

# Kafka 3.4 Documentation

---

Prior releases: [0.7.x](#), [0.8.0](#), [0.8.1.X](#), [0.8.2.X](#), [0.9.0.X](#), [0.10.0.X](#), [0.10.1.X](#), [0.10.2.X](#), [0.11.0.X](#), [1.0.X](#), [1.1.X](#), [2.0.X](#), [2.1.X](#), [2.2.X](#), [2.3.X](#), [2.4.X](#), [2.5.X](#), [2.6.X](#), [2.7.X](#), [2.8.X](#), [3.0.X](#), [3.1.X](#), [3.2.X](#), [3.3.X](#).

## 1. GETTING STARTED

---

### 1.1 Introduction

#### What is event streaming?

Event streaming is the digital equivalent of the human body's central nervous system. It is the technological foundation for the 'always-on' world where businesses are increasingly software-defined and automated, and where the user of software is more software.

Technically speaking, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

#### What can I use event streaming for?

Event streaming is applied to a [wide variety of use cases](#) across a plethora of industries and organizations. Its many examples include:

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.
- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.
- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.

#### Apache Kafka® is an event streaming platform. What does that mean?

Kafka combines three key capabilities so you can implement [your use cases](#) for event streaming end-to-end with a single battle-tested solution:

1. To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems.
2. To **store** streams of events durably and reliably for as long as you want.
3. To **process** streams of events as they occur or retrospectively.

And all this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud. You can choose between self-managing your Kafka environments and using fully managed services offered by a variety of vendors.

## How does Kafka work in a nutshell?

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance [TCP network protocol](#). It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

**Servers:** Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run [Kafka Connect](#) to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

**Clients:** They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by [dozens of clients](#) provided by the Kafka community: clients are available for Java and Scala including the higher-level [Kafka Streams](#) library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

## Main Concepts and Terminology

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of \$200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

**Producers** are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various [guarantees](#) such as the ability to process events exactly-once.

Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments". Topics in Kafka are always multi-producer and multi-consumer: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka [guarantees](#) that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

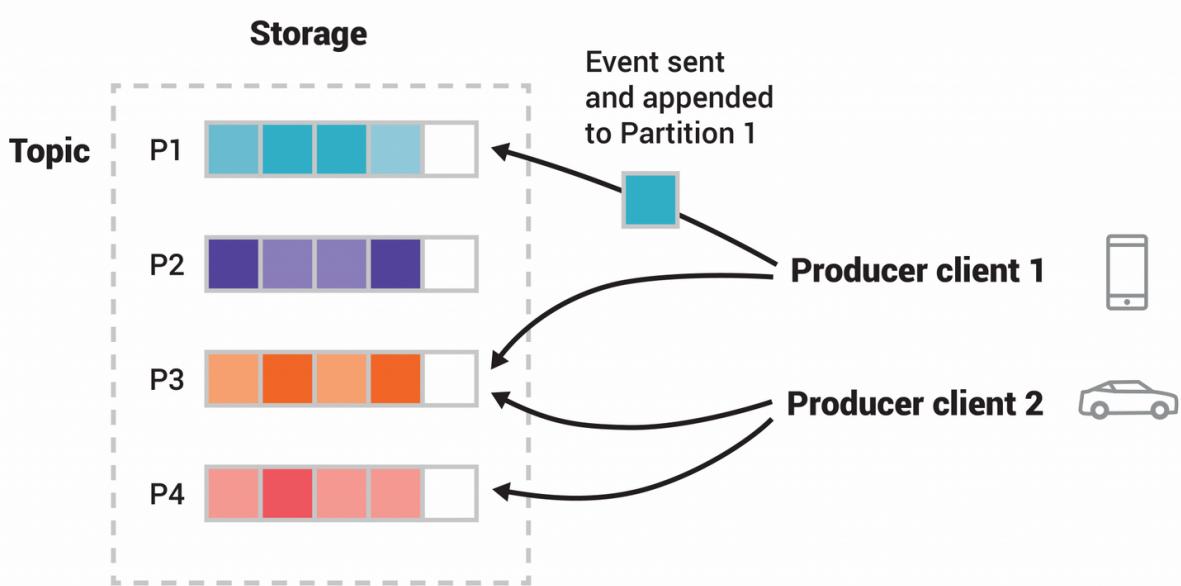


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

This primer should be sufficient for an introduction. The [Design](#) section of the documentation explains Kafka's various concepts in full detail, if you are interested.

## [Kafka APIs](#)

In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The [Admin API](#) to manage and inspect topics, brokers, and other Kafka objects.
- The [Producer API](#) to publish (write) a stream of events to one or more Kafka topics.
- The [Consumer API](#) to subscribe to (read) one or more topics and to process the stream of events produced to them.
- The [Kafka Streams API](#) to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.
- The [Kafka Connect API](#) to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. For example, a connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don't need to implement your own connectors because the Kafka community already provides hundreds of ready-to-use connectors.

## [Where to go from here](#)

- To get hands-on experience with Kafka, follow the [Quickstart](#).
- To understand Kafka in more detail, read the [Documentation](#). You also have your choice of [Kafka books and academic papers](#).

- Browse through the [Use Cases](#) to learn how other users in our world-wide community are getting value out of Kafka.
- Join a [local Kafka meetup group](#) and [watch talks from Kafka Summit](#), the main conference of the Kafka community.

## **1.2 Use Cases**

Here is a description of a few of the popular use cases for Apache Kafka®. For an overview of a number of these areas in action, see [this blog post](#).

### **Messaging**

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.

In this domain Kafka is comparable to traditional messaging systems such as [ActiveMQ](#) or [RabbitMQ](#).

