-bucket-name>/*"
            ]
        },
        {
            "Sid": "DLQBucketAccess",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::<dlq-bucket-name>",
                "arn:aws:s3:::<dlq-bucket-name>/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "<dlq-account-id>"
                }
            }
        },
        {
            "Sid": "KMSForCreateTimeValidation",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSForS3BucketEncryption",
            "Effect": "Allow",
            "Action": ["kms:Decrypt", "kms:GenerateDataKey"],
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<key-id>"],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3.<region>.amazonaws.com"
                },
                "StringLike": {
```

```
                    "kms:EncryptionContext:aws:s3:arn": [
                        "arn:aws:s3:::<delivery-bucket-name>/*",
                        "arn:aws:s3:::<dlq-bucket-name>/*"
                    ]
                }
            }
        },
        {
            "Sid": "KMSForDecryptSourceStreamRecords",
            "Effect": "Allow",
            "Action": "kms:Decrypt",
            "Resource": ["arn:aws:kms:<region>:<account-id>:key/<stream-key-id>"],
            "Condition": {
                "StringEqualsIfExists": {
                    "kms:ViaService": "kinesis.<region>.amazonaws.com"
                },
                "StringEquals": {
                    "kms:EncryptionContext:aws:kinesis:arn":
 "arn:aws:kinesis:<region>:<account-id>:stream/<stream-name>"
                }
            }
        }
    ]
}
```

- The `KMSForCreateTimeValidation` statement lets Amazon Kinesis Data Streams validate your AWS KMS key when you create the delivery. It is required only when your destination bucket or source stream uses a customer managed AWS KMS key.
- The `KMSForS3BucketEncryption` statement grants the AWS KMS operations that Amazon S3 uses to encrypt delivered objects at rest. It is required only when your destination bucket or dead-letter queue bucket uses SSE-KMS with a customer managed key. The `kms:EncryptionContext:aws:s3:arn` values shown use the object ARN, which applies when S3 Bucket Keys are disabled. If you enable S3 Bucket Keys on the destination bucket or the dead-letter queue bucket, Amazon S3 uses the bucket ARN as the encryption context instead of the object ARN, so you must change the corresponding value to that bucket ARN (for example, `arn:aws:s3:::`*`delivery-bucket-name`* or `arn:aws:s3:::`*`dlq-bucket-name`*). For more information, see Configuring an S3 Bucket Key in the *Amazon S3 User Guide*.
- The `KMSForDecryptSourceStreamRecords` statement lets the role decrypt records read from the source stream. It is required only when your Kinesis Data Streams stream is encrypted with a customer managed key.

> **Important**
>
> If you restrict the `s3:PutObject` resource to a specific prefix (for example, `arn:aws:s3:::my-bucket/data*`), the object keys generated by your output key template must start with that same prefix. A mismatch between the IAM resource prefix and the output key template results in access denied errors and no data being delivered. For example, if your policy grants `s3:PutObject` on `arn:aws:s3:::my-bucket/data*`, your output key template must begin with `data/`, such as `data/!{yyyy}/!{MM}/!{dd}/!{HH}/`. To avoid this, either scope the resource to the entire bucket (`arn:aws:s3:::my-bucket/*`) or ensure the prefixes match exactly.

Optional CloudWatch Logs permissions

If you enable CloudWatch Logs for your delivery, add the following permissions to the service execution role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CloudWatchLogsAccess",
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:<region>:<account-id>:log-group:<log-group-
name>:*"
        }
    ]
}
```

Security for data delivery

Use this topic to learn about the security features and best practices for data delivery in Amazon Kinesis Data Streams, including encryption, access control, and audit logging.

Encryption in transit

All communication between data delivery and its destinations is encrypted in transit using TLS 1.2 or later. This includes data delivered to streaming tables on Apache Iceberg and general purpose Amazon S3 buckets, as well as communication with AWS Glue Schema Registry and the dead-letter queue.

Encryption at rest

Streaming tables and Amazon S3 delivery store delivered data in Amazon S3. All delivered data is encrypted at rest server-side. This applies to both destination types, because streaming tables on Apache Iceberg and general purpose Amazon S3 buckets both store data in Amazon S3. You do not need to take any action for your delivered data to be encrypted.

Options for encryption at rest

You can choose one of the following encryption options for delivered data:

- **SSE-S3 (default)** – Server-side encryption with Amazon S3 managed keys. This is the default if no customer managed key is specified. No additional configuration is required.
- **SSE-KMS with a customer managed key** – Server-side encryption with a symmetric customer managed AWS KMS key that you own and manage.

Using a customer managed key for encryption at rest

Because delivered data is stored in Amazon S3 for both destination types, encryption at rest works the same way as Amazon S3 server-side encryption, including how AWS KMS keys are used, the required key permissions, encryption context, and auditing. Only symmetric encryption AWS KMS keys are supported. For complete details about how Amazon S3 uses AWS KMS keys for server-side encryption, see Using server-side encryption with AWS KMS keys (SSE-KMS) in the *Amazon S3 User Guide*.

To encrypt delivered data with a customer managed key, specify the key in the `EncryptionConfiguration` parameter when you create the delivery with `CreateChannel`. For more information, see the CreateChannel operation in the *Amazon Kinesis Data Streams API Reference*. The encryption configuration is immutable after a delivery is created; to change it, delete the delivery and create a new one.

For the specific AWS KMS permissions the service execution role needs to use a customer managed key, see the section called “IAM permissions for data delivery”.

> **Note**
>
> You must use a customer managed AWS KMS key for destination encryption. An AWS managed key (the `aws/kinesis` alias) is not supported. Source streams encrypted with an AWS managed key are also not supported; see the section called “Source stream encryption”.

Monitoring AWS KMS interaction

You can audit how your customer managed key is used with AWS CloudTrail. AWS KMS cryptographic operations that encrypt and decrypt delivered objects, such as `GenerateDataKey` and `Decrypt`, appear in your CloudTrail event history. For more information about finding AWS KMS events, see Auditing AWS KMS key usage in the *AWS Key Management Service Developer Guide*.

Source stream encryption

> **Important**
>
> If your Kinesis data stream is encrypted with an AWS managed key (the `aws/kinesis` alias), you cannot create a delivery. `CreateChannel` fails synchronously for streams encrypted with an AWS managed key.

To use streaming tables or Amazon S3 delivery on an encrypted stream, use a customer managed key for stream encryption. You can change the encryption key on an existing stream using `StartStreamEncryption`. Streams encrypted with a customer managed key are fully supported.

IAM access control

Follow these best practices for IAM access control with data delivery:

- **Least privilege** – Grant only the minimum permissions required for the service execution role. Scope Amazon S3 permissions to specific buckets and prefixes rather than using wildcards.
- **Scope to specific resources** – Restrict permissions to the specific bucket, table bucket, or table that the delivery writes to. Avoid using `Resource: "*"`.

- **Confused deputy prevention** – Always include `aws:SourceArn` and `aws:SourceAccount` conditions in the trust policy to prevent other services or accounts from assuming the execution role.

AWS CloudTrail logging

All delivery API calls are logged by AWS CloudTrail. This includes `CreateChannel`, `UpdateChannel`, `DeleteChannel`, `DescribeChannel`, and `ListChannels`. You can use AWS CloudTrail logs to audit who created, modified, or deleted delivery resources, and when those actions occurred.

Monitoring data delivery

Use this topic to learn how to monitor your delivery using Amazon CloudWatch metrics, CloudWatch Logs, and AWS CloudTrail.

CloudWatch metrics

Each delivery publishes metrics to CloudWatch in the `AWS/Kinesis` namespace. Metric names are prefixed by destination type: `DeliveryToS3` for general purpose Amazon S3 deliveries and `DeliveryToIceberg` for streaming table deliveries. Use these metrics to monitor the health and performance of your delivery.

The following table lists the metrics for general purpose Amazon S3 deliveries (`DeliveryToS3`).

Metric name	Unit	Description
`DeliveryToS3.DataFreshness`	Seconds	The age of the oldest record not yet delivered to the destination. Rising values indicate delivery is falling behind.
`DeliveryToS3.BytesIn`	Bytes	The volume of data read into the delivery path from the stream.
`DeliveryToS3.BytesProcessed`	Bytes	The volume of data processed by the delivery.
`DeliveryToS3.BytesOut`	Bytes	The volume of data written to the destination.

Metric name	Unit	Description
DeliveryToS3.RecordCount	Count	The total number of records read from the stream.
DeliveryToS3.SuccessfulRecordCount	Count	The number of records successfully delivered to the destination.
DeliveryToS3.FailedRecordCount	Count	The number of records that failed delivery. A non-zero value is the key error signal.
DeliveryToS3.DeliverySuccess	Count	The number of successful delivery operations.
DeliveryToS3.DLQDeliverySuccess	Count	The number of records successfully written to the dead-letter queue.

The following table lists the metrics for streaming table deliveries (DeliveryToIceberg).

Metric name	Unit	Description
DeliveryToIceberg.DataFreshness	Seconds	The age of the oldest record not yet delivered to the table. Rising values indicate delivery is falling behind.
DeliveryToIceberg.BytesIn	Bytes	The volume of data read into the delivery path from the stream.
DeliveryToIceberg.BytesProcessed	Bytes	The volume of data processed by the delivery.
DeliveryToIceberg.BytesOut	Bytes	The volume of data written to the table.
DeliveryToIceberg.TotalRowCount	Count	The total number of rows read from the stream.

Metric name	Unit	Description
`DeliveryToIceberg.SuccessfulRowCount`	Count	The number of rows successfully delivered to the table.
`DeliveryToIceberg.FailedRowCount`	Count	The number of rows that failed delivery. A non-zero value is the key error signal.
`DeliveryToIceberg.CommitSuccess`	Count	The number of successful Iceberg commits.
`DeliveryToIceberg.DLQDeliverySuccess`	Count	The number of records successfully written to the dead-letter queue.

Dimensions

Delivery metrics are published with the following dimensions:

- `ChannelName` – The delivery name.
- `ChannelId` – The unique identifier of the delivery.
- `StreamName` – The name of the source Kinesis Data Streams stream.

> **Note**
>
> To target a specific delivery in a CloudWatch alarm, specify all three dimensions. For browsing or querying with `GetMetricData` and `ListMetrics`, you can filter by a subset of dimensions, such as `ChannelName` alone.

Recommended alarms

The following table lists recommended CloudWatch alarms for deliveries.

Alarm	Metric	Condition	Description
High failure rate	`DeliveryToS3.FailedRecordCount`	> 0 for 5 minutes	Records are failing delivery. Investigate schema mismatches or destination issues.
Data freshness degradation	`DeliveryToS3.DataFreshness`	> threshold	Data delivery is falling behind. Investigate delivery issues such as missing permissions on the service execution role or throttling at the destination.
No data delivered	`DeliveryToS3.SuccessfulRecordCount`	= 0 for 15 minutes	No records have been delivered. Verify the stream has data and the delivery is active.
DLQ delivery failures	`DeliveryToS3.DLQDelivery Success`	= 0 when FailedRecordCount > 0	Failed records cannot be written to the dead-letter queue. Check DLQ bucket permissions.

Note

The preceding alarms use the `DeliveryToS3` metrics for an Amazon S3 delivery. For a streaming table delivery, use the equivalent `DeliveryToIceberg` metrics (`DeliveryToIceberg.FailedRowCount`, `DeliveryToIceberg.DataFreshness`, `DeliveryToIceberg.SuccessfulRowCount`, and `DeliveryToIceberg.DLQDeliverySuccess`).

CloudWatch Logs

Each delivery can publish detailed logs to CloudWatch Logs for debugging and operational visibility.

By default, a delivery writes to the following log group and log stream:

- **Default log group** – `/aws/kinesis/channelName/channelId`
- **Log stream** – `DestinationDelivery`

Enabling CloudWatch Logs via CLI

Enable CloudWatch Logs when creating or updating a delivery using the AWS CLI:

```
aws kinesis create-channel \
    --channel-name "my-channel" \
    --service-execution-role-arn "arn:aws:iam::111122223333:role/my-channel-role" \
    --stream-configuration-list '[
        {
            "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/my-stream",
            "RecordConfiguration": { "RecordFormatType": "JSON" }
        }
    ]' \
    --s3-destination-configuration '{
        "StorageConfiguration": {
            "BucketARN": "arn:aws:s3:::amzn-s3-demo-destination-bucket",
            "ExpectedBucketOwner": "111122223333",
            "CompressionType": "GZIP"
        }
    }' \
    --logging-configuration '{
        "CloudWatchLogs": {
            "Enabled": true
        }
    }'
```

When you enable CloudWatch Logs without specifying a log group, the delivery uses the default log group `/aws/kinesis/channelName/channelId`. If you specify a custom `LogGroupName`, make sure the service execution role's CloudWatch Logs permissions are scoped to that same log group. For the required permissions, see the section called “IAM permissions for data delivery”.

You can also enable or update CloudWatch Logs on an existing delivery with the `update-channel` command, passing the same `--logging-configuration` value.

Required permissions

To enable CloudWatch Logs, the delivery's service execution role must have the following permissions:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Scope these permissions to the specific log group ARN. See the section called “Optional CloudWatch Logs permissions” for the full policy example.

AWS CloudTrail logging

All delivery API calls are recorded by AWS CloudTrail. The following is an example AWS CloudTrail event for a `CreateChannel` call:

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA1EXAMPLE:user-session",
        "arn": "arn:aws:sts::<account-id>:assumed-role/<role-name>/user-session",
        "accountId": "<account-id>"
    },
    "eventTime": "2026-07-20T20:00:00Z",
    "eventSource": "kinesis.amazonaws.com",
    "eventName": "CreateChannel",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.1",
    "userAgent": "aws-cli/2.x",
    "requestParameters": {
        "streamName": "my-stream",
        "channelName": "my-channel"
    },
    "responseElements": {
        "channelARN": "arn:aws:kinesis:us-east-1:<account-id>:channel/<channel-id>"
    },
    "requestID": "a1b2c3d4-example",
    "eventID": "e5f6g7h8-example",
    "readOnly": false,
```