### **Website Activity Tracking**

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view.

### **Metrics**

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

### **Log Aggregation**

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

### **Stream Processing**

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called [Kafka Streams](#) is available in Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include [Apache Storm](#) and [Apache Samza](#).

## Event Sourcing

[Event sourcing](#) is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

## Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The [log compaction](#) feature in Kafka helps support this usage. In this usage Kafka is similar to [Apache BookKeeper](#) project.

## 1.3 Quick Start

### STEP 1: GET KAFKA

[Download](#) the latest Kafka release and extract it:

```
$ tar -xzf kafka_2.13-3.4.0.tgz  
$ cd kafka_2.13-3.4.0
```

### STEP 2: START THE KAFKA ENVIRONMENT

NOTE: Your local environment must have Java 8+ installed.

Apache Kafka can be started using ZooKeeper or KRaft. To get started with either configuration follow one the sections below but not both.

#### **Kafka with ZooKeeper**

Run the following commands in order to start all services in the correct order:

```
# Start the Zookeeper service  
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

Open another terminal session and run:

```
# Start the Kafka broker service  
$ bin/kafka-server-start.sh config/server.properties
```

Once all services have successfully launched, you will have a basic Kafka environment running and ready to use.

#### **Kafka with KRaft**

Generate a Cluster UUID

```
$ KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

Format Log Directories

```
$ bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
```

Start the Kafka Server

```
$ bin/kafka-server-start.sh config/kraft/server.properties
```

Once the Kafka server has successfully launched, you will have a basic Kafka environment running and ready to use.

## STEP 3: CREATE A TOPIC TO STORE YOUR EVENTS

Kafka is a distributed *event streaming platform* that lets you read, write, store, and process [events](#) (also called *records* or *messages* in the documentation) across many machines.

Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in [topics](#). Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder.

So before you can write your first events, you must create a topic. Open another terminal session and run:

```
$ bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server  
localhost:9092
```

All of Kafka's command line tools have additional options: run the `kafka-topics.sh` command without any arguments to display usage information. For example, it can also show you [details such as the partition count](#) of the new topic:

```
$ bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server  
localhost:9092  
Topic: quickstart-events          TopicId: NPmZHyhbR9y00wMg1MH2sg PartitionCount: 1  
ReplicationFactor: 1    Configs:  
    Topic: quickstart-events Partition: 0    Leader: 0    Replicas: 0 Isr: 0
```

## STEP 4: WRITE SOME EVENTS INTO THE TOPIC

A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever.

Run the console producer client to write a few events into your topic. By default, each line you enter will result in a separate event being written to the topic.

```
$ bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server  
localhost:9092  
This is my first event  
This is my second event
```

You can stop the producer client with `Ctrl-C` at any time.

## STEP 5: READ THE EVENTS

Open another terminal session and run the console consumer client to read the events you just created:

```
$ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --  
bootstrap-server localhost:9092  
This is my first event  
This is my second event
```

You can stop the consumer client with `Ctrl-C` at any time.

Feel free to experiment: for example, switch back to your producer terminal (previous step) to write additional events, and see how the events immediately show up in your consumer terminal.

Because events are durably stored in Kafka, they can be read as many times and by as many consumers as you want. You can easily verify this by opening yet another terminal session and re-running the previous command again.

## **STEP 6: IMPORT/EXPORT YOUR DATA AS STREAMS OF EVENTS WITH KAFKA CONNECT**

You probably have lots of data in existing systems like relational databases or traditional messaging systems, along with many applications that already use these systems. [Kafka Connect](#) allows you to continuously ingest data from external systems into Kafka, and vice versa. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. It is thus very easy to integrate existing systems with Kafka. To make this process even easier, there are hundreds of such connectors readily available.

In this quickstart we'll see how to run Kafka Connect with simple connectors that import data from a file to a Kafka topic and export data from a Kafka topic to a file.

First, make sure to add `connect-file-3.4.0.jar` to the `plugin.path` property in the Connect worker's configuration. For the purpose of this quickstart we'll use a relative path and consider the connectors' package as an uber jar, which works when the quickstart commands are run from the installation directory. However, it's worth noting that for production deployments using absolute paths is always preferable. See [plugin.path](#) for a detailed description of how to set this config.

Edit the `config/connect-standalone.properties` file, add or change the `plugin.path` configuration property match the following, and save the file:

```
> echo "plugin.path=libs/connect-file-3.4.0.jar"
```

Then, start by creating some seed data to test with:

```
> echo -e "foo\nbar" > test.txt
```

Or on Windows:

```
> echo foo> test.txt  
> echo bar>> test.txt
```

Next, we'll start two connectors running in *standalone* mode, which means they run in a single, local, dedicated process. We provide three configuration files as parameters. The first is always the configuration for the Kafka Connect process, containing common configuration such as the Kafka brokers to connect to and the serialization format for data. The remaining configuration files each specify a connector to create. These files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

```
> bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties
```

These sample configuration files, included with Kafka, use the default local cluster configuration you started earlier and create two connectors: the first is a source connector that reads lines from an input file and produces each to a Kafka topic and the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file.

During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated. Once the Kafka Connect process has started, the source connector should start reading lines from `test.txt` and producing them to the topic `connect-test`, and the sink connector should start reading messages from the topic `connect-test` and write them to the file `test.sink.txt`.

We can verify the data has been delivered through the entire pipeline by examining the contents of the output file:

```
> more test.sink.txt
foo
bar
```

Note that the data is being stored in the Kafka topic `connect-test`, so we can also run a console consumer to see the data in the topic (or use custom consumer code to process it):

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test
--from-beginning
>{"schema": {"type": "string", "optional": false}, "payload": "foo"}
>{"schema": {"type": "string", "optional": false}, "payload": "bar"}
...
```

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

```
> echo Another line>> test.txt
```

You should see the line appear in the console consumer output and in the sink file.

## **STEP 7: PROCESS YOUR EVENTS WITH KAFKA STREAMS**

Once your data is stored in Kafka as events, you can process the data with the [Kafka Streams](#) client library for Java/Scala. It allows you to implement mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka topics. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, and distributed. The library supports exactly-once processing, stateful operations and aggregations, windowing, joins, processing based on event-time, and much more.

To give you a first taste, here's how one would implement the popular `wordCount` algorithm:

```
KStream<String, String> textLines = builder.stream("quickstart-events");

KTable<String, Long> wordCounts = textLines
    .flatMapValues(line -> Arrays.asList(line.toLowerCase().split(" ")))
    .groupBy((keyIgnored, word) -> word)
    .count();

wordCounts.toStream().to("output-topic", Produced.with(Serdes.String(),
Serdes.Long()));
```

The [Kafka Streams demo](#) and the [app development tutorial](#) demonstrate how to code and run such a streaming application from start to finish.

## **STEP 8: TERMINATE THE KAFKA ENVIRONMENT**

Now that you reached the end of the quickstart, feel free to tear down the Kafka environment—or continue playing around.

1. Stop the producer and consumer clients with `ctrl-c`, if you haven't done so already.
2. Stop the Kafka broker with `ctrl-c`.
3. Lastly, if the Kafka with ZooKeeper section was followed, stop the ZooKeeper server with `ctrl-c`.

If you also want to delete any data of your local Kafka environment including any events you have created along the way, run the command:

```
$ rm -rf /tmp/kafka-logs /tmp/zookeeper /tmp/kraft-combined-logs
```

## [CONGRATULATIONS!](#)

You have successfully finished the Apache Kafka quickstart.

To learn more, we suggest the following next steps:

- Read through the brief [Introduction](#) to learn how Kafka works at a high level, its main concepts, and how it compares to other technologies. To understand Kafka in more detail, head over to the [Documentation](#).
- Browse through the [Use Cases](#) to learn how other users in our world-wide community are getting value out of Kafka.
- Join a [local Kafka meetup group](#) and [watch talks from Kafka Summit](#), the main conference of the Kafka community.

## [1.4 Ecosystem](#)

There are a plethora of tools that integrate with Kafka outside the main distribution. The [ecosystem page](#) lists many of these, including stream processing systems, Hadoop integration, monitoring, and deployment tools.

## [1.5 Upgrading From Previous Versions](#)

### [Upgrading to 3.4.0 from any version 0.8.x through 3.3.x](#)

**If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the inter.broker.protocol.version to the latest version, it will not be possible to downgrade to a version prior to 2.1.**

**For a rolling upgrade:**

1. Update server.properties on all brokers and add the following properties.  
CURRENT\_KAFKA\_VERSION refers to the version you are upgrading from.  
CURRENT\_MESSAGE\_FORMAT\_VERSION refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then CURRENT\_MESSAGE\_FORMAT\_VERSION should be set to match CURRENT\_KAFKA\_VERSION.
  - inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.3, 3.2, etc.)
  - log.message.format.version=CURRENT\_MESSAGE\_FORMAT\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.3, 3.2, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 3.4.

4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change log.message.format.version to 3.4 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## [Upgrading a KRaft-based cluster to 3.4.0 from any version 3.0.x through 3.3.x](#)

**If you are upgrading from a version prior to 3.3.0, please see the note below. Once you have changed the metadata.version to the latest version, it will not be possible to downgrade to a version prior to 3.3-IV0.**

**For a rolling upgrade:**

1. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations.
2. Once the cluster's behavior and performance has been verified, bump the metadata.version by running `./bin/kafka-features.sh upgrade --metadata 3.4`
3. Note that the cluster metadata version cannot be downgraded to a pre-production 3.0.x, 3.1.x, or 3.2.x version once it has been upgraded. However, it is possible to downgrade to production versions such as 3.3-IV0, 3.3-IV1, etc.

### [Notable changes in 3.4.0](#)

- Since Apache Kafka 3.4.0, we have added a system property ("org.apache.kafka.disallowed.login.modules") to disable the problematic login modules usage in SASL JAAS configuration. Also by default "com.sun.security.auth.module.JndiLoginModule" is disabled from Apache Kafka 3.4.0.

## [Upgrading to 3.3.1 from any version 0.8.x through 3.2.x](#)

**If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the inter.broker.protocol.version to the latest version, it will not be possible to downgrade to a version prior to 2.1.**

**For a rolling upgrade:**

1. Update server.properties on all brokers and add the following properties.  
CURRENT\_KAFKA\_VERSION refers to the version you are upgrading from.  
CURRENT\_MESSAGE\_FORMAT\_VERSION refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then CURRENT\_MESSAGE\_FORMAT\_VERSION should be set to match CURRENT\_KAFKA\_VERSION.
  - inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.2, 3.1, etc.)
  - log.message.format.version=CURRENT\_MESSAGE\_FORMAT\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.2, 3.1, etc.)

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to `3.3`.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 3.3 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## [Upgrading a KRaft-based cluster to 3.3.1 from any version 3.0.x through 3.2.x](#)

If you are upgrading from a version prior to 3.3.1, please see the note below. Once you have changed the `metadata.version` to the latest version, it will not be possible to downgrade to a version prior to 3.3-IV0.

For a rolling upgrade:

1. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations.
2. Once the cluster's behavior and performance has been verified, bump the `metadata.version` by running `./bin/kafka-features.sh upgrade -metadata 3.3`
3. Note that the cluster metadata version cannot be downgraded to a pre-production 3.0.x, 3.1.x, or 3.2.x version once it has been upgraded. However, it is possible to downgrade to production versions such as 3.3-IV0, 3.3-IV1, etc.

## [Notable changes in 3.3.1](#)

- KRaft mode is production ready for new clusters. See [KIP-833](#) for more details (including limitations).
- The partitioner used by default for records with no keys has been improved to avoid pathological behavior when one or more brokers are slow. The new logic may affect the batching behavior, which can be tuned using the `batch.size` and/or `linger.ms` configuration settings. The previous behavior can be restored by setting `partitioner.class=org.apache.kafka.clients.producer.internals.DefaultPartitioner`. See [KIP-794](#) for more details.
- There is now a slightly different upgrade process for KRaft clusters than for ZK-based clusters, as described above.
- Introduced a new API `addMetricIfAbsent` to `Metrics` which would create a new Metric if not existing or return the same metric if already registered. Note that this behaviour is different from `addMetric` API which throws an `IllegalArgumentException` when trying to create an already existing metric. (See [KIP-843](#) for more details).

## [Upgrading to 3.2.0 from any version 0.8.x through 3.1.x](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

## For a rolling upgrade:

1. Update server.properties on all brokers and add the following properties.  
CURRENT\_KAFKA\_VERSION refers to the version you are upgrading from.  
CURRENT\_MESSAGE\_FORMAT\_VERSION refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then CURRENT\_MESSAGE\_FORMAT\_VERSION should be set to match CURRENT\_KAFKA\_VERSION.
  - o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.1, 3.0, etc.)
  - o log.message.format.version=CURRENT\_MESSAGE\_FORMAT\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 3.1, 3.0, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 3.2.
  4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
  5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 3.2 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## [Notable changes in 3.2.0](#)

- Idempotence for the producer is enabled by default if no conflicting configurations are set. When producing to brokers older than 2.8.0, the `IDEMPOTENT_WRITE` permission is required. Check the compatibility section of [KIP-679 for details. In 3.0.0 and 3.1.0, a bug prevented this default from being applied, which meant that idempotence remained disabled unless the user had explicitly set enable.idempotence to true \(See KAFKA-13598 for more details\)](#). This issue was fixed and the default is properly applied in 3.0.1, 3.1.1, and 3.2.0.
- A notable exception is Connect that by default disables idempotent behavior for all of its producers in order to uniformly support using a wide range of Kafka broker versions. Users can change this behavior to enable idempotence for some or all producers via Connect worker and/or connector configuration. Connect may enable idempotent producers by default in a future major release.