```
    "eventType": "AwsApiCall",
    "recipientAccountId": "<account-id>"
}
```

Best practices for data delivery

The following best practices help you get the most out of data delivery in Amazon Kinesis Data Streams. For the complete list of delivery metrics referenced in these recommendations, see the section called “Monitoring data delivery”.

Throughput and data freshness

Tune delivery frequency and watch delivery latency so your output file sizes and data freshness match your workload:

- Set the `DataFreshnessInSeconds` parameter based on your stream throughput. Lower values increase delivery frequency but might result in smaller output files. Higher values allow more data to accumulate for each delivery cycle.
- Create a Amazon CloudWatch alarm on the `DataFreshness` metric to detect when delivery latency exceeds your acceptable threshold.
- You can create multiple deliveries from the same Kinesis Data Streams stream. Each delivery operates independently, allowing you to deliver data to multiple destinations simultaneously.

Schema management (streaming tables on Apache Iceberg)

Because streaming table deliveries bind to a fixed schema and don't support schema evolution, manage your schemas carefully to avoid failed deliveries:

- Register your schemas in AWS Glue Schema Registry before creating a delivery. The delivery uses the registered schema to define the destination table structure.
- Deliveries do not support schema evolution. If your schema changes, you must delete and recreate the delivery with the updated schema.
- Use producer-side schema validation to ensure all records conform to the registered schema before they are written to the stream. This minimizes failed record delivery.

S3 object layout (general purpose Amazon S3 buckets)

Lay out your delivered objects to improve downstream query performance and control storage costs:

- Choose an output key template that matches the query patterns of your downstream consumers. For example, use date-based prefixes if your queries frequently filter by time range.
- Enable GZIP or ZSTD compression to reduce storage costs and improve read performance for downstream analytics workloads.
- Choose the appropriate Amazon S3 storage class based on your data access patterns. Use STANDARD for frequently accessed data, INTELLIGENT_TIERING to let Amazon S3 move objects between access tiers automatically, or GLACIER_IR for rarely accessed data that still needs millisecond retrieval.

Security

Protect your delivery data and credentials with least-privilege access, encryption, and auditing:

- Scope your IAM policy to the specific bucket or table that the delivery writes to. Avoid using wildcard resource ARNs.
- Include the `aws:SourceArn` and `aws:SourceAccount` condition keys in the trust policy of the IAM role used by the delivery. This prevents the confused deputy problem by ensuring only your specific delivery can assume the role.
- Enable AWS CloudTrail logging to audit all delivery API calls and configuration changes.
- Use a customer-managed AWS KMS key for sensitive data. This gives you full control over key rotation, access policies, and audit trails.

Dead-letter queue

Capture and diagnose records that can't be delivered by configuring and monitoring a dead-letter queue:

- Always configure a dead-letter queue for your delivery. Records that cannot be delivered to the destination are sent to the dead-letter queue for later inspection.

- Monitor the `DLQDeliverySuccess` metric (`DeliveryToS3.DLQDeliverySuccess` or `DeliveryToIceberg.DLQDeliverySuccess`) in CloudWatch to detect when records are being routed to the dead-letter queue.
- Inspect dead-letter queue entries regularly to identify patterns in delivery failures and take corrective action at the producer or delivery configuration level.

Monitoring

Detect delivery problems early by alarming on key CloudWatch metrics and enabling delivery logs:

- Create CloudWatch alarms on the delivery's `DataFreshness` and failed-record metrics (`DeliveryToS3.FailedRecordCount` or `DeliveryToIceberg.FailedRowCount`) to proactively detect delivery issues.
- Enable CloudWatch Logs for your delivery to capture detailed delivery diagnostics and error messages.
- Track the delivered-bytes metric (`DeliveryToS3.BytesOut` or `DeliveryToIceberg.BytesOut`) to monitor throughput and identify unexpected drops in data volume.

Troubleshooting data delivery

Use this section to resolve common issues with data delivery.

Delivery stuck in CREATING state

When you create a delivery, it enters the CREATING state while resources are provisioned. Provisioning typically completes within a few minutes. If the delivery remains in CREATING for an extended period or transitions to FAILED, a configuration error is likely the cause.

Call `DescribeChannel` to check the current status and status reason. Common causes include:

- Invalid IAM role ARN or insufficient permissions in the role policy.
- The destination Amazon S3 bucket does not exist or is in a different region.
- The schema ARN in AWS Glue Schema Registry cannot be resolved.

Delivery in FAILED state

You cannot recover a delivery whose `ChannelStatus` is `FAILED` (as returned by `DescribeChannel`). Read the `ChannelStatusReason` field from `DescribeChannel` to identify the root cause. Fix the underlying issue, delete the failed delivery, and recreate it with corrected configuration.

High data freshness

The `DataFreshness` metric measures the age of the oldest undelivered record. High values indicate delivery is falling behind ingestion. Common causes:

- High partition count on the destination table increases metadata overhead.
- Table metadata growth from many small commits reduces commit throughput.
- Low stream throughput combined with a low freshness setting causes frequent small deliveries.

Resolution: For streaming tables on Apache Iceberg, enable Amazon S3 Tables maintenance (compaction and snapshot expiration) to manage metadata growth. For low-throughput streams, increase the `DataFreshnessInSeconds` value to allow more data to batch for each delivery cycle.

The tightest data freshness setting requires a minimum sustained stream throughput so that enough data accumulates for efficient delivery and inline compaction each cycle. If your stream produces less than that throughput, use a higher `DataFreshnessInSeconds` value.

Failed records greater than zero

When the failed-record metric is non-zero (`DeliveryToS3.FailedRecordCount` for Amazon S3 deliveries or `DeliveryToIceberg.FailedRowCount` for streaming table deliveries), records are being sent to the dead-letter queue instead of the destination.

For streaming tables on Apache Iceberg:

- Schema mismatch – the record does not conform to the registered schema.
- Missing required field – a non-nullable column has no value in the record.
- Not using the AWS Glue Schema Registry serializer for GSR_JSON format – producers must use the AWS Glue Schema Registry producer library.

For general purpose Amazon S3 buckets:

- Format mismatch – the record format does not match the configured input format.

Resolution: Inspect the dead-letter queue entries for detailed error information. Check CloudWatch Logs for the delivery to see specific parsing or validation errors. Fix the producer to send conforming records.

No data appearing at the destination

If the delivery is in ACTIVE state but no data appears at the destination, the most common causes are:

- Permission problems – the IAM role cannot write to the destination. Check CloudWatch Logs for `AccessDenied` errors.
- Output key prefix mismatch (general purpose Amazon S3 buckets) – if your `s3:PutObject` permission is scoped to a prefix such as `arn:aws:s3:::my-bucket/data*`, the keys generated by your output key template must start with `data/`. A mismatch causes every write to be denied.
- Missing table permission (streaming tables) – when you encrypt the destination table with a customer managed AWS KMS key, verify the service execution role includes `s3tables:PutTableEncryption` in addition to the other `s3tables` actions. Without it, `CreateTable` succeeds but table encryption fails, the table is never created, and no data is delivered.
- Missing logging permissions – if the service execution role lacks `logs:CreateLogStream` and `logs:PutLogEvents`, delivery failures are not recorded in CloudWatch Logs, which can make a permissions problem appear as a silent failure. Verify the logging permissions first when no logs are present.
- No new data after creation – the delivery does not backfill existing data from the stream. Only records written after the delivery becomes ACTIVE are delivered.

Delivery suspended

A delivery enters a suspended state when the destination becomes unavailable or incompatible. Common causes:

- The streaming tables on Apache Iceberg destination table was deleted.

- The Amazon S3 bucket owner does not match the expected account (ownership mismatch).
- An incompatible partition column was detected on the destination table.

A suspended delivery cannot be resumed. You must create a new delivery with a valid destination configuration.

Permission denied errors in CloudWatch Logs

`AccessDenied` errors in the delivery's CloudWatch Logs indicate a permissions problem. Common causes:

- The IAM role policy was modified after the delivery was created.
- The Amazon S3 bucket policy was changed to deny access from the role.
- The trust policy on the role does not allow the Kinesis Data Streams service to assume it.
- The AWS KMS key policy denies encrypt or decrypt access to the delivery's role.

Review and correct the relevant policy, then verify that the delivery resumes.

Delivery not available for my stream

Streaming tables and Amazon S3 delivery require the Kinesis Data Streams stream to be in On-Demand Standard or On-Demand Advantage capacity mode. If your stream uses Provisioned mode, you must switch it to On-Demand mode before you can create a delivery.

Delivery stops after a schema change

Deliveries do not support schema evolution. If you update the schema in AWS Glue Schema Registry after creating a delivery, records produced with the new schema version might fail validation and be routed to the dead-letter queue.

To resolve: revert the schema change at the producer, or delete the existing delivery and recreate it with the updated schema.

Cannot delete a stream

A `DeleteStream` request fails with `ResourceInUseException` when the stream has one or more active deliveries. A stream cannot be deleted while deliveries are attached to it.

To resolve: list the deliveries on the stream with `ListChannels` (using a stream filter), delete each delivery with `DeleteChannel`, and then delete the stream.

Delivery quotas and limits

The following tables describe the quotas and delivery constraints for streaming tables and Amazon S3 delivery in Amazon Kinesis Data Streams.

Resource quotas

Resource	Default	Adjustable	Notes
Deliveries per stream	2	No	One delivery to streaming tables on Apache Iceberg and one delivery to a general purpose Amazon S3 bucket.
Streams per delivery	1	No	Each delivery reads from exactly one stream.
Tables per delivery	1	No	Each delivery writes to exactly one destination table.

Note

The number of deliveries in an account is bounded by the On-Demand streams per account per Region quota, because each stream can have a maximum of two deliveries. For the streams-per-account quota, see *Quotas and limits*.

API limits

Limit	Default	Notes
API throttle rate (per account per Region)	5 transactions per second (TPS)	Applies to control plane API operations.

Limit	Default	Notes
ListChannels maximum results per page	100	Use pagination tokens to retrieve additional results.

Delivery constraints

The following constraints apply to all delivery configurations:

- Deliveries support append-only delivery. Records cannot be updated or deleted at the destination.
- Data delivery does not support cross-Region delivery for either destination type. The Kinesis Data Streams stream and the destination must be in the same Region.
- Streaming table (Apache Iceberg) delivery does not support cross-account delivery. The source stream, the destination S3 table bucket, and AWS Glue Schema Registry must all be in the same AWS account.
- For general purpose Amazon S3 delivery, only the destination bucket can be in a different AWS account. The channel and its source stream must be in the same account.
- No transformations are applied to the data. Records are delivered as-is from the stream to the destination.
- Data delivery does not support schema evolution. Changing the schema requires deleting and recreating the delivery.
- Deliveries do not backfill existing data. Only records written to the stream after the delivery reaches ACTIVE state are delivered.
- Each delivery creates a new table at the destination. You cannot deliver to an existing table.
- The source Kinesis Data Streams stream must be in On-Demand Standard or On-Demand Advantage capacity mode. Provisioned mode is not supported.

Update constraints

Only the following properties can be updated on an existing delivery:

- `DataFreshnessInSeconds`
- `LoggingConfiguration`

All other configuration properties are immutable after creation. To change any other setting, you must delete the existing delivery and create a new one with the desired configuration.

Monitor Kinesis Data Streams

You can monitor your data streams in Amazon Kinesis Data Streams using the following features:

- CloudWatch metrics— Kinesis Data Streams sends Amazon CloudWatch custom metrics with detailed monitoring for each stream.
- Kinesis Agent— The Kinesis Agent publishes custom CloudWatch metrics to help assess if the agent is working as expected.
- API logging— Kinesis Data Streams uses AWS CloudTrail to log API calls and store the data in an Amazon S3 bucket.
- Kinesis Client Library— Kinesis Client Library (KCL) provides metrics per shard, worker, and KCL application.
- Kinesis Producer Library— Amazon Kinesis Producer Library (KPL) provides metrics per shard, worker, and KPL application.

For more information about common monitoring issues, questions, and troubleshooting, see the following:

- Which metrics should I use to monitor and troubleshoot Kinesis Data Streams issues?
- Why does the IteratorAgeMilliseconds value in Kinesis Data Streams keep increasing?

Monitor the Amazon Kinesis Data Streams service with Amazon CloudWatch

Amazon Kinesis Data Streams and Amazon CloudWatch are integrated so that you can collect, view, and analyze CloudWatch metrics for your Kinesis data streams. For example, to track shard usage, you can monitor the `IncomingBytes` and `OutgoingBytes` metrics and compare them to the number of shards in the stream.

Stream metrics and shard-level metrics that you configure are automatically collected and pushed to CloudWatch every minute. Metrics are archived for two weeks; after that period, the data is discarded.

The following table describes basic stream-level and enhanced shard-level monitoring for Kinesis data streams.

Type	Description
Basic (stream-level)	Stream-level data is sent automatically every minute at no charge.
Enhanced (shard-level)	Shard-level data is sent every minute for an additional cost. To get this level of data, you must specifically enable it for the stream using the EnableEnhancedMonitoring operation. For information about pricing, see the Amazon CloudWatch product page.

Amazon Kinesis Data Streams dimensions and metrics

Kinesis Data Streams sends metrics to CloudWatch at two levels: the stream level and, optionally, the shard level. Stream-level metrics are for the most common monitoring use cases in normal conditions. Shard-level metrics are for specific monitoring tasks, usually related to troubleshooting, and are enabled using the EnableEnhancedMonitoring operation.

For an explanation of the statistics gathered from CloudWatch metrics, see CloudWatch Statistics in the *Amazon CloudWatch User Guide*.

Topics

- Basic stream-level metrics
- Enhanced shard-level metrics
- Dimensions for Amazon Kinesis Data Streams metrics
- Recommended Amazon Kinesis Data Streams metrics

Basic stream-level metrics

The `AWS/Kinesis` namespace includes the following stream-level metrics.

Kinesis Data Streams sends these stream-level metrics to CloudWatch every minute. These metrics are always available.

Metric	Description
GetRecords.Bytes	The number of bytes retrieved from the Kinesis stream, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes in a single GetRecords operation for the stream in the specified time period. Shard-level metric name: OutgoingBytes Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
GetRecords.IteratorAge	This metric is no longer used. Use GetRecords.IteratorAgeMilliseconds .
GetRecords.IteratorAgeMilliseconds	The age of the last record in all GetRecords calls made against a Kinesis stream, measured over the specified time period. Age is the difference between the current time and when the last record of the GetRecords call was written to the stream. The Minimum and Maximum statistics can be used to track the progress of Kinesis consumer applications. A value of zero indicates that the records being read are completely caught up with the stream. Shard-level metric name: IteratorAgeMilliseconds Dimensions: StreamName Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds
GetRecords.Latency	The time taken per GetRecords operation, measured over the specified time period.

Metric	Description
	Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds
GetRecords.Records	The number of records retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records in a single GetRecords operation for the stream in the specified time period. Shard-level metric name: OutgoingRecords Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
GetRecords.Success	The number of successful GetRecords operations per stream, measured over the specified time period. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count

Metric	Description
IncomingBytes	The number of bytes successfully put to the Kinesis stream over the specified time period. This metric includes bytes from PutRecord and PutRecords operations. Minimum, Maximum, and Average statistic s represent the bytes in a single put operation for the stream in the specified time period. Shard-level metric name: IncomingBytes Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
IncomingRecords	The number of records successfully put to the Kinesis stream over the specified time period. This metric includes record counts from PutRecord and PutRecords operations. Minimum, Maximum, and Average statistics represent the records in a single put operation for the stream in the specified time period. Shard-level metric name: IncomingRecords Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecord.Bytes	The number of bytes put to the Kinesis stream using the PutRecord operation over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes

Metric	Description
PutRecord.Latency	The time taken per PutRecord operation, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds
PutRecord.Success	The number of successful PutRecord operations per Kinesis stream, measured over the specified time period. Average reflects the percentage of successful writes to a stream. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count
PutRecords.Bytes	The number of bytes put to the Kinesis stream using the PutRecords operation over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
PutRecords.Latency	The time taken per PutRecords operation, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds

Metric	Description
PutRecords.Records	This metric is deprecated. Use PutRecords.SuccessfulRecords . Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.Success	The number of PutRecords operations where at least one record succeeded, per Kinesis stream, measured over the specified time period. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count
PutRecords.TotalRecords	The total number of records sent in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.SuccessfulRecords	The number of successful records in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
PutRecords.FailedRecords	The number of records rejected due to internal failures in a PutRecords operation per Kinesis data stream, measured over the specified time period. Occasional internal failures are to be expected and should be retried. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.ThrottledRecords	The number of records rejected due to throttling in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
ReadProvisionedThroughputExceeded	The number of GetRecords calls throttled for the stream over the specified time period. The most commonly used statistic for this metric is Average. When the Minimum statistic has a value of 1, all records were throttled for the stream during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were throttled for the stream during the specified time period. Shard-level metric name: ReadProvisionedThroughputExceeded Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
SubscribeToShard.RateExceeded	This metric is emitted when a new subscription attempt fails because there already is an active subscription by the same consumer or if you exceed the number of calls per second allowed for this operation. Dimensions: StreamName, ConsumerName
SubscribeToShard.Success	This metric records whether the SubscribeToShard subscription was successfully established. The subscription only lives for at most 5 minutes. Therefore, this metric is emitted at least once every 5 minutes. Dimensions: StreamName, ConsumerName

Metric	Description
SubscribeToShardEvent.Bytes	The number of bytes received from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes published in a single event for the specified time period. Shard-level metric name: OutgoingBytes Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
SubscribeToShardEvent.MillisBehindLatest	The number of milliseconds the read records are from the tip of the stream, indicating how far behind current time the consumer is. Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds
SubscribeToShardEvent.Records	The number of records received from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records in a single event for the specified time period. Shard-level metric name: OutgoingRecords Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
SubscribeToShardEvent.Success	This metric is emitted every time an event is published successfully. It is only emitted when there's an active subscription. Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
WriteProvisionedThroughputExceeded	The number of records rejected due to throttling for the stream over the specified time period. This metric includes throttling from PutRecord and PutRecords operations. The most commonly used statistic for this metric is Average. When the Minimum statistic has a non-zero value, records were being throttled for the stream during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were being throttled for the stream during the specified time period. Shard-level metric name: WriteProvisionedThroughputExceeded Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Enhanced shard-level metrics

The AWS/Kinesis namespace includes the following shard-level metrics.

Kinesis sends the following shard-level metrics to CloudWatch every minute. Each metric dimension creates 1 CloudWatch metric and makes approximately 43,200 `PutMetricData` API calls per month. These metrics are not enabled by default. There is a charge for enhanced metrics emitted from Kinesis. For more information, see Amazon CloudWatch Pricing under the heading *Amazon CloudWatch Custom Metrics*. The charges are given per shard per metric per month.

Metric	Description
`IncomingBytes`	The number of bytes successfully put to the shard over the specified time period. This metric includes bytes from `PutRecord` and `PutRecords` operations. Minimum, Maximum, and Average statistics represent the bytes in a single put operation for the shard in the specified time period. Stream-level metric name: `IncomingBytes` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
`IncomingRecords`	The number of records successfully put to the shard over the specified time period. This metric includes record counts from `PutRecord` and `PutRecords` operations. Minimum, Maximum, and Average statistics represent the records in a single put operation for the shard in the specified time period. Stream-level metric name: `IncomingRecords` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
`IteratorAgeMilliseconds`	The age of the last record in all `GetRecords` calls made against a shard, measured over the specified time

Metric	Description
	period. Age is the difference between the current time and when the last record of the `GetRecords` call was written to the stream. The Minimum and Maximum statistics can be used to track the progress of Kinesis consumer applications. A value of 0 (zero) indicates that the records being read are completely caught up with the stream. Stream-level metric name: `GetRecords.IteratorAgeMilliseconds` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds
`OutgoingBytes`	The number of bytes retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes returned in a single `GetRecords` operation or published in a single `SubscribeToShard` event for the shard in the specified time period. Stream-level metric name: `GetRecords.Bytes` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes

Metric	Description
OutgoingRecords	The number of records retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records returned in a single GetRecords operation or published in a single SubscribeToShard event for the shard in the specified time period. Stream-level metric name: GetRecords.Records Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
ReadProvisionedThroughputExceeded	The number of GetRecords calls throttled for the shard over the specified time period. This exception count covers all dimensions of the following limits: 5 reads per shard per second or 2 MB per second per shard. The most commonly used statistic for this metric is Average. When the Minimum statistic has a value of 1, all records were throttled for the shard during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were throttled for the shard during the specified time period. Stream-level metric name: ReadProvisionedThroughputExceeded Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
WriteProvisionedThroughputExceeded	The number of records rejected due to throttling for the shard over the specified time period. This metric includes throttling from PutRecord and PutRecords operations and covers all dimensions of the following limits: 1,000 records per second per shard or 1 MB per second per shard. The most commonly used statistic for this metric is Average. When the Minimum statistic has a non-zero value, records were being throttled for the shard during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were being throttled for the shard during the specified time period. Stream-level metric name: WriteProvisionedThroughputExceeded Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Dimensions for Amazon Kinesis Data Streams metrics

Dimension	Description
StreamName	The name of the Kinesis stream. All available statistics are filtered by StreamName .

Recommended Amazon Kinesis Data Streams metrics

Several Amazon Kinesis Data Streams metrics might be of particular interest to Kinesis Data Streams customers. The following list provides recommended metrics and their uses.

Metric	Usage Notes
`GetRecords.IteratorAgeMilliseconds`	Tracks the read position across all shards and consumers in the stream. If an iterator's age passes 50% of the retention period (by default, 24 hours, configurable up to 7 days), there is risk for data loss due to record expiration. We recommend that you use CloudWatch alarms on the Maximum statistic to alert you before this loss is a risk. For an example scenario that uses this metric, see Consumer record processing is falling behind.
`ReadProvisionedThroughputExceeded`	When your consumer-side record processing is falling behind, it is sometimes difficult to know where the bottleneck is. Use this metric to determine if your reads are being throttled due to exceeding your read throughput limits. The most commonly used statistic for this metric is Average.
`WriteProvisionedThroughputExceeded`	This is for the same purpose as the `ReadProvisionedThroughputExceeded` metric, but for the producer (put) side of the stream. The most commonly used statistic for this metric is Average.
`PutRecord.Success`, `PutRecords.Success`	We recommend using CloudWatch alarms on the Average statistic to indicate when records are failing to the stream. Choose one or both put types depending on what your producer uses. If using the Amazon Kinesis Producer Library (KPL), use `PutRecords.Success`.
`GetRecords.Success`	We recommend using CloudWatch alarms on the Average statistic to indicate when records are failing from the stream.

Access Amazon CloudWatch metrics for Kinesis Data Streams

You can monitor metrics for Kinesis Data Streams using the CloudWatch console, the command line, or the CloudWatch API. The following procedures show you how to access metrics using these different methods.

To access metrics using the CloudWatch console

1. Open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
2. On the navigation bar, choose a Region.
3. In the navigation pane, choose **Metrics**.
4. In the **CloudWatch Metrics by Category** pane, choose **Kinesis Metrics**.
5. Click the relevant row to view the statistics for the specified **MetricName** and **StreamName**.

   Note: Most console statistic names match the corresponding CloudWatch metric names listed previously, except for **Read Throughput** and **Write Throughput**. These statistics are calculated over 5-minute intervals: **Write Throughput** monitors the `IncomingBytes` CloudWatch metric, and **Read Throughput** monitors `GetRecords.Bytes`.
6. (Optional) In the graph pane, select a statistic and a time period, and then create a CloudWatch alarm using these settings.

To access metrics using the AWS CLI

Use the list-metrics and get-metric-statistics commands.

To access metrics using the CloudWatch CLI

Use the mon-list-metrics and mon-get-stats commands.

To access metrics using the CloudWatch API

Use the ListMetrics and GetMetricStatistics operations.

Monitor Kinesis Data Streams Agent health with Amazon CloudWatch

The agent publishes custom CloudWatch metrics with a namespace of **AWSKinesisAgent**. These metrics help you assess whether the agent is submitting data into Kinesis Data Streams as specified, and whether it is healthy and consuming the appropriate amount of CPU and memory resources on the data producer. Metrics such as the number of records and bytes sent are useful to understand the rate at which the agent is submitting data to the stream. When these metrics fall below expected thresholds by some percentage or drop to zero, it could indicate configuration issues, network errors, or agent health issues. Metrics such as on-host CPU and memory consumption and agent error counters indicate data producer resource usage, and provide insights into potential configuration or host errors. Finally, the agent also logs service exceptions to help investigate agent issues. These metrics are reported in the Region specified in the agent configuration setting `cloudwatch.endpoint`. CloudWatch metrics published from multiple Kinesis agents are aggregated or combined. For more information about agent configuration, see Specify the agent configuration settings.

Monitor with CloudWatch

The Kinesis Data Streams agent sends the following metrics to CloudWatch.

Metric	Description
`BytesSent`	The number of bytes sent to Kinesis Data Streams over the specified time period. Units: Bytes
`RecordSendAttempts`	The number of records attempted (either first time, or as a retry) in a call to `PutRecords` over the specified time period. Units: Count
`RecordSendErrors`	The number of records that returned failure status in a call to `PutRecords`, including retries, over the specified time period. Units: Count

Metric	Description
ServiceErrors	The number of calls to PutRecords that resulted in a service error (other than a throttling error) over the specified time period. Units: Count

Log Amazon Kinesis Data Streams API calls with AWS CloudTrail

Amazon Kinesis Data Streams is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Kinesis Data Streams. CloudTrail captures all API calls for Kinesis Data Streams as events. The calls captured include calls from the Kinesis Data Streams console and code calls to the Kinesis Data Streams API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Kinesis Data Streams. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Kinesis Data Streams, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Kinesis Data Streams information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in Kinesis Data Streams, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Kinesis Data Streams, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

Kinesis Data Streams supports logging the following actions as events in CloudTrail log files:

- AddTagsToStream
- CreateStream
- DecreaseStreamRetentionPeriod
- DeleteStream
- DeregisterStreamConsumer
- DescribeStream
- DescribeStreamConsumer
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- GetRecords
- GetShardIterator
- IncreaseStreamRetentionPeriod
- ListStreamConsumers
- ListStreams
- ListTagsForStream
- MergeShards
- PutRecord
- PutRecords
- RegisterStreamConsumer
- RemoveTagsFromStream
- SplitShard
- StartStreamEncryption
- StopStreamEncryption

- SubscribeToShard
- UpdateShardCount
- UpdateStreamMode

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 or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Example: Kinesis Data Streams log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateStream`, `DescribeStream`, `ListStreams`, `DeleteStream`, `SplitShard`, and `MergeShards` actions.