- Kafka has replaced log4j with reload4j due to security concerns. This only affects modules that specify a logging backend (`connect-runtime` and `kafka-tools` are two such examples). A number of modules, including `kafka-clients`, leave it to the application to specify the logging backend. More information can be found at [reload4j](#). Projects that depend on the affected modules from the Kafka project should use [slf4j-log4j12 version 1.7.35 or above](#) or slf4j-reload4j to avoid [possible compatibility issues originating from the logging framework](#).
- The example connectors, `FilestreamSourceConnector` and `FilestreamsinkConnector`, have been removed from the default classpath. To use them in Kafka Connect standalone or distributed mode they need to be explicitly added, for example `CLASSPATH=../lib/connect-file-3.2.0.jar ./bin/connect-distributed.sh`.

## [Upgrading to 3.1.0 from any version 0.8.x through 3.0.x](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

### For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 3.0, 2.8, etc.)
  - o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 3.0, 2.8, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 3.1.
  4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
  5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 3.1 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

### [Notable changes in 3.1.1](#)

- Idempotence for the producer is enabled by default if no conflicting configurations are set. When producing to brokers older than 2.8.0, the `IDEMPOTENT_WRITE` permission is required. Check the compatibility section of [KIP-679 for details. A bug prevented the producer idempotence default from being applied which meant that it remained disabled unless the user had explicitly set enable.idempotence to true. See KAFKA-13598](#) for more details. This issue was fixed and the default is properly applied.
- A notable exception is Connect that by default disables idempotent behavior for all of its producers in order to uniformly support using a wide range of Kafka broker versions. Users can change this behavior to enable idempotence for some or all producers via Connect worker and/or connector configuration. Connect may enable idempotent producers by default in a future major release.
- Kafka has replaced log4j with reload4j due to security concerns. This only affects modules that specify a logging backend (`connect-runtime` and `kafka-tools` are two such examples). A number of modules, including `kafka-clients`, leave it to the application to specify the logging backend. More information can be found at [reload4j](#). Projects that depend on the affected modules from the Kafka project should use [slf4j-log4j12 version 1.7.35 or above](#) or slf4j-reload4j to avoid [possible compatibility issues originating from the logging framework](#).

## Notable changes in 3.1.0

- Apache Kafka supports Java 17.
- The following metrics have been deprecated: `bufferpool-wait-time-total`, `io-waittime-total`, and `iotime-total`. Please use `bufferpool-wait-time-ns-total`, `io-wait-time-ns-total`, and `io-time-ns-total` instead. See [KIP-773](#) for more details.
- IBP 3.1 introduces topic IDs to FetchRequest as a part of [KIP-516](#).

## Upgrading to 3.0.1 from any version 0.8.x through 2.8.x

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.8, 2.7, etc.)
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.8, 2.7, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 3.0.
  4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
  5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 3.0 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## Notable changes in 3.0.1

- Idempotence for the producer is enabled by default if no conflicting configurations are set. When producing to brokers older than 2.8.0, the `IDEMPOTENT_WRITE` permission is required. Check the compatibility section of [KIP-679](#) for details. A bug prevented the producer idempotence default from being applied which meant that it remained disabled unless the user had explicitly set `enable.idempotence` to true. See [KAFKA-13598](#) for more details. This issue was fixed and the default is properly applied.

## [Notable changes in 3.0.0](#)

- The producer has stronger delivery guarantees by default: `idempotence` is enabled and `acks` is set to `a11` instead of `1`. See [KIP-679](#) for details. In 3.0.0 and 3.1.0, a bug prevented the idempotence default from being applied which meant that it remained disabled unless the user had explicitly set `enable.idempotence` to true. Note that the bug did not affect the `acks=a11` change. See [KAFKA-13598](#) for more details. This issue was fixed and the default is properly applied in 3.0.1, 3.1.1, and 3.2.0.
- Java 8 and Scala 2.12 support have been deprecated since Apache Kafka 3.0 and will be removed in Apache Kafka 4.0. See [KIP-750](#) and [KIP-751](#) for more details.
- ZooKeeper has been upgraded to version 3.6.3.
- A preview of KRaft mode is available, though upgrading to it from the 2.8 Early Access release is not possible. See the `config/kraft/README.md` file for details.
- The release tarball no longer includes test, sources, javadoc and test sources jars. These are still published to the Maven Central repository.
- A number of implementation dependency jars are [now available in the runtime classpath instead of compile and runtime classpaths](#). Compilation errors after the upgrade can be fixed by adding the missing dependency jar(s) explicitly or updating the application not to use internal classes.
- The default value for the consumer configuration `session.timeout.ms` was increased from 10s to 45s. See [KIP-735](#) for more details.
- The broker configuration `log.message.format.version` and topic configuration `message.format.version` have been deprecated. The value of both configurations is always assumed to be `3.0` if `inter.broker.protocol.version` is `3.0` or higher. If `log.message.format.version` or `message.format.version` are set, we recommend clearing them at the same time as the `inter.broker.protocol.version` upgrade to 3.0. This will avoid potential compatibility issues if the `inter.broker.protocol.version` is downgraded. See [KIP-724](#) for more details.
- The Streams API removed all deprecated APIs that were deprecated in version 2.5.0 or earlier. For a complete list of removed APIs compare the detailed Kafka Streams upgrade notes.
- Kafka Streams no longer has a compile time dependency on "connect:json" module ([KAFKA-5146](#)). Projects that were relying on this transitive dependency will have to explicitly declare it.
- Custom principal builder implementations specified through `principal.builder.class` must now implement the `KafkaPrincipalSerde` interface to allow for forwarding between brokers. See [KIP-590](#) for more details about the usage of KafkaPrincipalSerde.
- A number of deprecated classes, methods and tools have been removed from the `clients`, `connect`, `core` and `tools` modules:
  - The Scala `Authorizer`, `SimpleAclAuthorizer` and related classes have been removed. Please use the Java `Authorizer` and `AclAuthorizer` instead.
  - The `Metric#value()` method was removed ([KAFKA-12573](#)).
  - The `Sum` and `Total` classes were removed ([KAFKA-12584](#)). Please use `windowedsum` and `cumulativeSum` instead.
  - The `Count` and `SampledTotal` classes were removed. Please use `windowedCount` and `windowedsum` respectively instead.
  - The `PrincipalBuilder`, `DefaultPrincipalBuilder` and `ResourceFilter` classes were removed.
  - Various constants and constructors were removed from `sslConfigs`, `saslConfigs`, `AclBinding` and `AclBindingFilter`.
  - The `Admin.electedPreferredLeaders()` methods were removed. Please use `Admin.electLeaders` instead.

- The `kafka-preferred-replica-election` command line tool was removed. Please use `kafka-leader-election` instead.
  - The `--zookeeper` option was removed from the `kafka-topics` and `kafka-reassign-partitions` command line tools. Please use `--bootstrap-server` instead.
  - In the `kafka-configs` command line tool, the `--zookeeper` option is only supported for updating [SCRAM Credentials configuration](#) and [describing/updating dynamic broker configs when brokers are not running](#). Please use `--bootstrap-server` for other configuration operations.
  - The `ConfigEntry` constructor was removed ([KAFKA-12577](#)). Please use the remaining public constructor instead.
  - The config value `default` for the client config `client.dns.lookup` has been removed. In the unlikely event that you set this config explicitly, we recommend leaving the config unset (`use_all_dns_ips` is used by default).
  - The `ExtendedDeserializer` and `ExtendedSerializer` classes have been removed. Please use `Deserializer` and `Serializer` instead.
  - The `close(long, TimeUnit)` method was removed from the producer, consumer and admin client. Please use `close(Duration)`.
  - The `ConsumerConfig.addDeserializerToConfig` and `ProducerConfig.addSerializerToConfig` methods were removed. These methods were not intended to be public API and there is no replacement.
  - The `NoOffsetForPartitionException.partition()` method was removed. Please use `partitions()` instead.
  - The default `partition.assignment.strategy` is changed to "[RangeAssignor, CooperativeStickyAssignor]", which will use the RangeAssignor by default, but allows upgrading to the CooperativeStickyAssignor with just a single rolling bounce that removes the RangeAssignor from the list. Please check the client upgrade path guide [here](#) for more detail.
  - The Scala `kafka.common.MessageFormatter` was removed. Please use the Java `org.apache.kafka.common.MessageFormatter`.
  - The `MessageFormatter.init(Properties)` method was removed. Please use `configure(Map)` instead.
  - The `checksum()` method has been removed from `ConsumerRecord` and `RecordMetadata`. The message format v2, which has been the default since 0.11, moved the checksum from the record to the record batch. As such, these methods don't make sense and no replacements exist.
  - The `ChecksumMessageFormatter` class was removed. It is not part of the public API, but it may have been used with `kafka-console-consumer.sh`. It reported the checksum of each record, which has not been supported since message format v2.
  - The `org.apache.kafka.clients.consumer.internals.PartitionAssignor` class has been removed. Please use `org.apache.kafka.clients.consumer.ConsumerPartitionAssignor` instead.
  - The `quota.producer.default` and `quota.consumer.default` configurations were removed ([KAFKA-12591](#)). Dynamic quota defaults must be used instead.
  - The `port` and `host.name` configurations were removed. Please use `listeners` instead.
  - The `advertised.port` and `advertised.host.name` configurations were removed. Please use `advertised.listeners` instead.
  - The deprecated worker configurations `rest.host.name` and `rest.port` were removed ([KAFKA-12482](#)) from the Kafka Connect worker configuration. Please use `listeners` instead.
- The `Producer#sendOffsetsToTransaction(Map offsets, String consumerGroupId)` method has been deprecated. Please use `Producer#sendOffsetsToTransaction(Map offsets, ConsumerGroupMetadata metadata)` instead, where the `ConsumerGroupMetadata` can be retrieved via `KafkaConsumer#groupMetadata()` for stronger semantics. Note that the full set of consumer group metadata is only understood by brokers or version 2.5 or higher, so you must upgrade your

kafka cluster to get the stronger semantics. Otherwise, you can just pass in new `ConsumerGroupMetadata(consumerGroupId)` to work with older brokers. See [KIP-732](#) for more details.

- The Connect `internal.key.converter` and `internal.value.converter` properties have been completely [removed](#). The use of these Connect worker properties has been deprecated since version 2.0.0. Workers are now hardcoded to use the JSON converter with `schemas.enable` set to `false`. If your cluster has been using a different internal key or value converter, you can follow the migration steps outlined in [KIP-738](#) to safely upgrade your Connect cluster to 3.0.
- The Connect-based MirrorMaker (MM2) includes changes to support `IdentityReplicationPolicy`, enabling replication without renaming topics. The existing `DefaultReplicationPolicy` is still used by default, but identity replication can be enabled via the `replication.policy` configuration property. This is especially useful for users migrating from the older MirrorMaker (MM1), or for use-cases with simple one-way replication topologies where topic renaming is undesirable. Note that `IdentityReplicationPolicy`, unlike `DefaultReplicationPolicy`, cannot prevent replication cycles based on topic names, so take care to avoid cycles when constructing your replication topology.
- The original MirrorMaker (MM1) and related classes have been deprecated. Please use the Connect-based MirrorMaker (MM2), as described in the [Geo-Replication section](#).

## [Upgrading to 2.8.1 from any version 0.8.x through 2.7.x](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.7, 2.6, etc.)
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

1. `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.7, 2.6, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.8.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.8 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support

the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## [Notable changes in 2.8.0](#)

- The 2.8.0 release added a new method to the Authorizer Interface introduced in [KIP-679](#). The motivation is to unblock our future plan to enable the strongest message delivery guarantee by default. Custom authorizer should consider providing a more efficient implementation that supports audit logging and any custom configs or access rules.
- IBP 2.8 introduces topic IDs to topics as a part of [KIP-516](#). When using ZooKeeper, this information is stored in the TopicZNode. If the cluster is downgraded to a previous IBP or version, future topics will not get topic IDs and it is not guaranteed that topics will retain their topic IDs in ZooKeeper. This means that upon upgrading again, some topics or all topics will be assigned new IDs.
- Kafka Streams introduce a type-safe `split()` operator as a substitution for deprecated `KStream#branch()` method (cf. [KIP-418](#)).

## [Upgrading to 2.7.0 from any version 0.8.x through 2.6.x](#)

**If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.**

**For a rolling upgrade:**

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.6, 2.5, etc.)
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.6, 2.5, etc.)

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.7.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.7 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## Notable changes in 2.7.0