```
{
    "Records": [
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:31Z",
            "eventSource": "kinesis.amazonaws.com",
```

```
            "eventName": "CreateStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardCount": 1,
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "db6c59f8-c757-11e3-bc3b-57923b443c1c",
            "eventID": "b7acfcd0-6ca9-4ee1-a3d7-c4e8d420d99b"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:06Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DescribeStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f0944d86-c757-11e3-b4ae-25654b1d3136",
            "eventID": "0b2f1396-88af-4561-b16f-398f8eaea596"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
```

```
            },
            "eventTime": "2014-04-19T00:15:02Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "ListStreams",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "limit": 10
            },
            "responseElements": null,
            "requestID": "a68541ca-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "22a5fb8f-4e61-4bee-a8ad-3b72046b4c4d"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:07Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DeleteStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f10cd97c-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "607e7217-311a-4a08-a904-ec02944596dd"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
```

```
            "accessKeyId": "EXAMPLE_KEY_ID",
            "userName": "Alice"
        },
        "eventTime": "2014-04-19T00:15:03Z",
        "eventSource": "kinesis.amazonaws.com",
        "eventName": "SplitShard",
        "awsRegion": "us-east-1",
        "sourceIPAddress": "127.0.0.1",
        "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
        "requestParameters": {
            "shardToSplit": "shardId-000000000000",
            "streamName": "GoodStream",
            "newStartingHashKey": "11111111"
        },
        "responseElements": null,
        "requestID": "a6e6e9cd-c757-11e3-901b-cbcfe5b3677a",
        "eventID": "dcd2126f-c8d2-4186-b32a-192dd48d7e33"
    },
    {
        "eventVersion": "1.01",
        "userIdentity": {
            "type": "IAMUser",
            "principalId": "EX_PRINCIPAL_ID",
            "arn": "arn:aws:iam::012345678910:user/Alice",
            "accountId": "012345678910",
            "accessKeyId": "EXAMPLE_KEY_ID",
            "userName": "Alice"
        },
        "eventTime": "2014-04-19T00:16:56Z",
        "eventSource": "kinesis.amazonaws.com",
        "eventName": "MergeShards",
        "awsRegion": "us-east-1",
        "sourceIPAddress": "127.0.0.1",
        "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
        "requestParameters": {
            "streamName": "GoodStream",
            "adjacentShardToMerge": "shardId-000000000002",
            "shardToMerge": "shardId-000000000001"
        },
        "responseElements": null,
        "requestID": "e9f9c8eb-c757-11e3-bf1d-6948db3cd570",
        "eventID": "77cf0d06-ce90-42da-9576-71986fec411f"
    }
]
```