- The 2.7.0 release includes the core Raft implementation specified in [KIP-595](#). There is a separate "raft" module containing most of the logic. Until integration with the controller is complete, there is a standalone server that users can use for testing the performance of the Raft implementation. See the README.md in the raft module for details
- KIP-651 [adds support](#) for using PEM files for key and trust stores.
- KIP-612 [adds support](#) for enforcing broker-wide and per-listener connection create rates. The 2.7.0 release contains the first part of KIP-612 with dynamic configuration coming in the 2.8.0 release.
- The ability to throttle topic and partition creations or topics deletions to prevent a cluster from being harmed via [KIP-599](#)
- When new features become available in Kafka there are two main issues:
  1. How do Kafka clients become aware of broker capabilities?
  2. How does the broker decide which features to enable?

KIP-584

provides a flexible and operationally easy solution for client discovery, feature gating and rolling upgrades using a single restart.

- The ability to print record offsets and headers with the `ConsoleConsumer` is now possible via [KIP-431](#)
- The addition of [KIP-554](#) continues progress towards the goal of Zookeeper removal from Kafka. The addition of KIP-554 means you don't have to connect directly to ZooKeeper anymore for managing SCRAM credentials.
- Altering non-reconfigurable configs of existent listeners causes `InvalidRequestException`. By contrast, the previous (unintended) behavior would have caused the updated configuration to be persisted, but it wouldn't take effect until the broker was restarted. See [KAFKA-10479](#) for more discussion. See `DynamicBrokerConfig.DynamicSecurityConfigs` and `SocketServer.ListenerReconfigurableConfigs` for the supported reconfigurable configs of existent listeners.
- Kafka Streams adds support for [Sliding Windows Aggregations](#) in the KStreams DSL.
- Reverse iteration over state stores enabling more efficient most recent update searches with [KIP-617](#)
- End-to-End latency metrics in Kafka Streams see [KIP-613](#) for more details
- Kafka Streams added metrics reporting default RocksDB properties with [KIP-607](#)
- Better Scala implicit Serdes support from [KIP-616](#)

## Upgrading to 2.6.0 from any version 0.8.x through 2.5.x

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.5, 2.4, etc.)
- o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.5, 2.4, etc.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.6.
  4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
  5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.6 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

## [Notable changes in 2.6.0](#)

- Kafka Streams adds a new processing mode (requires broker 2.5 or newer) that improves application scalability using exactly-once guarantees (cf. [KIP-447](#))
- TLSv1.3 has been enabled by default for Java 11 or newer. The client and server will negotiate TLSv1.3 if both support it and fallback to TLSv1.2 otherwise. See [KIP-573](#) for more details.
- The default value for the `client.dns.lookup` configuration has been changed from `default` to `use_all_dns_ips`. If a hostname resolves to multiple IP addresses, clients and brokers will now attempt to connect to each IP in sequence until the connection is successfully established. See [KIP-602](#) for more details.
- `NotLeaderForPartitionException` has been deprecated and replaced with `NotLeaderOrFollowerException`. Fetch requests and other requests intended only for the leader or follower return NOT\_LEADER\_OR\_FOLLOWER(6) instead of REPLICA\_NOT\_AVAILABLE(9) if the broker is not a replica, ensuring that this transient error during reassignments is handled by all clients as a retriable exception.

## [Upgrading to 2.5.0 from any version 0.8.x through 2.4.x](#)

**If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.**

**For a rolling upgrade:**

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
 Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.4, 2.3, etc.)

- o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 2.4, 2.3, etc.)

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.5.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.5 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.
6. There are several notable changes to the reassignment tool `kafka-reassign-partitions.sh` following the completion of [KIP-455](#). This tool now requires the `-additional` flag to be provided when changing the throttle of an active reassignment. Reassignment cancellation is now possible using the `--cancel` command. Finally, reassignment with `--zookeeper` has been deprecated in favor of `--bootstrap-server`. See the KIP for more detail.

## [Notable changes in 2.5.0](#)

- When `RebalanceProtocol#COOPERATIVE` is used, `Consumer#poll` can still return data while it is in the middle of a rebalance for those partitions still owned by the consumer; in addition `Consumer#commitsync` now may throw a non-fatal `RebalanceInProgressException` to notify users of such an event, in order to distinguish from the fatal `CommitFailedException` and allow users to complete the ongoing rebalance and then reattempt committing offsets for those still-owned partitions.
- For improved resiliency in typical network environments, the default value of `zookeeper.session.timeout.ms` has been increased from 6s to 18s and `replica.lag.time.max.ms` from 10s to 30s.
- New DSL operator `cogroup()` has been added for aggregating multiple streams together at once.
- Added a new `kstream.toTable()` API to translate an input event stream into a KTable.
- Added a new Serde type `void` to represent null keys or null values from input topic.
- Deprecated `usePreviousTimeOnInvalidTimestamp` and replaced it with `usePartitionTimeOnInvalidTimestamp`.
- Improved exactly-once semantics by adding a pending offset fencing mechanism and stronger transactional commit consistency check, which greatly simplifies the implementation of a scalable exactly-once application. We also added a new exactly-once semantics code example under [examples](#) folder. Check out [KIP-447](#) for the full details.
- Added a new public api `KafkaStreams.queryMetadataForKey(String, K, Serializer)` to get detailed information on the key being queried. It provides information about the partition number where the key resides in addition to hosts containing the active and standby partitions for the key.
- Provided support to query stale stores (for high availability) and the stores belonging to a specific partition by deprecating `KafkaStreams.store(String, QueryableStoreType)` and replacing it with `KafkaStreams.store(StoreQueryParameters)`.

- Added a new public api to access lag information for stores local to an instance with `KafkaStreams.allLocalStorePartitionLags()`.
- Scala 2.11 is no longer supported. See [KIP-531](#) for details.
- All Scala classes from the package `kafka.security.auth` have been deprecated. See [KIP-504](#) for details of the new Java authorizer API added in 2.4.0. Note that `kafka.security.auth.Authorizer` and `kafka.security.auth.SimpleAclAuthorizer` were deprecated in 2.4.0.
- TLSv1 and TLSv1.1 have been disabled by default since these have known security vulnerabilities. Only TLSv1.2 is now enabled by default. You can continue to use TLSv1 and TLSv1.1 by explicitly enabling these in the configuration options `ssl.protocol` and `ss1.enabled.protocols`.