```
}
```

Monitor the Kinesis Client Library with Amazon CloudWatch

The Kinesis Client Library (KCL) for Amazon Kinesis Data Streams publishes custom Amazon CloudWatch metrics on your behalf, using the name of your KCL application as the namespace. You can view these metrics by navigating to the CloudWatch console and choosing **Custom Metrics**. For more information about custom metrics, see Publish Custom Metrics in the *Amazon CloudWatch User Guide*.

There is a nominal charge for the metrics uploaded to CloudWatch by the KCL; specifically, *Amazon CloudWatch Custom Metrics* and *Amazon CloudWatch API Requests* charges apply. For more information, see Amazon CloudWatch Pricing.

Topics

- Metrics and namespace
- Metric levels and dimensions
- Metric configuration
- List of metrics

Metrics and namespace

The namespace that is used to upload metrics is the application name that you specify when you launch the KCL.

Metric levels and dimensions

There are two options to control which metrics are uploaded to CloudWatch:

metric levels

> Every metric is assigned an individual level. When you set a metrics reporting level, metrics with an individual level below the reporting level are not sent to CloudWatch. The levels are: `NONE`, `SUMMARY`, and `DETAILED`. The default setting is `DETAILED`; that is, all metrics are sent to CloudWatch. A reporting level of `NONE` means that no metrics are sent at all. For information about which levels are assigned to what metrics, see List of metrics.

enabled dimensions

Every KCL metric has associated dimensions that also get sent to CloudWatch. In KCL 2.x, if KCL is configured to process a single data stream, all the metrics dimensions (`Operation`, `ShardId`, and `WorkerIdentifier`) are enabled by default. Also, in KCL 2.x, if KCL is configured to process a single data stream, `Operation` dimension cannot be disabled. In KCL 2.x, if KCL is configured to process multiple data streams, all the metrics dimensions (`Operation`, `ShardId`, `StreamId`, and `WorkerIdentifier`) are enabled by default. Also, in KCL 2.x, if KCL is configured to process multiple data streams, the `Operation` and the `StreamId` dimensions cannot be disabled. `StreamId` dimension is available only for the per-shard metrics.

In KCL 1.x, only the `Operation` and the `ShardId` dimensions are enabled by default, and the `WorkerIdentifier` dimension is disabled. In KCL 1.x, the `Operation` dimension cannot be disabled.

For more information about CloudWatch metric dimensions, see the Dimensions section in the Amazon CloudWatch Concepts topic, in the *Amazon CloudWatch User Guide*.

When the `WorkerIdentifier` dimension is enabled, if a different value is used for the worker ID property every time a particular KCL worker restarts, new sets of metrics with new `WorkerIdentifier` dimension values are sent to CloudWatch. If you need the `WorkerIdentifier` dimension value to be the same across specific KCL worker restarts, you must explicitly specify the same worker ID value during initialization for each worker. Note that the worker ID value for each active KCL worker must be unique across all KCL workers.

Metric configuration

Metric levels and enabled dimensions can be configured using the KinesisClientLibConfiguration instance, which is passed to Worker when launching the KCL application. In the MultiLangDaemon case, the `metricsLevel` and `metricsEnabledDimensions` properties can be specified in the .properties file used to launch the MultiLangDaemon KCL application.

Metric levels can be assigned one of three values: NONE, SUMMARY, or DETAILED. Enabled dimensions values must be comma-separated strings with the list of dimensions that are allowed for the CloudWatch metrics. The dimensions used by the KCL application are `Operation`, `ShardId`, and `WorkerIdentifier`.

List of metrics

The following tables list the KCL metrics, grouped by scope and operation.

Topics

- Per-KCL-application metrics
- Per-worker metrics
- Per-shard metrics

Per-KCL-application metrics

These metrics are aggregated across all KCL workers within the scope of the application, as defined by the Amazon CloudWatch namespace.

Topics

- LeaseAssignmentManager
- InitializeTask
- ShutdownTask
- ShardSyncTask
- BlockOnParentTask
- PeriodicShardSyncManager
- MultistreamTracker

LeaseAssignmentManager

The `LeaseAssignmentManager` operation is responsible for assigning leases to workers and rebalancing leases among workers to achieve even utilization of worker resources. The logic for this operation includes reading the lease related metadata from the lease table and metrics from the worker metrics table, and performing lease assignments.

Metric	Description
LeaseAndWorkerMetricsLoad.Time	Time taken to load all leases and worker metrics entry in the lease assignment manager (LAM), the new lease assignment and load balancing algorithm introduced in KCL 3.x.

Metric	Description
	Metric level: Detailed Units: Milliseconds
TotalLeases	Total number of leases for the current KCL application. Metric level: Summary Units: Count
NumWorkers	Total number of workers in the current KCL application. Metric level: Summary Units: Count
AssignExpiredOrUnassignedLeases.Time	Time to perform in-memory assignment of expired leases. Metric level: Detailed Units: Milliseconds
LeaseSpillover	Number of leases that were not assigned due to hitting the limit on the maximum number of leases or maximum throughput per worker. Metric level: Summary Units: Count
BalanceWorkerVariance.Time	Time to perform in-memory balancing of leases between workers. Metric level: Detailed Units: Milliseconds
NumOfLeasesReassignment	Total number of lease reassignments made in the current reassignment iteration. Metric level: Summary Units: Count

Metric	Description
FailedAssignmentCount	Number of failures in AssignLease calls to the DynamoDB lease table. Metric level: Detailed Units: Count
ParallelyAssignLeases.Time	Time to flush new assignments to the DynamoDB lease table. Metric level: Detailed Units: Milliseconds
ParallelyAssignLeases.Success	Number of successful flush of new assignments. Metric level: Detailed Units: Count
TotalStaleWorkerMetricsEntry	Total number of worker metrics entries that must be cleaned up. Metric level: Detailed Units: Count
StaleWorkerMetricsCleanup.Time	Time to perform worker metrics entry deletion from the DynamoDB worker metrics table. Metric level: Detailed Units: Milliseconds
Time	Time taken by the `LeaseAssignmentManager` operation. Metric level: Summary Units: Milliseconds

Metric	Description
Success	Number of times the `LeaseAssignmentManager` operation successfully completed. Metric level: Summary Units: Count
ForceLeaderRelease	Indicates that the lease assignment manager has failed 3 times consecutively and the leader worker is releasing the leadership. Metric level: Summary Units: Count
NumWorkersWithInvalidEntry	Number of worker metrics entries which are considered invalid. Metric level: Summary Units: Count
NumWorkersWithFailingWorkerMetric	Number of worker metrics entries which has -1 (representing worker metric value is not available) as one of the value for worker metrics. Metric level: Summary Units: Count
LeaseDeserializationFailureCount	Lease entry from the lease table which failed to deserialize. Metric level: Summary Units: Count

InitializeTask

The `InitializeTask` operation is responsible for initializing the record processor for the KCL application. The logic for this operation includes getting a shard iterator from Kinesis Data Streams and initializing the record processor.

Metric	Description
KinesisDataFetcher .getIterator.Success	Number of successful `GetShardIterator` operations per KCL application. Metric level: Detailed Units: Count
KinesisDataFetcher .getIterator.Time	Time taken per `GetShardIterator` operation for the given KCL application. Metric level: Detailed Units: Milliseconds
RecordProcessor.in itialize.Time	Time taken by the record processor's initialize method. Metric level: Summary Units: Milliseconds
Success	Number of successful record processor initializations. Metric level: Summary Units: Count
Time	Time taken by the KCL worker for the record processor initialization. Metric level: Summary Units: Milliseconds

ShutdownTask

The `ShutdownTask` operation initiates the shutdown sequence for shard processing. This can occur because a shard is split or merged, or when the shard lease is lost from the worker. In both cases, the record processor `shutdown()` function is invoked. New shards are also discovered in the case where a shard was split or merged, resulting in the creation of one or two new shards.

Metric	Description
CreateLease.Success	Number of times that new child shards are successfully added into the KCL application DynamoDB table following parent shard shutdown. Metric level: Detailed Units: Count
CreateLease.Time	Time taken for adding new child shard information in the KCL applicati on DynamoDB table. Metric level: Detailed Units: Milliseconds
UpdateLease.Success	Number of successful final checkpoints during the record processor shutdown. Metric level: Detailed Units: Count
UpdateLease.Time	Time taken by the checkpoint operation during the record processor shutdown. Metric level: Detailed Units: Milliseconds
RecordProcessor.sh utdown.Time	Time taken by the record processor’s shutdown method. Metric level: Summary Units: Milliseconds
Success	Number of successful shutdown tasks. Metric level: Summary Units: Count

Metric	Description
Time	Time taken by the KCL worker for the shutdown task. Metric level: Summary Units: Milliseconds

ShardSyncTask

The `ShardSyncTask` operation discovers changes to shard information for the Kinesis data stream, so new shards can be processed by the KCL application.

Metric	Description
CreateLease.Success	Number of successful attempts to add new shard information into the KCL application DynamoDB table. Metric level: Detailed Units: Count
CreateLease.Time	Time taken for adding new shard information in the KCL application DynamoDB table. Metric level: Detailed Units: Milliseconds
Success	Number of successful shard sync operations. Metric level: Summary Units: Count
Time	Time taken for the shard sync operation. Metric level: Summary Units: Milliseconds

BlockOnParentTask

If the shard is split or merged with other shards, then new child shards are created. The `BlockOnParentTask` operation ensures that record processing for the new shards does not start until the parent shards are completely processed by the KCL.

Metric	Description
Success	Number of successful checks for parent shard completion. Metric level: Summary Units: Count
Time	Time taken for parent shards completion. Metric level: Summary Unit: Milliseconds

PeriodicShardSyncManager

The `PeriodicShardSyncManager` is responsible for examining the data streams that are being processed by the KCL consumer application, identifying data streams with partial leases and handing them off for synchronization.

The following metrics are available when KCL is configured to process a single data stream (then the value of NumStreamsToSync and NumStreamsWithPartialLeases is set to 1) and also when KCL is configured to process multiple data streams.

Metric	Description
NumStreamsToSync	The number of data streams (per AWS account) being processed by the consumer application that contains partial leases and that must be handed off for synchronization. Metric level: Summary Units: Count

Metric	Description
NumStreamsWithPartialLeases	The number of data streams (per AWS account) that the consumer application is processing that contains partial leases. Metric level: Summary Units: Count
Success	The number of times `PeriodicShardSyncManager` was able to successfully identify partial leases in the data streams that the consumer application is processing. Metric level: Summary Units: Count
Time	The amount of the time (in milliseconds) that the `PeriodicShardSyncManager` takes to examine the data streams that the consumer application is processing, in order to determine which data streams require shard synchronization. Metric level: Summary Units: Milliseconds

MultistreamTracker

The `MultistreamTracker` interface enables you to build KCL consumer applications that can process multiple data streams at the same time.

Metric	Description
DeletedStreams.Count	The number of data streams deleted at this time period. Metric level: Summary Units: Count
ActiveStreams.Count	The number of active data streams being processed.

Metric	Description
	Metric level: Summary Units: Count
StreamsPendingDeletion.Count	The number of data streams that are pending deletion based on `FormerStreamsLeasesDeletionStrategy` . Metric level: Summary Units: Count

Per-worker metrics

These metrics are aggregated across all record processors consuming data from a Kinesis data stream, such as an Amazon EC2 instance.

Topics

- WorkerMetricStatsReporter
- LeaseDiscovery
- RenewAllLeases
- TakeLeases

WorkerMetricStatsReporter

The `WorkerMetricStatReporter` operation is responsible for periodically publishing metrics of the current worker to the worker metrics table. These metrics are used by the `LeaseAssignmentManager` operation to perform lease assignments.

Metric	Description
InMemoryMetricStatsReporterFailure	Number of failures to capture the in-memory worker metric value, due to failure of some worker metrics. Metric level: Summary Units: Count

Metric	Description
WorkerMetricStatsReporter.Time	Time taken by the `WorkerMetricsStats` operation. Metric level: Summary Units: Milliseconds
WorkerMetricStatsReporter.Success	Number of times the `WorkerMetricsStats` operation successfully completed. Metric level: Summary Units: Count

LeaseDiscovery

The `LeaseDiscovery` operation is responsible for identifying the new leases assigned to the current worker by the `LeaseAssignmentManager` operation. The logic for this operation involves identifying leases assigned to the current worker by reading the global secondary index of the lease table.

Metric	Description
ListLeaseKeysForWorker.Time	Time to call the global secondary index on the lease table and get lease keys assigned to the current worker. Metric level: Detailed Units: Milliseconds
FetchNewLeases.Time	Time to fetch all new leases from the lease table. Metric level: Detailed Units: Milliseconds
NewLeasesDiscovered	Total number of new leases assigned to workers. Metric level: Detailed

Metric	Description
	Units: Count
Time	Time taken by the `LeaseDiscovery` operation. Metric level: Summary Units: Milliseconds
Success	Number of times the `LeaseDiscovery` operation successfully completed. Metric level: Summary Units: Count
OwnerMismatch	Number of owner mismatches from GSI response and lease table consistent read. Metric level: Detailed Units: Count

RenewAllLeases

The `RenewAllLeases` operation periodically renews shard leases owned by a particular worker instance.

Metric	Description
RenewLease.Success	Number of successful lease renewals by the worker. Metric level: Detailed Units: Count
RenewLease.Time	Time taken by the lease renewal operation. Metric level: Detailed

Metric	Description
	Units: Milliseconds
CurrentLeases	Number of shard leases owned by the worker after all leases are renewed. Metric level: Summary Units: Count
LostLeases	Number of shard leases that were lost following an attempt to renew all leases owned by the worker. Metric level: Summary Units: Count
Success	Number of times the lease renewal operation was successful for the worker. Metric level: Summary Units: Count
Time	Time taken for renewing all leases for the worker. Metric level: Summary Units: Milliseconds

TakeLeases

The `TakeLeases` operation balances record processing between all KCL workers. If the current KCL worker has fewer shard leases than required, it takes shard leases from another worker that is overloaded.

Metric	Description
ListLeases.Success	Number of times all shard leases were successfully retrieved from the KCL application DynamoDB table.

Metric	Description
	Metric level: Detailed Units: Count
ListLeases.Time	Time taken to retrieve all shard leases from the KCL application DynamoDB table. Metric level: Detailed Units: Milliseconds
TakeLease.Success	Number of times the worker successfully took shard leases from other KCL workers. Metric level: Detailed Units: Count
TakeLease.Time	Time taken to update the lease table with leases taken by the worker. Metric level: Detailed Units: Milliseconds
NumWorkers	Total number of workers, as identified by a specific worker. Metric level: Summary Units: Count
NeededLeases	Number of shard leases that the current worker needs for a balanced shard-processing load. Metric level: Detailed Units: Count

Metric	Description
LeasesToTake	Number of leases that the worker will attempt to take. Metric level: Detailed Units: Count
TakenLeases	Number of leases taken successfully by the worker. Metric level: Summary Units: Count
TotalLeases	Total number of shards that the KCL application is processing. Metric level: Detailed Units: Count
ExpiredLeases	Total number of shards that are not being processed by any worker, as identified by the specific worker. Metric level: Summary Units: Count
Success	Number of times the `TakeLeases` operation successfully completed. Metric level: Summary Units: Count
Time	Time taken by the `TakeLeases` operation for a worker. Metric level: Summary Units: Milliseconds

Per-shard metrics

These metrics are aggregated across a single record processor.

ProcessTask

The `ProcessTask` operation calls GetRecords with the current iterator position to retrieve records from the stream and invokes the record processor `processRecords` function.

Metric	Description
KinesisDataFetcher .getRecords.Success	Number of successful `GetRecords` operations per Kinesis data stream shard. Metric level: Detailed Units: Count
KinesisDataFetcher .getRecords.Time	Time taken per `GetRecords` operation for the Kinesis data stream shard. Metric level: Detailed Units: Milliseconds
UpdateLease.Success	Number of successful checkpoints made by the record processor for the given shard. Metric level: Detailed Units: Count
UpdateLease.Time	Time taken for each checkpoint operation for the given shard. Metric level: Detailed Units: Milliseconds
DataBytesProcessed	Total size of records processed in bytes on each `ProcessTask` invocation. Metric level: Summary Units: Byte
RecordsProcessed	Number of records processed on each `ProcessTask` invocation.

Metric	Description
	Metric level: Summary Units: Count
ExpiredIterator	Number of ExpiredIteratorException received when calling `GetRecords`. Metric level: Summary Units: Count
MillisBehindLatest	Time that the current iterator is behind from the latest record (tip) in the shard. This value is less than or equal to the difference in time between the latest record in a response and the current time. This is a more accurate reflection of how far a shard is from the tip than comparing timestamps in the last response record. This value applies to the latest batch of records, not an average of all timestamps in each record. Metric level: Summary Units: Milliseconds
RecordProcessor.processRecords.Time	Time taken by the record processor's `processRecords` method. Metric level: Summary Units: Milliseconds
Success	Number of successful process task operations. Metric level: Summary Units: Count
Time	Time taken for the process task operation. Metric level: Summary Units: Milliseconds

Monitor the Kinesis Producer Library with Amazon CloudWatch

The Amazon Kinesis Producer Library (KPL) for Amazon Kinesis Data Streams publishes custom Amazon CloudWatch metrics on your behalf. You can view these metrics by navigating to the CloudWatch console and choosing **Custom Metrics**. For more information about custom metrics, see Publish Custom Metrics in the *Amazon CloudWatch User Guide*.

There is a nominal charge for the metrics uploaded to CloudWatch by the KPL; specifically, Amazon CloudWatch Custom Metrics and Amazon CloudWatch API Requests charges apply. For more information, see Amazon CloudWatch Pricing. Local metrics gathering does not incur CloudWatch charges.

Topics

- Metrics, dimensions, and namespaces
- Metric level and granularity
- Local access and Amazon CloudWatch upload
- List of metrics

Metrics, dimensions, and namespaces

You can specify an application name when launching the KPL, which is then used as part of the namespace when uploading metrics. This is optional; the KPL provides a default value if an application name is not set.

You can also configure the KPL to add arbitrary additional dimensions to the metrics. This is useful if you want finer-grained data in your CloudWatch metrics. For example, you can add the hostname as a dimension, which then allows you to identify uneven load distributions across your fleet. All KPL configuration settings are immutable, so you can't change these additional dimensions after the KPL instance is initialized.

Metric level and granularity

There are two options to control the number of metrics uploaded to CloudWatch:

metric level

This is a rough gauge of how important a metric is. Every metric is assigned a level. When you set a level, metrics with levels below that are not sent to CloudWatch. The levels are `NONE`,

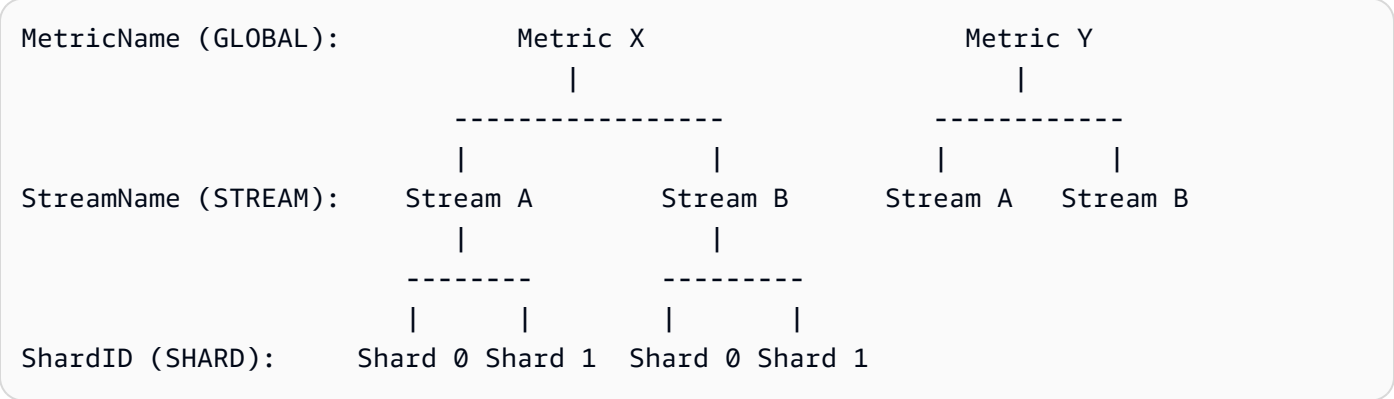
SUMMARY, and DETAILED. The default setting is DETAILED; that is, all metrics. NONE means no metrics at all, so no metrics are actually assigned to that level.

granularity

This controls whether the same metric is emitted at additional levels of granularity. The levels are GLOBAL, STREAM, and SHARD. The default setting is SHARD, which contains the most granular metrics.

When SHARD is chosen, metrics are emitted with the stream name and shard ID as dimensions. In addition, the same metric is also emitted with only the stream name dimension, and the metric without the stream name. This means that, for a particular metric, two streams with two shards each will produce seven CloudWatch metrics: one for each shard, one for each stream, and one overall; all describing the same statistics but at different levels of granularity. For an illustration, see the following diagram.

The different granularity levels form a hierarchy, and all the metrics in the system form trees, rooted at the metric names:



Not all metrics are available at the shard level; some are stream level or global by nature. These are not produced at the shard level, even if you have enabled shard-level metrics (`Metric Y` in the preceding diagram).

When you specify an additional dimension, you must provide values for `tuple:<DimensionName, DimensionValue, Granularity>`. The granularity is used to determine where the custom dimension is inserted in the hierarchy: GLOBAL means that the additional dimension is inserted after the metric name, STREAM means it's inserted after the stream name, and SHARD means it's inserted after the shard ID. If multiple additional dimensions are given per granularity level, they are inserted in the order given.

Local access and Amazon CloudWatch upload

Metrics for the current KPL instance are available locally in real time; you can query the KPL at any time to get them. The KPL locally computes the sum, average, minimum, maximum, and count of every metric, as in CloudWatch.

You can get statistics that are cumulative from the start of the program to the present point in time, or using a rolling window over the past *N* seconds, where *N* is an integer between 1 and 60.

All metrics are available for upload to CloudWatch. This is especially useful for aggregating data across multiple hosts, monitoring, and alarming. This functionality is not available locally.

As described previously, you can select which metrics to upload with the *metric level* and *granularity* settings. Metrics that are not uploaded are available locally.

Uploading data points individually is untenable because it could produce millions of uploads per second, if traffic is high. For this reason, the KPL aggregates metrics locally into 1-minute buckets and uploads a statistics object to CloudWatch one time per minute, per enabled metric.

List of metrics

Metric	Description
`UserRecordsReceived`	Count of how many logical user records were received by the KPL core for put operations. Not available at shard level. Metric level: Detailed Unit: Count
`UserRecordsPending`	Periodic sample of how many user records are currently pending. A record is pending if it is either currently buffered and waiting to be sent, or sent and in-flight to the backend service. Not available at shard level. The KPL provides a dedicated method to retrieve this metric at the global level for customers to manage their put rate. Metric level: Detailed

Metric	Description
	Unit: Count
UserRecordsPut	Count of how many logical user records were put successfully. The KPL outputs a zero for failed records. This allows the average to give the success rate, the count to give the total attempts, and the difference between the count and sum to give the failure count. Metric level: Summary Unit: Count
UserRecordsDataPut	Bytes in the logical user records successfully put. Metric level: Detailed Unit: Bytes
KinesisRecordsPut	Count of how many Kinesis Data Streams records were put successfully (each Kinesis Data Streams record can contain multiple user records). The KPL outputs a zero for failed records. This allows the average to give the success rate, the count to give the total attempts, and the difference between the count and sum to give the failure count. Metric level: Summary Unit: Count
KinesisRecordsDataPut	Bytes in the Kinesis Data Streams records. Metric level: Detailed Unit: Bytes

Metric	Description
ErrorsByCode	Count of each type of error code. This introduces an additional dimension of ErrorCode , in addition to the normal dimensions such as StreamName and ShardId. Not every error can be traced to a shard. The errors that cannot be traced are only emitted at stream or global levels. This metric captures information about such things as throttling, shard map changes, internal failures, service unavailable, timeouts, and so on. Kinesis Data Streams API errors are counted one time per Kinesis Data Streams record. Multiple user records within a Kinesis Data Streams record do not generate multiple counts. Metric level: Summary Unit: Count
AllErrors	This is triggered by the same errors as **Errors by Code**, but does not distinguish between types. This is useful as a general monitor of the error rate without requiring a manual sum of the counts from all the different types of errors. Metric level: Summary Unit: Count
RetriesPerRecord	Number of retries performed per user record. Zero is emitted for records that succeed in one try. Data is emitted at the moment a user record finishes (when it either succeeds or can no longer be retried). If record time-to-live is a large value, this metric may be significantly delayed. Metric level: Detailed Unit: Count

Metric	Description
BufferingTime	The time between a user record arriving at the KPL and leaving for the backend. This information is transmitted back to the user on a per-record basis, but is also available as an aggregated statistic. Metric level: Summary Unit: Milliseconds
Request Time	The time it takes to perform PutRecordsRequests . Metric level: Detailed Unit: Milliseconds
User Records per Kinesis Record	The number of logical user records aggregated into a single Kinesis Data Streams record. Metric level: Detailed Unit: Count
Amazon Kinesis Records per PutRecordsRequest	The number of Kinesis Data Streams records aggregated into a single PutRecordsRequest . Not available at shard level. Metric level: Detailed Unit: Count
User Records per PutRecordsRequest	The total number of user records contained within a PutRecordsRequest . This is roughly equivalent to the product of the previous two metrics. Not available at shard level. Metric level: Detailed Unit: Count

Security in Amazon Kinesis Data Streams

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Kinesis Data Streams, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Kinesis Data Streams. The following topics show you how to configure Kinesis Data Streams to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Kinesis Data Streams resources.

Topics

- Data protection in Amazon Kinesis Data Streams
- Controlling access to Amazon Kinesis Data Streams resources using IAM
- Test your permissions and request inputs with dry run
- Compliance validation for Amazon Kinesis Data Streams
- Resilience in Amazon Kinesis Data Streams
- Infrastructure Security in Amazon Kinesis Data Streams
- Security best practices for Kinesis Data Streams

Data protection in Amazon Kinesis Data Streams

Server-side encryption using AWS Key Management Service (AWS KMS) keys makes it easy for you to meet strict data management requirements by encrypting your data at rest within Amazon Kinesis Data Streams.

> **Note**
>
> If you require FIPS 140-2 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) 140-2.

Topics

- What is server-side encryption for Kinesis Data Streams?
- Costs, Regions, and performance considerations
- How do I get started with server-side encryption?
- Create and use user-generated KMS keys
- Permissions to use user-generated KMS keys
- Verify and Troubleshoot KMS key permissions
- Use Amazon Kinesis Data Streams with interface VPC endpoints

What is server-side encryption for Kinesis Data Streams?

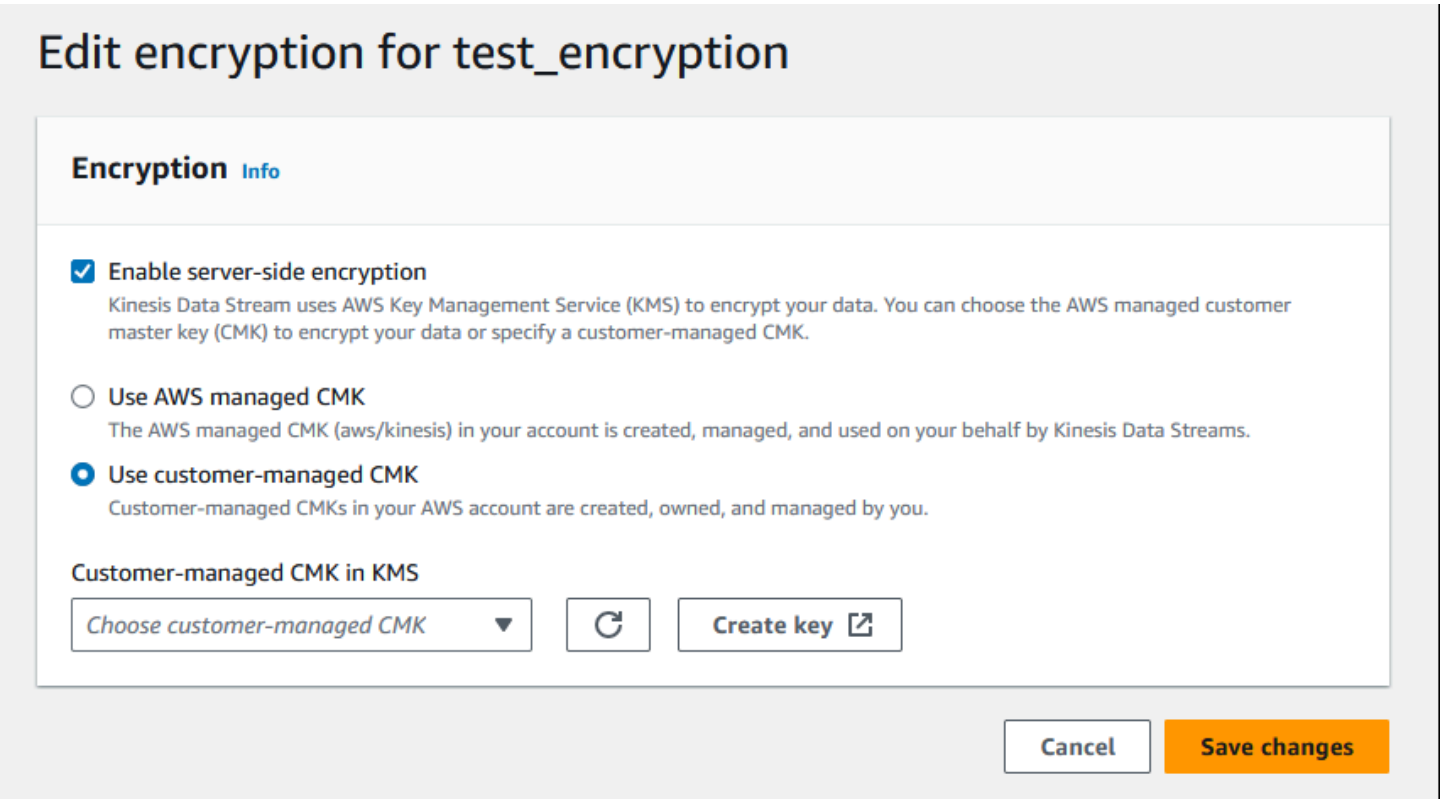
Server-side encryption is a feature in Amazon Kinesis Data Streams that automatically encrypts data before it's at rest by using an AWS KMS customer master key (CMK) you specify. Data is encrypted before it's written to the Kinesis stream storage layer, and decrypted after it’s retrieved from storage. As a result, your data is encrypted at rest within the Kinesis Data Streams service. This allows you to meet strict regulatory requirements and enhance the security of your data.

With server-side encryption, your Kinesis stream producers and consumers don't need to manage master keys or cryptographic operations. Your data is automatically encrypted as it enters and leaves the Kinesis Data Streams service, so your data at rest is encrypted. AWS KMS provides all the master keys that are used by the server-side encryption feature. AWS KMS makes it easy to use a CMK for Kinesis that is managed by AWS, a user-specified AWS KMS CMK, or a master key imported into the AWS KMS service.

> **Note**
>
> Server-side encryption encrypts incoming data only after encryption is enabled. Preexisting data in an unencrypted stream is not encrypted after server-side encryption is enabled.

When encrypting your data streams and sharing access to other principals, you must grant permission in both the key policy for the AWS KMS key and the IAM policies in the external account. For more information, see Allowing users in other accounts to use a KMS key.

If you have enabled server-side encryption for a data stream with AWS managed KMS key and want to share access via a resource policy, you must switch to using customer-managed key (CMK), as shown following:



In addition, you must allow your sharing principal entities to have access to your CMK, using KMS cross account sharing capabilities. Make sure to also make the change in the IAM policies for the sharing principal entities. For more information, see Allowing users in other accounts to use a KMS key.

Costs, Regions, and performance considerations

When you apply server-side encryption, you are subject to AWS KMS API usage and key costs. Unlike custom KMS master keys, the `(Default) aws/kinesis` customer master key (CMK) is offered free of charge. However, you still must pay for the API usage costs that Amazon Kinesis Data Streams incurs on your behalf.

API usage costs apply for every CMK, including custom ones. Kinesis Data Streams calls AWS KMS approximately every five minutes when it is rotating the data key. In a 30-day month, the total cost of AWS KMS API calls that are initiated by a Kinesis stream should be less than a few dollars. This cost scales with the number of user credentials that you use on your data producers and consumers because each user credential requires a unique API call to AWS KMS. When you use an IAM role for authentication, each assume role call results in unique user credentials. To save KMS costs, you might want to cache user credentials that are returned by the assume role call.

The following describes the costs by resource:

Keys

- The CMK for Kinesis that's managed by AWS (alias = `aws/kinesis`) is free.
- User-generated KMS keys are subject to KMS key costs. For more information, see AWS Key Management Service Pricing.

API usage costs apply for every CMK, including custom ones. Kinesis Data Streams calls KMS approximately every 5 minutes when it is rotating the data key. In a 30-day month, the total cost of KMS API calls initiated by a Kinesis data stream should be less than a few dollars. Please note that this cost scales with the number of user credentials you use on your data producers and consumers because each user credential requires a unique API call to AWS KMS. When you use IAM role for authentication, each assume-role-call will result in unique user credentials and you might want to cache user credentials returned by the assume-role-call to save KMS costs.

KMS API usage

For every encrypted stream, when reading from TIP and using a single IAM account/user access key across readers and writers, Kinesis service calls the AWS KMS service approximately 12 times every 5 minutes. Not reading from TIP could lead to higher calls to AWS KMS service. API requests to generate new data encryption keys are subject to AWS KMS usage costs. For more information, see AWS Key Management Service Pricing: Usage.

Availability of server-side encryption by Region

Currently, server-side encryption of Kinesis streams is available in all the Regions supported for Kinesis Data Streams, including AWS GovCloud (US-West), and the China Regions. For more information about supported Regions for Kinesis Data Streams see https://docs.aws.amazon.com/general/latest/gr/ak.html.

Performance Considerations

Due to the service overhead of applying encryption, applying server-side encryption increases the typical latency of `PutRecord`, `PutRecords`, and `GetRecords` by less than 100µs.

How do I get started with server-side encryption?

The easiest way to get started with server-side encryption is to use the AWS Management Console and the Amazon Kinesis KMS Service Key, `aws/kinesis`.

The following procedure demonstrates how to enable server-side encryption for a Kinesis stream.

To enable server-side encryption for a Kinesis stream

1. Sign in to the AWS Management Console and open the Amazon Kinesis Data Streams console.
2. Create or select a Kinesis stream in the AWS Management Console.
3. Choose the **details** tab.
4. In **Server-side encryption**, choose **edit**.
5. Unless you want to use a user-generated KMS master key, ensure the **(Default) aws/kinesis** KMS master key is selected. This is the KMS master key generated by the Kinesis service. Choose **Enabled**, and then choose **Save**.

   > **Note**
   >
   > The default Kinesis service master key is free, however, the API calls made by Kinesis to the AWS KMS service are subject to KMS usage costs.

6. The stream transitions through a **pending** state. After the stream returns to an **active** state with encryption enabled, all incoming data written to the stream is encrypted using the KMS master key you selected.
7. To disable server-side encryption, choose **Disabled** in **Server-side encryption** in the AWS Management Console, and then choose **Save**.

Create and use user-generated KMS keys

This section describes how to create and use your own KMS keys, instead of using the master key administered by Amazon Kinesis.

Create user-generated KMS keys

For instructions on creating your own keys, see Creating Keys in the *AWS Key Management Service Developer Guide*. After you create keys for your account, the Kinesis Data Streams service returns these keys in the **KMS master key** list.

Use user-generated KMS keys

After the correct permissions are applied to your consumers, producers, and administrators, you can use custom KMS keys in your own AWS account or another AWS account. All KMS master keys in your account appear in the **KMS Master Key** list within the AWS Management Console.

To use custom KMS master keys located in another account, you need permissions to use those keys. You must also specify the ARN of the KMS master key in the ARN input box in the AWS Management Console.

Permissions to use user-generated KMS keys

Before you can use server-side encryption with a user-generated KMS key, you must configure AWS KMS key policies to allow encryption of streams and encryption and decryption of stream records. For examples and more information about AWS KMS permissions, see AWS KMS API Permissions: Actions and Resources Reference.

> **Note**
>
> The use of the default service key for encryption does not require application of custom IAM permissions.

Before you use user-generated KMS master keys, ensure that your Kinesis stream producers and consumers (IAM principals) are users in the KMS master key policy. Otherwise, writes and reads from a stream will fail, which could ultimately result in data loss, delayed processing, or hung applications. You can manage permissions for KMS keys using IAM policies. For more information, see Using IAM Policies with AWS KMS.

Kinesis Data Streams encryption context

When Amazon Kinesis Data Streams calls AWS KMS on your behalf, it passes an encryption context to AWS KMS that can be used as a condition for authorization in key policies and grants. Kinesis Data Streams uses the stream ARN as the encryption context in all AWS KMS calls.

```
"encryptionContext": {
    "aws:kinesis:arn": "arn:aws:kinesis:region:account-id:stream/stream-name"
}
```

You can use the encryption context to identify the use of your KMS key in audit records and logs. It also appears in plaintext in logs, such as AWS CloudTrail.

To limit the use of your KMS key to requests from Kinesis Data Streams for a specific stream, use the `kms:EncryptionContext:aws:kinesis:arn` condition key in the KMS key policy or IAM policy.

Example producer permissions

Your Kinesis stream producers must have the `kms:GenerateDataKey` permission.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:GenerateDataKey"
        ],
        "Resource": "arn:aws-eusc:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:PutRecord",
            "kinesis:PutRecords"
        ],
```

```
        "Resource": "arn:aws-eusc:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

Example consumer permissions

Your Kinesis stream consumers must have the `kms:Decrypt` permission.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:Decrypt"
        ],
        "Resource": "arn:aws-eusc:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:GetRecords",
            "kinesis:DescribeStream"
        ],
        "Resource": "arn:aws-eusc:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

Amazon Managed Service for Apache Flink and AWS Lambda use roles to consume Kinesis streams. Make sure to add the `kms:Decrypt` permission to the roles that these consumers use.

Stream administrator permissions

Kinesis stream administrators must have authorization to call `kms:List*` and `kms:DescribeKey*`.

Verify and Troubleshoot KMS key permissions

After enabling encryption on a Kinesis stream, we recommend that you monitor the success of your `putRecord`, `putRecords`, and `getRecords` calls using the following Amazon CloudWatch metrics:

- `PutRecord.Success`
- `PutRecords.Success`
- `GetRecords.Success`

For more information, see Monitor Kinesis Data Streams

Use Amazon Kinesis Data Streams with interface VPC endpoints

You can use an interface VPC endpoint to prevent traffic between your Amazon VPC and Kinesis Data Streams from leaving the Amazon network. Interface VPC endpoints don't require an internet gateway, NAT device, VPN connection, or Direct Connect connection. Interface VPC endpoints are powered by AWS PrivateLink, an AWS technology that enables private communication between AWS services using an elastic network interface with private IPs in your Amazon VPC. For more information, see Amazon Virtual Private Cloud and Interface VPC Endpoints (AWS PrivateLink).

Topics

- Use interface VPC endpoints for Kinesis Data Streams
- Control access to VPC endpoints for Kinesis Data Streams
- Availability of VPC endpoint policies for Kinesis Data Streams

Use interface VPC endpoints for Kinesis Data Streams

To get started, you do not need to change the settings for your streams, producers, or consumers. Create an interface VPC endpoint for your Kinesis Data Streams to start traffic flowing from and to your Amazon VPC resources through the interface VPC endpoint. FIPS-enabled interface VPC endpoints are available for US Regions. For more information, see Creating an Interface Endpoint.

The Amazon Kinesis Producer Library (KPL) and Kinesis Consumer Library (KCL) call AWS services like Amazon CloudWatch and Amazon DynamoDB using either public endpoints or private interface VPC endpoints, whichever are in use. For example, if your KCL application is running in a VPC

with DynamoDB interface with VPC endpoints enabled, calls between DynamoDB and your KCL application flow through the interface VPC endpoint.

Control access to VPC endpoints for Kinesis Data Streams

VPC endpoint policies let you control access by either attaching a policy to a VPC endpoint or by using additional fields in a policy that is attached to an IAM user, group, or role to restrict access to occur only through the specified VPC endpoint. Use these policies to restrict access to specific streams to a specified VPC endpoint when using them together with the IAM policies to grant only access to Kinesis data stream actions through the specified VPC endpoint.

The following are example endpoint policies for accessing Kinesis data streams.

- **VPC policy example: read-only access** - this sample policy can be attached to a VPC endpoint. (For more information, see Controlling Access to Amazon VPC Resources). It restricts actions to only listing and describing a Kinesis data stream through the VPC endpoint to which it is attached.

```
{
  "Statement": [
    {
      "Sid": "ReadOnly",
      "Principal": "*",
      "Action": [
        "kinesis:List*",
        "kinesis:Describe*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

- **VPC policy example: restrict access to a specific Kinesis data stream** - this sample policy can be attached to a VPC endpoint. It restricts access to a specific data stream through the VPC endpoint to which it is attached.

```
{
  "Statement": [
    {
```

```
      "Sid": "AccessToSpecificDataStream",
      "Principal": "*",
      "Action": "kinesis:*",
      "Effect": "Allow",
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream"
    }
  ]
}
```

- **IAM policy example: restrict access to a specific stream from a specific VPC endpoint only** - this sample policy can be attached to an IAM user, role, or group. It restricts access to a specified Kinesis data stream to occur only from a specified VPC endpoint.

  JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Sid": "AccessFromSpecificEndpoint",
         "Action": "kinesis:*",
         "Effect": "Deny",
         "Resource": "arn:aws-eusc:kinesis:eusc-de-east-1:123456789012:stream/
MyStream",
         "Condition": { "StringNotEquals" : { "aws:sourceVpce":
 "vpce-11aa22bb" } }
      }
   ]
}
```

Availability of VPC endpoint policies for Kinesis Data Streams

Kinesis Data Streams interface VPC endpoints with policies are supported in the following Regions:

- Europe (Paris)
- Europe (Ireland)
- US East (N. Virginia)
- Europe (Stockholm)

- US East (Ohio)
- Europe (Frankfurt)
- South America (São Paulo)
- Europe (London)
- Asia Pacific (Tokyo)
- US West (N. California)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- China (Beijing)
- China (Ningxia)
- Asia Pacific (Hong Kong)
- Middle East (Bahrain)
- Middle East (UAE)
- Europe (Milan)
- Africa (Cape Town)
- Asia Pacific (Mumbai)
- Asia Pacific (Seoul)
- Canada (Central)
- US West (Oregon) except usw2-az4
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)
- Asia Pacific (Osaka)
- Europe (Zurich)
- Asia Pacific (Hyderabad)

Controlling access to Amazon Kinesis Data Streams resources using IAM

AWS Identity and Access Management (IAM) enables you to do the following:

- Create users and groups under your AWS account
- Assign unique security credentials to each user under your AWS account
- Control each user's permissions to perform tasks using AWS resources
- Allow the users in another AWS account to share your AWS resources
- Create roles for your AWS account and define the users or services that can assume them
- Use existing identities for your enterprise to grant permissions to perform tasks using AWS resources

By using IAM with Kinesis Data Streams, you can control whether users in your organization can perform a task using specific Kinesis Data Streams API actions and whether they can use specific AWS resources.

If you are developing an application using the Kinesis Client Library (KCL), your policy must include permissions for Amazon DynamoDB and Amazon CloudWatch; the KCL uses DynamoDB to track state information for the application, and CloudWatch to send KCL metrics to CloudWatch on your behalf. For more information about the KCL, see Develop KCL 1.x consumers.

For more information about IAM, see the following:

- AWS Identity and Access Management (IAM)
- Getting started with IAM
- IAM User Guide

For more information about IAM and Amazon DynamoDB, see Using IAM to Control Access to Amazon DynamoDB Resources in the *Amazon DynamoDB Developer Guide*.

For more information about IAM and Amazon CloudWatch, see Controlling User Access to Your AWS Account in the *Amazon CloudWatch User Guide*.

Contents

- Policy syntax
- Actions for Kinesis Data Streams
- Amazon Resource Names (ARNs) for Kinesis Data Streams
- Example policies for Kinesis Data Streams

- Share your data stream with another account
- Configure an AWS Lambda function to read from Kinesis Data Streams in another account
- Share access using resource-based policies
- Access control for streaming tables and Amazon S3 delivery

Policy syntax

An IAM policy is a JSON document that consists of one or more statements. Each statement is structured as follows:

```
{
  "Statement":[{
    "Effect":"effect",
    "Action":"action",
    "Resource":"arn",
    "Condition":{
      "condition":{
        "key":"value"
        }
      }
    }
  ]
}
```

There are various elements that make up a statement:

- **Effect:** The *effect* can be `Allow` or `Deny`. By default, IAM users don't have permission to use resources and API actions, so all requests are denied. An explicit allow overrides the default. An explicit deny overrides any allows.
- **Action**: The *action* is the specific API action for which you are granting or denying permission.
- **Resource**: The resource that's affected by the action. To specify a resource in the statement, you need to use its Amazon Resource Name (ARN).
- **Condition**: Conditions are optional. They can be used to control when your policy will be in effect.

As you create and manage IAM policies, you might want to use the IAM Policy Generator and the IAM Policy Simulator.

Actions for Kinesis Data Streams

In an IAM policy statement, you can specify any API action from any service that supports IAM. For Kinesis Data Streams, use the following prefix with the name of the API action: `kinesis:`. For example: `kinesis:CreateStream`, `kinesis:ListStreams`, and `kinesis:DescribeStreamSummary`.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": ["kinesis:action1", "kinesis:action2"]
```

You can also specify multiple actions using wildcards. For example, you can specify all actions whose name begins with the word "Get" as follows:

```
"Action": "kinesis:Get*"
```

To specify all Kinesis Data Streams operations, use the * wildcard as follows:

```
"Action": "kinesis:*"
```

For the complete list of Kinesis Data Streams API actions, see the Amazon Kinesis API Reference.

Amazon Resource Names (ARNs) for Kinesis Data Streams

Each IAM policy statement applies to the resources that you specify using their ARNs.

Use the following ARN resource format for Kinesis data streams:

```
arn:aws-eusc:kinesis:region:account-id:stream/stream-name
```

For example:

```
"Resource": arn:aws-eusc:kinesis:*:111122223333:stream/my-stream
```

Example policies for Kinesis Data Streams

The following example policies demonstrate how you could control user access to your Kinesis data streams.

Example 1: Allow users to get data from a stream

Example

This policy allows a user or group to perform the `DescribeStreamSummary`, `GetShardIterator`, and `GetRecords` operations on the specified stream and `ListStreams` on any stream. This policy could be applied to users who should be able to get data from a specific stream.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:Get*",
                "kinesis:DescribeStreamSummary"
            ],
            "Resource": [
            "arn:aws-eusc:kinesis:eusc-de-east-1:111122223333:stream/stream1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Example 2: Allow users to add data to any stream in the account

Example

This policy allows a user or group to use the `PutRecord` operation with any of the account's streams. This policy could be applied to users that should be able to add data records to all streams in an account.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord"
            ],
            "Resource": [
                "arn:aws-eusc:kinesis:eusc-de-east-1:111122223333:stream/*"
            ]
        }
    ]
}
```

Example 3: Allow any Kinesis Data Streams action on a specific stream

Example

This policy allows a user or group to use any Kinesis Data Streams operation on the specified stream. This policy could be applied to users that should have administrative control over a specific stream.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": "kinesis:*",
            "Resource": [
            "arn:aws-eusc:kinesis:eusc-de-east-1:111122223333:stream/stream1"
            ]
        }
    ]
}
```

Example 4: Allow any Kinesis Data Streams action on any stream

Example

This policy allows a user or group to use any Kinesis Data Streams operation on any stream in an account. Because this policy grants full access to all your streams, you should restrict it to administrators only.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws-eusc:kinesis:*:111122223333:stream/*"
            ]
        }
    ]
}
```

Share your data stream with another account

Note

Kinesis Producer Library currently does not support specifying a stream ARN when writing to a data stream. Use the AWS SDK if you want to write to a cross-account data stream.

Attach a resource-based policy to your data stream to grant access to another account, IAM user, or IAM role. Resource-based policies are JSON policy documents that you attach to a resource such as a data stream. These policies grant the specified principal permission to perform specific actions on that resource and define under what conditions this applies. A policy can have multiple statements. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services. You can configure policies in the Kinesis Data Streams console, API or SDK.

Note that sharing access to registered consumers such as Enhanced Fan Out requires a policy on both the data stream ARN and the consumer ARN.

Enable cross-account access

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in separate AWS accounts, you must also use an identity-based policy to grant the principal access to the resource. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required.

For more information about using resource-based policies for cross-account access, see Cross account resource access in IAM.

Data stream administrators can use AWS Identity and Access Management policies to specify who has access to what. That is, which *principal* can perform *actions* on what *resources*, and under what *conditions*. The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation.

Kinesis Data Streams actions that can be shared:

Action	Level of access
DescribeStreamConsumer	Consumer
DescribeStreamSummary	Data stream

Action	Level of access
GetRecords	Data stream
GetShardIterator	Data stream
ListShards	Data stream
PutRecord	Data stream
PutRecords	Data stream
SubscribeToShard	Consumer

Following are examples of using a resource-based policy to grant cross-account access to your data stream or registered consumer.

To perform a cross-account action, you must specify the stream ARN for data stream access and the consumer ARN for registered consumer access.

Example resource-based policies for Kinesis data streams

Sharing a registered consumer involves both a data stream policy and a consumer policy due to the actions needed.

Note

Following are examples of valid values for `Principal`:

- `{"AWS": "123456789012"}`
- IAM User – `{"AWS": "arn:aws:iam::123456789012:user/user-name"}`
- IAM Role – `{"AWS":["arn:aws:iam::123456789012:role/role-name"]}`
- Multiple Principals (can be combination of account, user, role) – `{"AWS": ["123456789012", "123456789013", "arn:aws:iam::123456789012:user/user-name"]}`

Example 1: Write access to the data stream

Example

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_write_policy_ID",
    "Statement": [
        {
            "Sid": "writestatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "Account12345"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards",
                "kinesis:PutRecord",
                "kinesis:PutRecords"
            ],
            "Resource": "arn:aws-eusc:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

Example 2: Read access to the data stream

Example

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
        {
            "Sid": "sharedthroughputreadstatement",
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": "Account12345"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards",
                "kinesis:GetRecords",
                "kinesis:GetShardIterator"
            ],
            "Resource": "arn:aws-eusc:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
    ]
}
```

Example 3: Share enhanced fan-out read access to a registered consumer

Example

Data stream policy statement:

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_sharedthroughput_read_policy_ID",
    "Statement": [
        {
            "Sid": "consumerreadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws-eusc:iam::111122223333:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamSummary",
                "kinesis:ListShards"
            ],
            "Resource": "arn:aws-eusc:kinesis:us-east-2:123456789012:stream/
datastreamABC"
        }
```

```
    ]
}
```

Consumer policy statement:

JSON

```
{
    "Version":"2012-10-17",
    "Id": "__default_efo_read_policy_ID",
    "Statement": [
        {
            "Sid": "eforeadstatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws-eusc:iam::111122223333:role/role-name"
            },
            "Action": [
                "kinesis:DescribeStreamConsumer",
                "kinesis:SubscribeToShard"
            ],
            "Resource": "arn:aws-eusc:kinesis:us-east-2:123456789012:stream/
datastreamABC/consumer/consumerDEF:1674696300"
        }
    ]
}
```

Wildcard (*) is not supported for actions or principal field in order maintain the principle of least privilege..

Manage the policy for your data stream programmatically

Outside of the AWS Management Console, Kinesis Data Streams has three APIS for managing your data stream policy:

- PutResourcePolicy
- GetResourcePolicy
- DeleteResourcePolicy

Use `PutResourePolicy` to attach or overwrite a policy for a data stream or consumer. Use `GetResourcePolicy` to check and view a policy for the specified data stream or consumer. Use `DeleteResourcePolicy` to delete a policy for the specified data stream or consumer.

Policy limits

Kinesis Data Streams resource policies have the following restrictions:

- Wildcards (*) are not supported to help prevent broad access from being granted through the resource policies that are directly attached to a data stream or registered consumer. In addition, carefully inspect the following policies to confirm that they do not grant broad access:
  - Identity-based policies attached to associated AWS principals (for example, IAM roles)
  - Resource-based policies attached to associated AWS resources (for example, AWS Key Management Service KMS keys)
- AWS Service Principals are not supported for principals to prevent potential confused deputies.
- Federated principals are not supported.
- Canonical user IDs are not supported.
- The size of the policy cannot exceed 20KB.

Share access to encrypted data

If you have enabled server-side encryption for a data stream with AWS managed KMS key and want to share access via a resource policy, you must switch to using customer-managed key (CMK). For more information, see What is server-side encryption for Kinesis Data Streams?. In addition, you must allow your sharing principal entities to have access to your CMK, using KMS cross account sharing capabilities. Make sure to also make the change in the IAM policies for the sharing principal entities. For more information, see Allowing users in other accounts to use a KMS key.

Configure an AWS Lambda function to read from Kinesis Data Streams in another account

For an example of how to configure a Lambda function to read from Kinesis Data Streams in another account, see Share access with cross-account AWS Lambda functions.

Share access using resource-based policies

> **Note**
>
> Updating an existing resource-based policy means replacing the existing one, so make sure to include all the necessary information in your new policy.

Share access with cross-account AWS Lambda functions

Lambda operator

1. Go to the IAM console to create an IAM role that will be used as the Lambda execution role for your AWS Lambda function. Add the managed IAM policy `AWSLambdaKinesisExecutionRole` which has the required Kinesis Data Streams and Lambda invocation permissions. This policy also grants access to all potential Kinesis Data Streams resources you may have access to.
2. In the AWS Lambda console, create an AWS Lambda function to process records in an Kinesis Data Streams data stream and during the setup for the execution role, choose the role you created in the previous step.
3. Provide the execution role to the Kinesis Data Streams resource owner for configuring the resource policy.
4. Finish setting up the Lambda function.

Kinesis Data Streams resource owner

1. Get the cross-account Lambda execution role that will invoke the Lambda function.
2. On the Amazon Kinesis Data Streams console, choose the data stream. Choose the **Data stream sharing** tab and then the **Create sharing policy** button to start the visual policy editor. To share a registered consumer within a data stream, choose the consumer and then choose **Create sharing policy**. You can also write the JSON policy directly.
3. Specify the cross-account Lambda execution role as the principal and the exact Kinesis Data Streams actions you are sharing access to. Make sure to include the action `kinesis:DescribeStream`. For more information on example resource policies for Kinesis Data Streams, see Example resource-based policies for Kinesis data streams.
4. Choose **Create policy** or use the PutResourcePolicy to attach the policy to your resource.

Share access with cross-account KCL consumers

- If you are using KCL 1.x, ensure you are using KCL 1.15.0 or higher.
- If you are using KCL 2.x, ensure you are using KCL 2.5.3 or higher.

KCL operator

1. Provide your IAM user or IAM role that will run the KCL application to the resource owner.
2. Ask the resource owner for the data stream or consumer ARN.
3. Ensure that you specify the provided stream ARN as a part of your KCL configuration.
   - For KCL 1.x: use the KinesisClientLibConfiguration constructor and provide the stream ARN.
   - For KCL 2.x: You can provide just the stream ARN or StreamTracker to the Kinesis Client Library ConfigsBuilder. For StreamTracker, provide the stream ARN and creation Epoch from the DynamoDB Lease Table that is generated by the library. If you want to read from a shared registered consumer like Enhanced Fan-Out, use StreamTracker and also provide the consumer ARN.

Kinesis Data Streams resource owner

1. Get the cross-account IAM user or IAM role that will run the KCL application.
2. On the Amazon Kinesis Data Streams console, choose the data stream. Choose the **Data stream sharing** tab and then the **Create sharing policy** button to start the visual policy editor. To share a registered consumer within a data stream, choose the consumer and then choose **Create sharing policy**. You can also write the JSON policy directly.
3. Specify the cross-account KCL application's IAM user or IAM role as the principal and the exact Kinesis Data Streams actions you are sharing access to. For more information on example resource policies for Kinesis Data Streams, see Example resource-based policies for Kinesis data streams.
4. Choose **Create policy** or use the PutResourcePolicy to attach the policy to your resource.

Share access to encrypted data

If you have enabled server-side encryption for a data stream with AWS managed KMS key and want to share access via a resource policy, you must switch to using customer-managed key (CMK). For more information, see What is server-side encryption for Kinesis Data Streams?. In addition, you

must allow your sharing principal entities to have access to your CMK, using KMS cross account sharing capabilities. Make sure to also make the change in the IAM policies for the sharing principal entities. For more information, see Allowing users in other accounts to use a KMS key.

Access control for streaming tables and Amazon S3 delivery

Streaming tables and Amazon S3 delivery use an IAM service execution role that Amazon Kinesis Data Streams assumes to write data to your destination. Follow these access control practices:

- **Least privilege** – Grant only the minimum permissions the service execution role needs. Scope Amazon S3 permissions to specific buckets and prefixes rather than using wildcards.
- **Scope to specific resources** – Restrict permissions to the specific bucket, table bucket, or table that the delivery writes to. Avoid using `Resource: "*"`.
- **Confused deputy prevention** – Include `aws:SourceArn` and `aws:SourceAccount` conditions in the service execution role trust policy so that only your delivery resources can assume the role.
- **Source stream encryption** – If your stream is encrypted with an AWS managed key (the `aws/kinesis` alias), you cannot create a delivery. Use a customer managed key for stream encryption instead.

For the complete service execution role policies, see the section called "IAM permissions for data delivery". For encryption details and audit logging, see the section called "Security for data delivery".

Test your permissions and request inputs with dry run

You can use the `DryRun` parameter with Kinesis Data Streams data plane APIs to verify that you have the required permissions and that your request parameters are valid. This validation occurs before you run the operation on an Amazon Kinesis Data Streams resource. When you set `DryRun` to true, Kinesis Data Streams validates that you have the required IAM permissions to access your stream, checks your request parameters, and confirms that the target resource exists. It performs these checks without running the operation or reading or writing any data.

If all checks pass, the API returns HTTP 400 with a `DryRunOperationException`, confirming that the request would have succeeded without the `DryRun` parameter. If any check fails, the API returns the same error that the actual operation would return (for example, `AccessDeniedException`). You can receive the following exceptions, among others:

- `DryRunOperationException` – The request would have succeeded without the `DryRun` parameter.
- `AccessDeniedException` – The caller does not have the required IAM permissions for the API action.
- `ResourceNotFoundException` – The specified stream or consumer does not exist.
- `ValidationException` – One or more request parameters are invalid (for example, missing required fields, malformed ARN).
- `InvalidArgumentException` – A parameter value is out of range or not supported (for example, invalid ShardIteratorType, future timestamp).

Requests with `DryRun` enabled are subject to a dedicated throttle limit of 1 transaction per second (TPS) per stream, separate from the stream's normal per-shard throughput limits. This limit is shared across all supported dry-run APIs (`PutRecord`, `PutRecords`, `GetRecords`, `GetShardIterator`, and `SubscribeToShard`) on the same stream. If you exceed this limit, the API returns a `ThrottlingException`.

There is no additional charge for using the `DryRun` parameter. Requests with `DryRun` enabled are billed the same as the equivalent request with `DryRun` disabled. For example, for `PutRecord` and `PutRecords`, you are charged based on the input payload size in Provisioned mode. Any charges for the stream and shard-hours continue to apply. For more information about pricing, see Amazon Kinesis Data Streams pricing.

Note

The `DryRun` parameter validates IAM permissions for the Kinesis Data Streams API actions. It does not validate AWS Key Management Service key permissions for encrypted streams. If your stream uses server-side encryption with a customer managed AWS KMS key, you must separately verify that your producers have `kms:GenerateDataKey` permission and your consumers have `kms:Decrypt` permission on the AWS KMS key. To validate AWS KMS permissions, see Testing your permissions in the AWS Key Management Service Developer Guide.

Supported APIs

The `DryRun` parameter is supported for the following APIs:

- PutRecord
- PutRecords
- GetRecords
- GetShardIterator
- SubscribeToShard

AWS CloudTrail logging

If you enable data events in AWS CloudTrail, AWS CloudTrail logs data plane API calls that use the DryRun parameter set to true. The corresponding CloudTrail event includes the DryRun parameter in the request, and the errorCode field shows DryRunOperationException for successful validations.

Using dry run with the AWS CLI

Specify the --dry-run flag in your AWS CLI commands.

The following example tests PutRecord permissions:

```
aws kinesis put-record \
    --stream-name <your-stream-name> \
    --data <your-data-payload> \
    --partition-key <your-partition-key> \
    --dry-run
```

If the request would have succeeded, the response includes the following message: DryRunOperation validation succeeded while calling PutRecord operation.: Request would have succeeded, but DryRun flag is set.

Using dry run with AWS SDKs

Python (Boto3)

```
import boto3
from botocore.exceptions import ClientError

kinesis = boto3.client('kinesis')
```

```
try:
    kinesis.put_record(
        StreamName='my-stream',
        Data=b'test-data',
        PartitionKey='my-partition-key',
        DryRun=True
    )
except ClientError as e:
    if e.response['Error']['Code'] == 'DryRunOperationException':
        print('SUCCESS: Request would have succeeded')
    else:
        print(f'FAILURE: {e.response["Error"]["Code"]}')
```

Java (AWS SDK for Java 2.x)

```
PutRecordRequest request = PutRecordRequest.builder()
    .streamName("my-stream")
    .data(SdkBytes.fromUtf8String("test-data"))
    .partitionKey("my-partition-key")
    .dryRun(true)
    .build();

try {
    kinesisClient.putRecord(request);
} catch (DryRunOperationException e) {
    System.out.println("SUCCESS: " + e.getMessage());
} catch (AccessDeniedException e) {
    System.out.println("FAILURE: " + e.getMessage());
}
```

Compliance validation for Amazon Kinesis Data Streams

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more

information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in Amazon Kinesis Data Streams

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 Availability 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.

In addition to the AWS global infrastructure, Kinesis Data Streams offers several features to help support your data resiliency and backup needs.

Disaster recovery in Amazon Kinesis Data Streams

Failure can occur at the following levels when you use an Amazon Kinesis Data Streams application to process data from a stream:

- A record processor could fail
- A worker could fail, or the instance of the application that instantiated the worker could fail
- An EC2 instance that is hosting one or more instances of the application could fail

Record processor failure

The worker invokes record processor methods using Java ExecutorService tasks. If a task fails, the worker retains control of the shard that the record processor was processing. The worker starts a new record processor task to process that shard. For more information, see Read throttling.

Worker or application failure

If a worker—or an instance of the Amazon Kinesis Data Streams application—fails, you should detect and handle the situation. For example, if the `Worker.run` method throws an exception, you should catch and handle it.

If the application itself fails, you should detect this and restart it. When the application starts up, it instantiates a new worker, which in turn instantiates new record processors that are automatically assigned shards to process. These could be the same shards that these record processors were processing before the failure, or shards that are new to these processors.

In a situation where the worker or application fails, the failure isn't detected, and there are other instances of the application running on other EC2 instances, workers on these other instances handle the failure. They create additional record processors to process the shards that are no longer being processed by the failed worker. The load on these other EC2 instances increases accordingly.

The scenario described here assumes that although the worker or application has failed, the hosting EC2 instance is still running and is therefore not restarted by an Auto Scaling group.

Amazon EC2 instance failure

We recommend that you run the EC2 instances for your application in an Auto Scaling group. This way, if one of the EC2 instances fails, the Auto Scaling group automatically launches a new instance to replace it. You should configure the instances to launch your Amazon Kinesis Data Streams application at startup.

Infrastructure Security in Amazon Kinesis Data Streams

As a managed service, 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 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.

Security best practices for Kinesis Data Streams

Amazon Kinesis Data Streams provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Implement least privilege access

When granting permissions, you decide who is getting what permissions to which Kinesis Data Streams resources. You enable specific actions that you want to allow on those resources. Therefore you should grant only the permissions that are required to perform a task. Implementing least privilege access is fundamental in reducing security risk and the impact that could result from errors or malicious intent.

Use IAM roles

Producer and client applications must have valid credentials to access Kinesis data streams. You should not store AWS credentials directly in a client application or in an Amazon S3 bucket. These are long-term credentials that are not automatically rotated and could have a significant business impact if they are compromised.

Instead, you should use an IAM role to manage temporary credentials for your producer and client applications to access Kinesis data streams. When you use a role, you don't have to use long-term credentials (such as a user name and password or access keys) to access other resources.

For more information, see the following topics in the *IAM User Guide*:

- IAM Roles
- Common Scenarios for Roles: Users, Applications, and Services

Implement server-side encryption in dependent resources

Data at rest and data in transit can be encrypted in Kinesis Data Streams. For more information, see Data protection in Amazon Kinesis Data Streams.

Use CloudTrail to monitor API calls

Kinesis Data Streams is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Kinesis Data Streams.

Using the information collected by CloudTrail, you can determine the request that was made to Kinesis Data Streams, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information, see the section called “Log Amazon Kinesis Data Streams API calls with AWS CloudTrail”.

Using this service 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 this service, see Code examples for Kinesis using AWS SDKs.

Code examples for Kinesis using AWS SDKs

The following code examples show how to use Kinesis 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.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Basic examples for Kinesis using AWS SDKs
  - Learn the basics of Kinesis with an AWS SDK
  - Actions for Kinesis using AWS SDKs
    - Use AddTagsToStream with an AWS SDK or CLI
    - Use CreateStream with an AWS SDK or CLI
    - Use DeleteStream with an AWS SDK or CLI
    - Use DeregisterStreamConsumer with an AWS SDK or CLI
    - Use DescribeStream with an AWS SDK or CLI
    - Use GetRecords with an AWS SDK or CLI
    - Use GetShardIterator with a CLI
    - Use ListStreamConsumers with an AWS SDK
    - Use ListStreams with an AWS SDK or CLI
    - Use ListTagsForStream with an AWS SDK or CLI
    - Use PutRecord with an AWS SDK or CLI
    - Use PutRecords with an AWS SDK or CLI
    - Use RegisterStreamConsumer with an AWS SDK or CLI
- Serverless examples for Kinesis
  - Invoke a Lambda function from a Kinesis trigger

- Reporting batch item failures for Lambda functions with a Kinesis trigger

Basic examples for Kinesis using AWS SDKs

The following code examples show how to use the basics of Amazon Kinesis with AWS SDKs.

Examples

- Learn the basics of Kinesis with an AWS SDK
- Actions for Kinesis using AWS SDKs
  - Use AddTagsToStream with an AWS SDK or CLI
  - Use CreateStream with an AWS SDK or CLI
  - Use DeleteStream with an AWS SDK or CLI
  - Use DeregisterStreamConsumer with an AWS SDK or CLI
  - Use DescribeStream with an AWS SDK or CLI
  - Use GetRecords with an AWS SDK or CLI
  - Use GetShardIterator with a CLI
  - Use ListStreamConsumers with an AWS SDK
  - Use ListStreams with an AWS SDK or CLI
  - Use ListTagsForStream with an AWS SDK or CLI
  - Use PutRecord with an AWS SDK or CLI
  - Use PutRecords with an AWS SDK or CLI
  - Use RegisterStreamConsumer with an AWS SDK or CLI

Learn the basics of Kinesis with an AWS SDK

The following code example shows how to:

- Create a stream and put a record in it.
- Create a shard iterator.
- Read the record, then clean up resources.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   DATA lo_stream_describe_result TYPE REF TO /aws1/cl_knsdescrstreamoutput.
   DATA lo_stream_description TYPE REF TO /aws1/cl_knsstreamdescription.
   DATA lo_sharditerator TYPE REF TO /aws1/cl_knsgetsharditerator01.
   DATA lo_record_result TYPE REF TO /aws1/cl_knsputrecordoutput.

   "Create stream."
   TRY.
       lo_kns->createstream(
           iv_streamname = iv_stream_name
           iv_shardcount = iv_shard_count ).
       MESSAGE 'Stream created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceeded
exception.' TYPE 'E'.
     CATCH /aws1/cx_knsresourceinuseex.
       MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
   ENDTRY.

   "Wait for stream to becomes active."
   lo_stream_describe_result = lo_kns->describestream( iv_streamname =
iv_stream_name ).
   lo_stream_description = lo_stream_describe_result->get_streamdescription( ).
   WHILE lo_stream_description->get_streamstatus( ) <> 'ACTIVE'.
     IF sy-index = 30.
       EXIT.               "maximum 5 minutes"
     ENDIF.
     WAIT UP TO 10 SECONDS.
```

```
      lo_stream_describe_result = lo_kns->describestream( iv_streamname =
 iv_stream_name ).
      lo_stream_description = lo_stream_describe_result-
>get_streamdescription( ).
    ENDWHILE.

    "Create record."
    TRY.
        lo_record_result = lo_kns->putrecord(
            iv_streamname = iv_stream_name
            iv_data       = iv_data
            iv_partitionkey = iv_partition_key ).
        MESSAGE 'Record created.' TYPE 'I'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knskmsaccessdeniedex.
        MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
 'E'.
      CATCH /aws1/cx_knskmsdisabledex.
        MESSAGE 'KMS key used is disabled.' TYPE 'E'.
      CATCH /aws1/cx_knskmsinvalidstateex.
        MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
 quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
 data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Create a shard iterator in order to read the record."
    TRY.
        lo_sharditerator = lo_kns->getsharditerator(
          iv_shardid = lo_record_result->get_shardid( )
          iv_sharditeratortype = iv_sharditeratortype
          iv_streamname = iv_stream_name ).
        MESSAGE 'Shard iterator created.' TYPE 'I'.
      CATCH /aws1/cx_knsinvalidargumentex.
```

```
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_sgmresourcenotfound.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Read the record."
    TRY.
        oo_result = lo_kns->getrecords(                    " oo_result is
returned for testing purposes. "
            iv_shariterator   = lo_shariterator->get_shariterator( ) ).
        MESSAGE 'Shard iterator created.' TYPE 'I'.
      CATCH /aws1/cx_knsexpirediteratorex.
        MESSAGE 'Iterator expired.' TYPE 'E'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knskmsaccessdeniedex.
        MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
      CATCH /aws1/cx_knskmsdisabledex.
        MESSAGE 'KMS key used is disabled.' TYPE 'E'.
      CATCH /aws1/cx_knskmsinvalidstateex.
        MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.

    "Delete stream."
    TRY.
        lo_kns->deletestream(
            iv_streamname = iv_stream_name ).
        MESSAGE 'Stream deleted.' TYPE 'I'.
```

```
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceeded
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourceinuseex.
        MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
    ENDTRY.
```

- For API details, see the following topics in *AWS SDK for SAP ABAP API reference*.
  - CreateStream
  - DeleteStream
  - GetRecords
  - GetShardIterator
  - PutRecord

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Actions for Kinesis using AWS SDKs

The following code examples demonstrate how to perform individual Kinesis actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Amazon Kinesis API Reference.

Examples

- Use AddTagsToStream with an AWS SDK or CLI
- Use CreateStream with an AWS SDK or CLI
- Use DeleteStream with an AWS SDK or CLI
- Use DeregisterStreamConsumer with an AWS SDK or CLI
- Use DescribeStream with an AWS SDK or CLI
- Use GetRecords with an AWS SDK or CLI
- Use GetShardIterator with a CLI

- Use ListStreamConsumers with an AWS SDK
- Use ListStreams with an AWS SDK or CLI
- Use ListTagsForStream with an AWS SDK or CLI
- Use PutRecord with an AWS SDK or CLI
- Use PutRecords with an AWS SDK or CLI
- Use RegisterStreamConsumer with an AWS SDK or CLI

Use `AddTagsToStream` with an AWS SDK or CLI

The following code examples show how to use `AddTagsToStream`.

.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 System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// This example shows how to apply key/value pairs to an Amazon Kinesis
/// stream.
/// </summary>
public class TagStream
{
    public static async Task Main()
    {
        IAmazonKinesis client = new AmazonKinesisClient();

        string streamName = "AmazonKinesisStream";
        var tags = new Dictionary<string, string>
```

```
            {
                { "Project", "Sample Kinesis Project" },
                { "Application", "Sample Kinesis App" },
            };

            var success = await ApplyTagsToStreamAsync(client, streamName, tags);

            if (success)
            {
                Console.WriteLine($"Taggs successfully added to {streamName}.");
            }
            else
            {
                Console.WriteLine("Tags were not added to the stream.");
            }
        }

        /// <summary>
        /// Applies the set of tags to the named Kinesis stream.
        /// </summary>
        /// <param name="client">The initialized Kinesis client.</param>
        /// <param name="streamName">The name of the Kinesis stream to which
        /// the tags will be attached.</param>
        /// <param name="tags">A sictionary containing key/value pairs which
        /// will be used to create the Kinesis tags.</param>
        /// <returns>A Boolean value which represents the success or failure
        /// of AddTagsToStreamAsync.</returns>
        public static async Task<bool> ApplyTagsToStreamAsync(
            IAmazonKinesis client,
            string streamName,
            Dictionary<string, string> tags)
        {
            var request = new AddTagsToStreamRequest
            {
                StreamName = streamName,
                Tags = tags,
            };

            var response = await client.AddTagsToStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- For API details, see AddTagsToStream in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To add tags to a data stream

The following `add-tags-to-stream` example assigns a tag with the key `samplekey` and value `example` to the specified stream.

```
aws kinesis add-tags-to-stream \
    --stream-name samplestream \
    --tags samplekey=example
```

This command produces no output.

For more information, see Tagging Your Streams in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see AddTagsToStream in *AWS CLI Command Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateStream` with an AWS SDK or CLI

The following code examples show how to use `CreateStream`.

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.

```
    using System;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// This example shows how to create a new Amazon Kinesis stream.
    /// </summary>
    public class CreateStream
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();

            string streamName = "AmazonKinesisStream";
            int shardCount = 1;

            var success = await CreateNewStreamAsync(client, streamName,
shardCount);
            if (success)
            {
                Console.WriteLine($"The stream, {streamName} successfully
created.");
            }
        }

        /// <summary>
        /// Creates a new Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client.</param>
        /// <param name="streamName">The name for the new stream.</param>
        /// <param name="shardCount">The number of shards the new stream will
```

```
        /// use. The throughput of the stream is a function of the number of
        /// shards; more shards are required for greater provisioned
        /// throughput.</param>
        /// <returns>A Boolean value indicating whether the stream was created.</
returns>
        public static async Task<bool> CreateNewStreamAsync(IAmazonKinesis
 client, string streamName, int shardCount)
        {
            var request = new CreateStreamRequest
            {
                StreamName = streamName,
                ShardCount = shardCount,
            };

            var response = await client.CreateStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- For API details, see CreateStream in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To create a data stream

The following `create-stream` example creates a data stream named samplestream with 3 shards.

```
aws kinesis create-stream \
    --stream-name samplestream \
    --shard-count 3
```

This command produces no output.

For more information, see Creating a Stream in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see CreateStream 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.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.CreateStreamRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;

/**
 * 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 CreateDataStream {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream (for example,
 StockTradeStream).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }
```

```
        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();
        createStream(kinesisClient, streamName);
        System.out.println("Done");
        kinesisClient.close();
    }

    public static void createStream(KinesisClient kinesisClient, String
 streamName) {
        try {
            CreateStreamRequest streamReq = CreateStreamRequest.builder()
                    .streamName(streamName)
                    .shardCount(1)
                    .build();

            kinesisClient.createStream(streamReq);

        } catch (KinesisException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see CreateStream in *AWS SDK for Java 2.x API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Creates a new stream. By default this cmdlet returns no output so the -PassThru switch is added to return the value supplied to the -StreamName parameter for subsequent use.

```
$streamName = New-KINStream -StreamName "mystream" -ShardCount 1 -PassThru
```

- For API details, see CreateStream in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Creates a new stream.

```
New-KINStream -StreamName "mystream" -ShardCount 1
```

- For API details, see CreateStream in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def create(self, name, wait_until_exists=True):
        """
        Creates a stream.

        :param name: The name of the stream.
        :param wait_until_exists: When True, waits until the service reports that
                                  the stream exists, then queries for its
 metadata.
        """
        try:
```

```
            self.kinesis_client.create_stream(StreamName=name, ShardCount=1)
            self.name = name
            logger.info("Created stream %s.", name)
            if wait_until_exists:
                logger.info("Waiting until exists.")
                self.stream_exists_waiter.wait(StreamName=name)
                self.describe(name)
        except ClientError:
            logger.exception("Couldn't create stream %s.", name)
            raise
```

- For API details, see CreateStream in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn make_stream(client: &Client, stream: &str) -> Result<(), Error> {
    client
        .create_stream()
        .stream_name(stream)
        .shard_count(4)
        .send()
        .await?;

    println!("Created stream");

    Ok(())
}
```

- For API details, see CreateStream in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        lo_kns->createstream(
            iv_streamname = iv_stream_name
            iv_shardcount = iv_shard_count ).
        MESSAGE 'Stream created.' TYPE 'I'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourceinuseex.
        MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
    ENDTRY.
```

- For API details, see CreateStream in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteStream` with an AWS SDK or CLI

The following code examples show how to use `DeleteStream`.

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.

```
using System;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// Shows how to delete an Amazon Kinesis stream.
/// </summary>
public class DeleteStream
{
    public static async Task Main()
    {
        IAmazonKinesis client = new AmazonKinesisClient();
        string streamName = "AmazonKinesisStream";

        var success = await DeleteStreamAsync(client, streamName);

        if (success)
        {
            Console.WriteLine($"Stream, {streamName} successfully deleted.");
        }
        else
        {
            Console.WriteLine("Stream not deleted.");
        }
    }

    /// <summary>
    /// Deletes a Kinesis stream.
    /// </summary>
    /// <param name="client">An initialized Kinesis client object.</param>
    /// <param name="streamName">The name of the string to delete.</param>
```

```
        /// <returns>A Boolean value representing the success of the operation.</
returns>
        public static async Task<bool> DeleteStreamAsync(IAmazonKinesis client,
 string streamName)
        {
            // If EnforceConsumerDeletion is true, any consumers
            // of this stream will also be deleted. If it is set
            // to false and this stream has any consumers, the
            // call will fail with a ResourceInUseException.
            var request = new DeleteStreamRequest
            {
                StreamName = streamName,
                EnforceConsumerDeletion = true,
            };

            var response = await client.DeleteStreamAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- For API details, see DeleteStream in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To delete a data stream

The following `delete-stream` example deletes the specified data stream.

```
aws kinesis delete-stream \
    --stream-name samplestream
```

This command produces no output.

For more information, see Deleting a Stream in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see DeleteStream 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.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.DeleteStreamRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;

/**
 * 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 DeleteDataStream {

    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream (for example,
 StockTradeStream)
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }
```

```
        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();

        deleteStream(kinesisClient, streamName);
        kinesisClient.close();
        System.out.println("Done");
    }

    public static void deleteStream(KinesisClient kinesisClient, String
 streamName) {
        try {
            DeleteStreamRequest delStream = DeleteStreamRequest.builder()
                    .streamName(streamName)
                    .build();

            kinesisClient.deleteStream(delStream);

        } catch (KinesisException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see DeleteStream in *AWS SDK for Java 2.x API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Deletes the specified stream. You are prompted for confirmation before the command executes. To suppress confirmation prompting use the -Force switch.

```
Remove-KINStream -StreamName "mystream"
```

- For API details, see DeleteStream in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Deletes the specified stream. You are prompted for confirmation before the command executes. To suppress confirmation prompting use the -Force switch.

```
Remove-KINStream -StreamName "mystream"
```

- For API details, see DeleteStream in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def delete(self):
        """
        Deletes a stream.
        """
        try:
            self.kinesis_client.delete_stream(StreamName=self.name)
            self._clear()
            logger.info("Deleted stream %s.", self.name)
        except ClientError:
```

```
            logger.exception("Couldn't delete stream %s.", self.name)
            raise
```

- For API details, see DeleteStream in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn remove_stream(client: &Client, stream: &str) -> Result<(), Error> {
    client.delete_stream().stream_name(stream).send().await?;

    println!("Deleted stream.");

    Ok(())
}
```

- For API details, see DeleteStream in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
```

```
        lo_kns->deletestream(
            iv_streamname = iv_stream_name ).
        MESSAGE 'Stream deleted.' TYPE 'I'.
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourceinuseex.
        MESSAGE 'The request processing has failed because the resource is in
use.' TYPE 'E'.
    ENDTRY.
```

- For API details, see DeleteStream in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeregisterStreamConsumer` with an AWS SDK or CLI

The following code examples show how to use `DeregisterStreamConsumer`.

.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 System;
using System.Threading.Tasks;
using Amazon.Kinesis;
using Amazon.Kinesis.Model;

/// <summary>
/// Shows how to deregister a consumer from an Amazon Kinesis stream.
/// </summary>
public class DeregisterConsumer
```

```
    {
        public static async Task Main(string[] args)
        {
            IAmazonKinesis client = new AmazonKinesisClient();

            string streamARN = "arn:aws:kinesis:us-west-2:000000000000:stream/
AmazonKinesisStream";
            string consumerName = "CONSUMER_NAME";
            string consumerARN = "arn:aws:kinesis:us-west-2:000000000000:stream/
AmazonKinesisStream/consumer/CONSUMER_NAME:000000000000";

            var success = await DeregisterConsumerAsync(client, streamARN,
 consumerARN, consumerName);

            if (success)
            {
                Console.WriteLine($"{consumerName} successfully deregistered.");
            }
            else
            {
                Console.WriteLine($"{consumerName} was not successfully
 deregistered.");
            }
        }

        /// <summary>
        /// Deregisters a consumer from a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamARN">The ARN of a Kinesis stream.</param>
        /// <param name="consumerARN">The ARN of the consumer.</param>
        /// <param name="consumerName">The name of the consumer.</param>
        /// <returns>A Boolean value representing the success of the operation.</
returns>
        public static async Task<bool> DeregisterConsumerAsync(
            IAmazonKinesis client,
            string streamARN,
            string consumerARN,
            string consumerName)
        {
            var request = new DeregisterStreamConsumerRequest
            {
                StreamARN = streamARN,
                ConsumerARN = consumerARN,
```

```
                ConsumerName = consumerName,
            };

            var response = await client.DeregisterStreamConsumerAsync(request);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
    }
```

- For API details, see DeregisterStreamConsumer in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To deregister a data stream consumer

The following `deregister-stream-consumer` example deregisters the specified consumer from the specified data stream.

```
aws kinesis deregister-stream-consumer \
    --stream-arn arn:aws:kinesis:us-west-2:123456789012:stream/samplestream \
    --consumer-name KinesisConsumerApplication
```

This command produces no output.

For more information, see Developing Consumers with Enhanced Fan-Out Using the Kinesis Data Streams API in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see DeregisterStreamConsumer in *AWS CLI Command Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeStream` with an AWS SDK or CLI

The following code examples show how to use `DescribeStream`.

CLI

AWS CLI

To describe a data stream

The following `describe-stream` example returns the details of the specified data stream.

```
aws kinesis describe-stream \
    --stream-name samplestream
```

Output:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "113427455640312821154458202477256070484"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49600871682957036442365024926191073437251060580128653314"
                }
            },
            {
                "ShardId": "shardId-000000000001",
                "HashKeyRange": {
                    "StartingHashKey": "113427455640312821154458202477256070485",
                    "EndingHashKey": "226854911280625642308916404954512140969"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49600871682979337187563555549332609155523708941634633746"
                }
            },
            {
                "ShardId": "shardId-000000000002",
                "HashKeyRange": {
                    "StartingHashKey": "226854911280625642308916404954512140970",
                    "EndingHashKey": "340282366920938463463374607431768211455"
```

```
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "496008716830016379327620861724741448737963573031406141 78"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/
samplestream",
        "StreamName": "samplestream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1572297168.0
    }
}
```

For more information, see Creating and Managing Streams in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see DescribeStream in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns details of the specified stream.

```
Get-KINStream -StreamName "mystream"
```

Output:

```
HasMoreShards        : False
RetentionPeriodHours : 24
Shards               : {}
StreamARN            : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName           : mystream
```

```
StreamStatus         : ACTIVE
```

- For API details, see DescribeStream in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns details of the specified stream.

```
Get-KINStream -StreamName "mystream"
```

Output:

```
HasMoreShards        : False
RetentionPeriodHours : 24
Shards               : {}
StreamARN            : arn:aws:kinesis:us-west-2:123456789012:stream/mystream
StreamName           : mystream
StreamStatus         : ACTIVE
```

- For API details, see DescribeStream in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
```

```
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def describe(self, name):
        """
        Gets metadata about a stream.

        :param name: The name of the stream.
        :return: Metadata about the stream.
        """
        try:
            response = self.kinesis_client.describe_stream(StreamName=name)
            self.name = name
            self.details = response["StreamDescription"]
            logger.info("Got stream %s.", name)
        except ClientError:
            logger.exception("Couldn't get %s.", name)
            raise
        else:
            return self.details
```

- For API details, see DescribeStream in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn show_stream(client: &Client, stream: &str) -> Result<(), Error> {
    let resp = client.describe_stream().stream_name(stream).send().await?;

    let desc = resp.stream_description.unwrap();

    println!("Stream description:");
    println!("  Name:             {}:", desc.stream_name());
```

```
    println!("  Status:            {:?}", desc.stream_status());
    println!("  Open shards:       {:?}", desc.shards.len());
    println!("  Retention (hours): {}", desc.retention_period_hours());
    println!("  Encryption:        {:?}", desc.encryption_type.unwrap());

    Ok(())
}
```

- For API details, see DescribeStream in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_kns->describestream(
            iv_streamname = iv_stream_name ).
        DATA(lt_stream_description) = oo_result->get_streamdescription( ).
        MESSAGE 'Streams retrieved.' TYPE 'I'.
      CATCH /aws1/cx_knslimitexceededex.
        MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- For API details, see DescribeStream in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetRecords` with an AWS SDK or CLI

The following code examples show how to use `GetRecords`.

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

CLI

AWS CLI

To obtain records from a shard

The following `get-records` example gets data records from a Kinesis data stream's shard using the specified shard iterator.

```
aws kinesis get-records \
    --shard-iterator AAAAAAAAAAF7/0mWD7IuHj1yGv/
TKuNgx2ukD5xipCY4cy4gU96orWwZwcSXh3K9tAmGYeOZyLZrvzzeOFVf9iN99hUPw/w/
b0YWYeehfNvnf1DYt5XpDJghLKr3DzgznkTmMymDP3R+3wRKeuEw6/kdxY2yKJH0veaiekaVc4N2VwK/
GvaGP2Hh9Fg7N++q0Adg6fIDQPt4p8RpavDbk+A4sL9SWGE1
```

Output:

```
{
    "Records": [],
    "MillisBehindLatest": 80742000
}
```

For more information, see Developing Consumers Using the Kinesis Data Streams API with the AWS SDK for Java in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see GetRecords 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.

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamResponse;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamRequest;
import software.amazon.awssdk.services.kinesis.model.Shard;
import software.amazon.awssdk.services.kinesis.model.GetShardIteratorRequest;
import software.amazon.awssdk.services.kinesis.model.GetShardIteratorResponse;
import software.amazon.awssdk.services.kinesis.model.Record;
import software.amazon.awssdk.services.kinesis.model.GetRecordsRequest;
import software.amazon.awssdk.services.kinesis.model.GetRecordsResponse;
import java.util.ArrayList;
import java.util.List;

/**
 * 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 GetRecords {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <streamName>

                Where:
```

```
                    streamName - The Amazon Kinesis data stream to read from (for
example, StockTradeStream).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();

        getStockTrades(kinesisClient, streamName);
        kinesisClient.close();
    }

    public static void getStockTrades(KinesisClient kinesisClient, String
streamName) {
        String shardIterator;
        String lastShardId = null;
        DescribeStreamRequest describeStreamRequest =
DescribeStreamRequest.builder()
                .streamName(streamName)
                .build();

        List<Shard> shards = new ArrayList<>();
        DescribeStreamResponse streamRes;
        do {
            streamRes = kinesisClient.describeStream(describeStreamRequest);
            shards.addAll(streamRes.streamDescription().shards());

            if (shards.size() > 0) {
                lastShardId = shards.get(shards.size() - 1).shardId();
            }
        } while (streamRes.streamDescription().hasMoreShards());

        GetShardIteratorRequest itReq = GetShardIteratorRequest.builder()
                .streamName(streamName)
                .shardIteratorType("TRIM_HORIZON")
                .shardId(lastShardId)
                .build();
```

```
        GetShardIteratorResponse shardIteratorResult =
 kinesisClient.getShardIterator(itReq);
        shardIterator = shardIteratorResult.shardIterator();

        // Continuously read data records from shard.
        List<Record> records;

        // Create new GetRecordsRequest with existing shardIterator.
        // Set maximum records to return to 1000.
        GetRecordsRequest recordsRequest = GetRecordsRequest.builder()
                .shardIterator(shardIterator)
                .limit(1000)
                .build();

        GetRecordsResponse result = kinesisClient.getRecords(recordsRequest);

        // Put result into record list. Result may be empty.
        records = result.records();

        // Print records
        for (Record record : records) {
            SdkBytes byteBuffer = record.data();
            System.out.printf("Seq No: %s - %s%n", record.sequenceNumber(), new
 String(byteBuffer.asByteArray()));
        }
    }
}
```

- For API details, see GetRecords in *AWS SDK for Java 2.x API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example shows how to return and extract data from a series of one or more records. The iterator supplierd to Get-KINRecord determines the starting position of the records to return which in this example are captured into a variable, $records. Each individual record can then be accessed by indexing the $records collection. Assuming the data in the record is UTF-8 encoded text, the final command shows how

you can extract the data from the MemoryStream in the object and return it as text to the console.

```
$records
$records = Get-KINRecord -ShardIterator "AAAAAAAAAAGIc....9VnbiRNaP"
```

Output:

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
0                  AAAAAAAAAAERNIq...uDn11HuUs  {Key1, Key2}
```

```
$records.Records[0]
```

Output:

```
ApproximateArrivalTimestamp Data                    PartitionKey SequenceNumber
--------------------------- ----                    ------------ --------------
3/7/2016 5:14:33 PM         System.IO.MemoryStream Key1
 4955986459776...931586
```

```
[Text.Encoding]::UTF8.GetString($records.Records[0].Data.ToArray())
```

Output:

```
test data from string
```

- For API details, see [GetRecords](#) in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example shows how to return and extract data from a series of one or more records. The iterator supplierd to Get-KINRecord determines the starting position of the records to return which in this example are captured into a variable, $records. Each individual record can then be accessed by indexing the $records collection. Assuming the data in the record is UTF-8 encoded text, the final command shows how you can extract the data from the MemoryStream in the object and return it as text to the console.

```
$records
```

```
$records = Get-KINRecord -ShardIterator "AAAAAAAAAAGIc....9VnbiRNaP"
```

Output:

```
MillisBehindLatest NextShardIterator            Records
------------------ -----------------            -------
0                  AAAAAAAAAAERNIq...uDn11HuUs  {Key1, Key2}
```

```
$records.Records[0]
```

Output:

```
ApproximateArrivalTimestamp Data                    PartitionKey SequenceNumber
--------------------------- ----                    ------------ --------------
3/7/2016 5:14:33 PM         System.IO.MemoryStream Key1
 4955986459776...931586
```

```
[Text.Encoding]::UTF8.GetString($records.Records[0].Data.ToArray())
```

Output:

```
test data from string
```

- For API details, see GetRecords in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
```

```
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def get_records(self, max_records):
        """
        Gets records from the stream. This function is a generator that first
gets
        a shard iterator for the stream, then uses the shard iterator to get
records
        in batches from the stream. The shard iterator can be accessed through
the
        'details' property, which is populated using the 'describe' function of
this class.
        Each batch of records is yielded back to the caller until the specified
        maximum number of records has been retrieved.

        :param max_records: The maximum number of records to retrieve.
        :return: Yields the current batch of retrieved records.
        """
        try:
            response = self.kinesis_client.get_shard_iterator(
                StreamName=self.name,
                ShardId=self.details["Shards"][0]["ShardId"],
                ShardIteratorType="LATEST",
            )
            shard_iter = response["ShardIterator"]
            record_count = 0
            while record_count < max_records:
                response = self.kinesis_client.get_records(
                    ShardIterator=shard_iter, Limit=10
                )
                shard_iter = response["NextShardIterator"]
                records = response["Records"]
                logger.info("Got %s records.", len(records))
                record_count += len(records)
                yield records
        except ClientError:
            logger.exception("Couldn't get records from stream %s.", self.name)
```

```
            raise



    def describe(self, name):
        """
        Gets metadata about a stream.

        :param name: The name of the stream.
        :return: Metadata about the stream.
        """
        try:
            response = self.kinesis_client.describe_stream(StreamName=name)
            self.name = name
            self.details = response["StreamDescription"]
            logger.info("Got stream %s.", name)
        except ClientError:
            logger.exception("Couldn't get %s.", name)
            raise
        else:
            return self.details
```

- For API details, see GetRecords in *AWS SDK for Python (Boto3) API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_kns->getrecords(             " oo_result is returned for
testing purposes. "
            iv_sharditerator = iv_shard_iterator ).
        DATA(lt_records) = oo_result->get_records( ).
        MESSAGE 'Record retrieved.' TYPE 'I'.
```

```
      CATCH /aws1/cx_knsexpirediteratorex.
        MESSAGE 'Iterator expired.' TYPE 'E'.
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_knskmsaccessdeniedex.
        MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
      CATCH /aws1/cx_knskmsdisabledex.
        MESSAGE 'KMS key used is disabled.' TYPE 'E'.
      CATCH /aws1/cx_knskmsinvalidstateex.
        MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
      CATCH /aws1/cx_knskmsnotfoundex.
        MESSAGE 'KMS key used is not found.' TYPE 'E'.
      CATCH /aws1/cx_knskmsoptinrequired.
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- For API details, see GetRecords in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetShardIterator` with a CLI

The following code examples show how to use `GetShardIterator`.

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

CLI

AWS CLI

To obtain a shard iterator

The following `get-shard-iterator` example uses the `AT_SEQUENCE_NUMBER` shard iterator type and generates a shard iterator to start reading data records exactly from the position denoted by the specified sequence number.

```
aws kinesis get-shard-iterator \
    --stream-name samplestream \
    --shard-id shardId-000000000001 \
    --shard-iterator-type LATEST
```

Output:

```
{
    "ShardIterator": "AAAAAAAAAAFEvJjIYI+3jw/4aqgH9FifJ+n48XWTh/
IFIsbILP6o5eDueD39NXNBfpZ10WL5K6ADXk8w+5H+Qhd9cFA9k268CPXCz/kebq1TGYI7Vy
+lUkA9BuN3xvATxMBGxRY3zYK05gqgvaIRn94O8SqeEqwhigwZxNWxID3Ej7YYYcxQi8Q/fIrCjGAy/
n2r5Z9G864YpWDfN9upNNQAR/iiOWKs"
}
```

For more information, see Developing Consumers Using the Kinesis Data Streams API with the AWS SDK for Java in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see GetShardIterator in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Returns a shard iterator for the specified shard and starting position. Details of the shard identifiers and sequence numbers can be obtained from the output of the Get-KINStream cmdlet, by referencing the Shards collection of the returned stream object. The returned iterator can be used with the Get-KINRecord cmdlet to pull data records in the shard.

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

Output:

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- For API details, see GetShardIterator in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Returns a shard iterator for the specified shard and starting position. Details of the shard identifiers and sequence numbers can be obtained from the output of the Get-KINStream cmdlet, by referencing the Shards collection of the returned stream object. The returned iterator can be used with the Get-KINRecord cmdlet to pull data records in the shard.

```
Get-KINShardIterator -StreamName "mystream" -ShardId "shardId-000000000000" -
ShardIteratorType AT_SEQUENCE_NUMBER -StartingSequenceNumber "495598645..."
```

Output:

```
AAAAAAAAAAGIc....9VnbiRNaP
```

- For API details, see GetShardIterator in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListStreamConsumers` with an AWS SDK

The following code example shows how to use `ListStreamConsumers`.

.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 System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// List the consumers of an Amazon Kinesis stream.
    /// </summary>
    public class ListConsumers
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();

            string streamARN = "arn:aws:kinesis:us-east-2:000000000000:stream/
AmazonKinesisStream";
            int maxResults = 10;

            var consumers = await ListConsumersAsync(client, streamARN,
 maxResults);

            if (consumers.Count > 0)
            {
                consumers
                    .ForEach(c => Console.WriteLine($"Name: {c.ConsumerName} ARN:
 {c.ConsumerARN}"));
            }
            else
            {
                Console.WriteLine("No consumers found.");
            }
        }

        /// <summary>
        /// Retrieve a list of the consumers for a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamARN">The ARN of the stream for which we want to
        /// retrieve a list of clients.</param>
        /// <param name="maxResults">The maximum number of results to return.</
param>
        /// <returns>A list of Consumer objects.</returns>
```

```
        public static async Task<List<Consumer>>
ListConsumersAsync(IAmazonKinesis client, string streamARN, int maxResults)
        {
            var request = new ListStreamConsumersRequest
            {
                StreamARN = streamARN,
                MaxResults = maxResults,
            };

            var response = await client.ListStreamConsumersAsync(request);

            return response.Consumers;
        }
    }
```

- For API details, see ListStreamConsumers in *AWS SDK for .NET API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListStreams` with an AWS SDK or CLI

The following code examples show how to use `ListStreams`.

.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 System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Kinesis;
```

```
    using Amazon.Kinesis.Model;

    /// <summary>
    /// Retrieves and displays a list of existing Amazon Kinesis streams.
    /// </summary>
    public class ListStreams
    {
        public static async Task Main(string[] args)
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            var response = await client.ListStreamsAsync(new
ListStreamsRequest());

            List<string> streamNames = response.StreamNames;

            if (streamNames.Count > 0)
            {
                streamNames
                    .ForEach(s => Console.WriteLine($"Stream name: {s}"));
            }
            else
            {
                Console.WriteLine("No streams were found.");
            }
        }
    }
```

- For API details, see ListStreams in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To list data streams

The following `list-streams` example lists all active data streams in the current account and region.

```
aws kinesis list-streams
```

Output:

```
{
    "StreamNames": [
        "samplestream",
        "samplestream1"
    ]
}
```

For more information, see Listing Streams in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see ListStreams in *AWS CLI Command Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn show_streams(client: &Client) -> Result<(), Error> {
    let resp = client.list_streams().send().await?;

    println!("Stream names:");

    let streams = resp.stream_names;
    for stream in &streams {
        println!("  {}", stream);
    }

    println!("Found {} stream(s)", streams.len());

    Ok(())
}
```

- For API details, see ListStreams in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       oo_result = lo_kns->liststreams(        " oo_result is returned for
testing purposes. "
           "Set Limit to specify that a maximum of streams should be returned."
           iv_limit = iv_limit ).
       DATA(lt_streams) = oo_result->get_streamnames( ).
       MESSAGE 'Streams listed.' TYPE 'I'.
     CATCH /aws1/cx_knslimitexceededex.
       MESSAGE 'The request processing has failed because of a limit exceed
exception.' TYPE 'E'.
   ENDTRY.
```

- For API details, see ListStreams in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListTagsForStream` with an AWS SDK or CLI

The following code examples show how to use `ListTagsForStream`.

.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 System;
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// Shows how to list the tags that have been attached to an Amazon Kinesis
    /// stream.
    /// </summary>
    public class ListTags
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            string streamName = "AmazonKinesisStream";

            await ListTagsAsync(client, streamName);
        }

        /// <summary>
        /// List the tags attached to a Kinesis stream.
        /// </summary>
        /// <param name="client">An initialized Kinesis client object.</param>
        /// <param name="streamName">The name of the Kinesis stream for which you
        /// wish to display tags.</param>
        public static async Task ListTagsAsync(IAmazonKinesis client, string
streamName)
        {
            var request = new ListTagsForStreamRequest
            {
                StreamName = streamName,
```

```
                Limit = 10,
            };

            var response = await client.ListTagsForStreamAsync(request);
            DisplayTags(response.Tags);

            while (response.HasMoreTags)
            {
                request.ExclusiveStartTagKey = response.Tags[response.Tags.Count
- 1].Key;
                response = await client.ListTagsForStreamAsync(request);
            }
        }

        /// <summary>
        /// Displays the items in a list of Kinesis tags.
        /// </summary>
        /// <param name="tags">A list of the Tag objects to be displayed.</param>
        public static void DisplayTags(List<Tag> tags)
        {
            tags
                .ForEach(t => Console.WriteLine($"Key: {t.Key} Value:
{t.Value}"));
        }
    }
```

- For API details, see ListTagsForStream in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To list tags for a data stream

The following `list-tags-for-stream` example lists the tags attached to the specified data stream.

```
aws kinesis list-tags-for-stream \
    --stream-name samplestream
```

Output:

```
{
    "Tags": [
        {
            "Key": "samplekey",
            "Value": "example"
        }
    ],
    "HasMoreTags": false
}
```

For more information, see Tagging Your Streams in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see ListTagsForStream in *AWS CLI Command Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutRecord` with an AWS SDK or CLI

The following code examples show how to use `PutRecord`.

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

CLI

AWS CLI

To write a record into a data stream

The following `put-record` example writes a single data record into the specified data stream using the specified partition key.

```
aws kinesis put-record \
    --stream-name samplestream \
    --data sampledatarecord \
```

```
    --partition-key samplepartitionkey
```

Output:

```
{
    "ShardId": "shardId-000000000009",
    "SequenceNumber": "49600902273357540915989931256901506243878407835297513618",
    "EncryptionType": "KMS"
}
```

For more information, see Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see PutRecord 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.

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.awssdk.services.kinesis.model.KinesisException;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamRequest;
import software.amazon.awssdk.services.kinesis.model.DescribeStreamResponse;

/**
 * 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 StockTradesWriter {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <streamName>

                Where:
                    streamName - The Amazon Kinesis data stream to which records
 are written (for example, StockTradeStream)
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String streamName = args[0];
        Region region = Region.US_EAST_1;
        KinesisClient kinesisClient = KinesisClient.builder()
                .region(region)
                .build();

        // Ensure that the Kinesis Stream is valid.
        validateStream(kinesisClient, streamName);
        setStockData(kinesisClient, streamName);
        kinesisClient.close();
    }

    public static void setStockData(KinesisClient kinesisClient, String
 streamName) {
        try {
            // Repeatedly send stock trades with a 100 milliseconds wait in
 between.
            StockTradeGenerator stockTradeGenerator = new StockTradeGenerator();

            // Put in 50 Records for this example.
            int index = 50;
            for (int x = 0; x < index; x++) {
                StockTrade trade = stockTradeGenerator.getRandomTrade();
                sendStockTrade(trade, kinesisClient, streamName);
                Thread.sleep(100);
            }
```

```
        } catch (KinesisException | InterruptedException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Done");
    }

    private static void sendStockTrade(StockTrade trade, KinesisClient
kinesisClient,
            String streamName) {
        byte[] bytes = trade.toJsonAsBytes();

        // The bytes could be null if there is an issue with the JSON
serialization by
        // the Jackson JSON library.
        if (bytes == null) {
            System.out.println("Could not get JSON bytes for stock trade");
            return;
        }

        System.out.println("Putting trade: " + trade);
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(trade.getTickerSymbol()) // We use the ticker
symbol as the partition key, explained in
                                                       // the Supplemental
Information section below.
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(bytes))
                .build();

        try {
            kinesisClient.putRecord(request);
        } catch (KinesisException e) {
            System.err.println(e.getMessage());
        }
    }

    private static void validateStream(KinesisClient kinesisClient, String
streamName) {
        try {
            DescribeStreamRequest describeStreamRequest =
DescribeStreamRequest.builder()
                    .streamName(streamName)
```

```
                    .build();

            DescribeStreamResponse describeStreamResponse =
 kinesisClient.describeStream(describeStreamRequest);

            if (!
describeStreamResponse.streamDescription().streamStatus().toString().equals("ACTIVE"))
 {
                System.err.println("Stream " + streamName + " is not active.
 Please wait a few moments and try again.");
                System.exit(1);
            }

        } catch (KinesisException e) {
            System.err.println("Error found while describing the stream " +
 streamName);
            System.err.println(e);
            System.exit(1);
        }
    }
}
```

- For API details, see PutRecord in *AWS SDK for Java 2.x API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Writes a record containing the string supplied to the -Text parameter.

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

Example 2: Writes a record containing the data contained in the specified file. The file is treated as a sequence of bytes so if it contains text, it should be written with any necessary encoding before using it with this cmdlet.

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
 "Key2"
```

- For API details, see PutRecord in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Writes a record containing the string supplied to the -Text parameter.

```
Write-KINRecord -Text "test data from string" -StreamName "mystream" -
PartitionKey "Key1"
```

Example 2: Writes a record containing the data contained in the specified file. The file is treated as a sequence of bytes so if it contains text, it should be written with any necessary encoding before using it with this cmdlet.

```
Write-KINRecord -FilePath "C:\TestData.txt" -StreamName "mystream" -PartitionKey
 "Key2"
```

- For API details, see PutRecord in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class KinesisStream:
    """Encapsulates a Kinesis stream."""

    def __init__(self, kinesis_client):
        """
        :param kinesis_client: A Boto3 Kinesis client.
        """
        self.kinesis_client = kinesis_client
        self.name = None
        self.details = None
        self.stream_exists_waiter = kinesis_client.get_waiter("stream_exists")


    def put_record(self, data, partition_key):
        """
```

```
        Puts data into the stream. The data is formatted as JSON before it is
passed
        to the stream.

        :param data: The data to put in the stream.
        :param partition_key: The partition key to use for the data.
        :return: Metadata about the record, including its shard ID and sequence
number.
        """
        try:
            response = self.kinesis_client.put_record(
                StreamName=self.name, Data=json.dumps(data),
PartitionKey=partition_key
            )
            logger.info("Put record in stream %s.", self.name)
        except ClientError:
            logger.exception("Couldn't put record in stream %s.", self.name)
            raise
        else:
            return response
```

- For API details, see PutRecord in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
async fn add_record(client: &Client, stream: &str, key: &str, data: &str) ->
 Result<(), Error> {
    let blob = Blob::new(data);

    client
        .put_record()
        .data(blob)
```

```
        .partition_key(key)
        .stream_name(stream)
        .send()
        .await?;

    println!("Put data into stream.");

    Ok(())
}
```

- For API details, see PutRecord in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       oo_result = lo_kns->putrecord(            " oo_result is returned for
testing purposes. "
           iv_streamname = iv_stream_name
           iv_data       = iv_data
           iv_partitionkey = iv_partition_key ).
       MESSAGE 'Record created.' TYPE 'I'.
     CATCH /aws1/cx_knsinvalidargumentex.
       MESSAGE 'The specified argument was not valid.' TYPE 'E'.
     CATCH /aws1/cx_knskmsaccessdeniedex.
       MESSAGE 'You do not have permission to perform this AWS KMS action.' TYPE
'E'.
     CATCH /aws1/cx_knskmsdisabledex.
       MESSAGE 'KMS key used is disabled.' TYPE 'E'.
     CATCH /aws1/cx_knskmsinvalidstateex.
       MESSAGE 'KMS key used is in an invalid state. ' TYPE 'E'.
     CATCH /aws1/cx_knskmsnotfoundex.
       MESSAGE 'KMS key used is not found.' TYPE 'E'.
     CATCH /aws1/cx_knskmsoptinrequired.
```

```
        MESSAGE 'KMS key option is required.' TYPE 'E'.
      CATCH /aws1/cx_knskmsthrottlingex.
        MESSAGE 'The rate of requests to AWS KMS is exceeding the request
 quotas.' TYPE 'E'.
      CATCH /aws1/cx_knsprovthruputexcdex.
        MESSAGE 'The request rate for the stream is too high, or the requested
 data is too large for the available throughput.' TYPE 'E'.
      CATCH /aws1/cx_knsresourcenotfoundex.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- For API details, see PutRecord in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutRecords` with an AWS SDK or CLI

The following code examples show how to use `PutRecords`.

CLI

AWS CLI

To write multiple records into a data stream

The following `put-records` example writes a data record using the specified partition key and another data record using a different partition key in a single call.

```
aws kinesis put-records \
    --stream-name samplestream \
    --
records Data=blob1,PartitionKey=partitionkey1 Data=blob2,PartitionKey=partitionkey2
```

Output:

```
{
    "FailedRecordCount": 0,
    "Records": [
        {
```

```
            "SequenceNumber":
 "49600883331171471519674795588238531498465399900093808706",
            "ShardId": "shardId-000000000004"
        },
        {
            "SequenceNumber":
 "49600902273357540915989931256902715169698037101720764562",
            "ShardId": "shardId-000000000009"
        }
    ],
    "EncryptionType": "KMS"
}
```

For more information, see Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see PutRecords in *AWS CLI Command 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 { PutRecordsCommand, KinesisClient } from "@aws-sdk/client-kinesis";

/**
 * Put multiple records into a Kinesis stream.
 * @param {{ streamArn: string }} config
 */
export const main = async ({ streamArn }) => {
  const client = new KinesisClient({});
  try {
    await client.send(
      new PutRecordsCommand({
        StreamARN: streamArn,
        Records: [
```

```
          {
            Data: new Uint8Array(),
            /**
             * Determines which shard in the stream the data record is assigned
 to.
             * Partition keys are Unicode strings with a maximum length limit of
 256
             * characters for each key. Amazon Kinesis Data Streams uses the
 partition
             * key as input to a hash function that maps the partition key and
             * associated data to a specific shard.
             */
            PartitionKey: "TEST_KEY",
          },
          {
            Data: new Uint8Array(),
            PartitionKey: "TEST_KEY",
          },
        ],
      }),
    );
  } catch (caught) {
    if (caught instanceof Error) {
      //
    } else {
      throw caught;
    }
  }
};

// Call function if run directly.
import { fileURLToPath } from "node:url";
import { parseArgs } from "node:util";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const options = {
    streamArn: {
      type: "string",
      description: "The ARN of the stream.",
    },
  };

  const { values } = parseArgs({ options });
  main(values);
```

```
}
```

- For API details, see PutRecords in *AWS SDK for JavaScript API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `RegisterStreamConsumer` with an AWS SDK or CLI

The following code examples show how to use `RegisterStreamConsumer`.

.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 System;
    using System.Threading.Tasks;
    using Amazon.Kinesis;
    using Amazon.Kinesis.Model;

    /// <summary>
    /// This example shows how to register a consumer to an Amazon Kinesis
    /// stream.
    /// </summary>
    public class RegisterConsumer
    {
        public static async Task Main()
        {
            IAmazonKinesis client = new AmazonKinesisClient();
            string consumerName = "NEW_CONSUMER_NAME";
            string streamARN = "arn:aws:kinesis:us-east-2:000000000000:stream/
AmazonKinesisStream";
```

```
            var consumer = await RegisterConsumerAsync(client, consumerName,
 streamARN);

            if (consumer is not null)
            {
                Console.WriteLine($"{consumer.ConsumerName}");
            }
        }

        /// <summary>
        /// Registers the consumer to a Kinesis stream.
        /// </summary>
        /// <param name="client">The initialized Kinesis client object.</param>
        /// <param name="consumerName">A string representing the consumer.</
param>
        /// <param name="streamARN">The ARN of the stream.</param>
        /// <returns>A Consumer object that contains information about the
 consumer.</returns>
        public static async Task<Consumer> RegisterConsumerAsync(IAmazonKinesis
 client, string consumerName, string streamARN)
        {
            var request = new RegisterStreamConsumerRequest
            {
                ConsumerName = consumerName,
                StreamARN = streamARN,
            };

            var response = await client.RegisterStreamConsumerAsync(request);
            return response.Consumer;
        }
    }
```

- For API details, see [RegisterStreamConsumer](#) in *AWS SDK for .NET API Reference*.

CLI

AWS CLI

To register a data stream consumer

The following `register-stream-consumer` example registers a consumer called `KinesisConsumerApplication` with the specified data stream.

```
aws kinesis register-stream-consumer \
    --stream-arn arn:aws:kinesis:us-west-2:012345678912:stream/samplestream \
    --consumer-name KinesisConsumerApplication
```

Output:

```
{
    "Consumer": {
        "ConsumerName": "KinesisConsumerApplication",
        "ConsumerARN": "arn:aws:kinesis:us-west-2: 123456789012:stream/
samplestream/consumer/KinesisConsumerApplication:1572383852",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1572383852.0
    }
}
```

For more information, see Developing Consumers with Enhanced Fan-Out Using the Kinesis Data Streams API in the *Amazon Kinesis Data Streams Developer Guide*.

- For API details, see RegisterStreamConsumer in *AWS CLI Command Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_kns->registerstreamconsumer(       " oo_result is returned
 for testing purposes. "
            iv_streamarn = iv_stream_arn
            iv_consumername = iv_consumer_name ).
        MESSAGE 'Stream consumer registered.' TYPE 'I'.
```

```
      CATCH /aws1/cx_knsinvalidargumentex.
        MESSAGE 'The specified argument was not valid.' TYPE 'E'.
      CATCH /aws1/cx_sgmresourcelimitexcd.
        MESSAGE 'You have reached the limit on the number of resources.' TYPE
'E'.
      CATCH /aws1/cx_sgmresourceinuse.
        MESSAGE 'Resource being accessed is in use.' TYPE 'E'.
      CATCH /aws1/cx_sgmresourcenotfound.
        MESSAGE 'Resource being accessed is not found.' TYPE 'E'.
    ENDTRY.
```

- For API details, see RegisterStreamConsumer in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Serverless examples for Kinesis

The following code examples show how to use Kinesis with AWS SDKs.

Examples

- Invoke a Lambda function from a Kinesis trigger
- Reporting batch item failures for Lambda functions with a Kinesis trigger

Invoke a Lambda function from a Kinesis trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a Kinesis stream. The function retrieves the Kinesis payload, decodes from Base64, and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

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 Serverless examples repository.

Consuming a Kinesis event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

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 Serverless examples repository.

Consuming a Kinesis event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Consuming a Kinesis event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Reporting batch item failures for Lambda functions with a Kinesis trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from a Kinesis stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
```

```
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

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 Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
```

```
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

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 Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
```

```
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
```

```
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();
```

```
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
```

```
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());
```

```
    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Document history

The following table describes the important changes to the Amazon Kinesis Data Streams documentation.

Change	Description	Date Changed
Added support for resilience testing with AWS Fault Injection Service.	Added Perform resilience testing with AWS Fault Injection Service.	October 15, 2025
Added support for large record sizes up to 10 MiB.	Added Handle large records.	October 27, 2025
End-of-support dates and version lifecycle policy for KPL and KCL.	Added information about end-of-support dates and version lifecycle policy for Amazon Kinesis Producer Library (KPL) and Amazon Kinesis Client Library (KCL). For more information, see KPL version lifecycle policy and KCL version lifecycle policy.	March 13, 2025
Added support for sharing data streams across acounts.	Added Share your data stream with another account.	November 22, 2023
Added support for the on-demand and provision ed data stream capacity modes.	Added Choose the right mode to stream in.	November 29, 2021
New content for server-side encryption.	Added Data protection in Amazon Kinesis Data Streams.	July 7, 2017

Change	Description	Date Changed
New content for enhanced CloudWatch metrics.	Updated Monitor Kinesis Data Streams.	April 19, 2016
New content for enhanced Kinesis agent.	Updated Write to Amazon Kinesis Data Streams using Kinesis Agent.	April 11, 2016
New content for using Kinesis agents.	Added Write to Amazon Kinesis Data Streams using Kinesis Agent.	October 2, 2015
Update KPL content for release 0.10.0.	Added Develop producers using the Amazon Kinesis Producer Library (KPL).	July 15, 2015
Update KCL metrics topic for configurable metrics.	Added Monitor the Kinesis Client Library with Amazon CloudWatch.	July 9, 2015
Re-organized content.	Significantly re-organized content topics for more concise tree view and more logical grouping.	July 01, 2015
New KPL developer's guide topic.	Added Develop producers using the Amazon Kinesis Producer Library (KPL).	June 02, 2015
New KCL metrics topic.	Added Monitor the Kinesis Client Library with Amazon CloudWatch.	May 19, 2015
Support for KCL .NET	Added Develop a Kinesis Client Library consumer in .NET.	May 1, 2015
Support for KCL Node.js	Added Develop a Kinesis Client Library consumer in Node.js.	March 26, 2015

Change	Description	Date Changed
Support for KCL Ruby	Added links to KCL Ruby library.	January 12, 2015
New API PutRecords	Added information about new PutRecords API to the section called "Add multiple records with PutRecords".	December 15, 2014
Support for tagging	Added Tag your Amazon Kinesis Data Streams resources.	September 11, 2014
New CloudWatch metric	Added the metric `GetRecords.IteratorAgeMilliseconds` to Amazon Kinesis Data Streams dimensions and metrics.	September 3, 2014
New monitoring chapter	Added Monitor Kinesis Data Streams and Monitor the Amazon Kinesis Data Streams service with Amazon CloudWatch.	July 30, 2014
Default shard limit	Updated the Quotas and limits: the default shard limit has been raised from 5 to 10.	February 25, 2014
Default shard limit	Updated the Quotas and limits: the default shard limit has been raised from 2 to 5.	January 28, 2014
API version updates	Updates for version 2013-12-02 of the Kinesis Data Streams API.	December 12, 2013
Initial release	Initial release of the Amazon Kinesis Developer Guide.	November 14, 2013