- ZooKeeper has been upgraded to 3.5.7, and a ZooKeeper upgrade from 3.4.X to 3.5.7 can fail if there are no snapshot files in the 3.4 data directory. This usually happens in test upgrades where ZooKeeper 3.5.7 is trying to load an existing 3.4 data dir in which no snapshot file has been created. For more details about the issue please refer to [ZOOKEEPER-3056](#). A fix is given in [ZOOKEEPER-3056](#), which is to set `snapshot.trust.empty=true` config in `zookeeper.properties` before the upgrade.
- ZooKeeper version 3.5.7 supports TLS-encrypted connectivity to ZooKeeper both with or without client certificates, and additional Kafka configurations are available to take advantage of this. See [KIP-515](#) for details.

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, 1.0.x, 1.1.x, 2.0.x or 2.1.x or 2.2.x or 2.3.x to 2.4.0](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.10.0, 0.11.0, 1.0, 2.0, 2.2).
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from version 0.11.0.x or above, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

1. `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (0.11.0, 1.0, 1.1, 2.0, 2.1, 2.2, 2.3).
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.4.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.4 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support

the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

### Additional Upgrade Notes:

1. ZooKeeper has been upgraded to 3.5.6. ZooKeeper upgrade from 3.4.X to 3.5.6 can fail if there are no snapshot files in 3.4 data directory. This usually happens in test upgrades where ZooKeeper 3.5.6 is trying to load an existing 3.4 data dir in which no snapshot file has been created. For more details about the issue please refer to [ZOOKEEPER-3056](#). A fix is given in [ZOOKEEPER-3056](#), which is to set `snapshot.trust.empty=true` config in `zookeeper.properties` before the upgrade. But we have observed data loss in standalone cluster upgrades when using `snapshot.trust.empty=true` config. For more details about the issue please refer to [ZOOKEEPER-3644](#). So we recommend the safe workaround of copying empty `snapshot` file to the 3.4 data directory, if there are no snapshot files in 3.4 data directory. For more details about the workaround please refer to [ZooKeeper Upgrade FAQ](#).
2. An embedded Jetty based [AdminServer](#) added in ZooKeeper 3.5. AdminServer is enabled by default in ZooKeeper and is started on port 8080. AdminServer is disabled by default in the ZooKeeper config (`zookeeper.properties`) provided by the Apache Kafka distribution. Make sure to update your local `zookeeper.properties` file with `admin.enableServer=false` if you wish to disable the AdminServer. Please refer [AdminServer config](#) to configure the AdminServer.

### [Notable changes in 2.4.0](#)

- A new Admin API has been added for partition reassignments. Due to changing the way Kafka propagates reassignment information, it is possible to lose reassignment state in failure edge cases while upgrading to the new version. It is not recommended to start reassignments while upgrading.
- ZooKeeper has been upgraded from 3.4.14 to 3.5.6. TLS and dynamic reconfiguration are supported by the new version.
- The `bin/kafka-preferred-replica-election.sh` command line tool has been deprecated. It has been replaced by `bin/kafka-leader-election.sh`.
- The methods `electPreferredLeaders` in the Java `AdminClient` class have been deprecated in favor of the methods `electLeaders`.
- Scala code leveraging the `NewTopic(String, int, short)` constructor with literal values will need to explicitly call `toshort` on the second literal.
- The argument in the constructor `GroupAuthorizationException(String)` is now used to specify an exception message. Previously it referred to the group that failed authorization. This was done for consistency with other exception types and to avoid potential misuse. The constructor `TopicAuthorizationException(string)` which was previously used for a single unauthorized topic was changed similarly.
- The internal `PartitionAssignor` interface has been deprecated and replaced with a new `ConsumerPartitionAssignor` in the public API. Some methods/signatures are slightly different between the two interfaces. Users implementing a custom PartitionAssignor should migrate to the new interface as soon as possible.
- The `DefaultPartitioner` now uses a sticky partitioning strategy. This means that records for specific topic with null keys and no assigned partition will be sent to the same partition until the batch is ready to be sent. When a new batch is created, a new partition is chosen. This decreases latency to produce, but it may result in uneven distribution of records across partitions in edge cases. Generally users will not be impacted, but this difference may be noticeable in tests and other situations producing records for a very short amount of time.
- The blocking `kafkaConsumer#committed` methods have been extended to allow a list of partitions as input parameters rather than a single partition. It enables fewer request/response iterations between clients and brokers fetching for the committed offsets for the consumer group. The old overloaded functions are deprecated and we would recommend users to make their code changes to leverage the new methods (details can be found in [KIP-520](#)).
- We've introduced a new `INVALID_RECORD` error in the produce response to distinguish from the `CORRUPT_MESSAGE` error. To be more concrete, previously when a batch of records was sent as part

of a single request to the broker and one or more of the records failed the validation due to various causes (mismatch magic bytes, crc checksum errors, null key for log compacted topics, etc), the whole batch would be rejected with the same and misleading `CORRUPT_MESSAGE`, and the caller of the producer client would see the corresponding exception from either the future object of `RecordMetadata` returned from the `send` call as well as in the

`Callback#onCompletion(RecordMetadata metadata, Exception exception)` Now with the new error code and improved error messages of the exception, producer callers would be better informed about the root cause why their sent records were failed.

- We are introducing incremental cooperative rebalancing to the clients' group protocol, which allows consumers to keep all of their assigned partitions during a rebalance and at the end revoke only those which must be migrated to another consumer for overall cluster balance. The `ConsumerCoordinator` will choose the latest `RebalanceProtocol` that is commonly supported by all of the consumer's supported assignors. You can use the new built-in `CooperativestickyAssignor` or plug in your own custom cooperative assignor. To do so you must implement the `ConsumerPartitionAssignor` interface and include `RebalanceProtocol.COOPERATIVE` in the list returned by `ConsumerPartitionAssignor#supportedProtocols`. Your custom assignor can then leverage the `ownedPartitions` field in each consumer's `Subscription` to give partitions back to their previous owners whenever possible. Note that when a partition is to be reassigned to another consumer, it *must* be removed from the new assignment until it has been revoked from its original owner. Any consumer that has to revoke a partition will trigger a followup rebalance to allow the revoked partition to safely be assigned to its new owner. See the [ConsumerPartitionAssignor RebalanceProtocol javadocs](#) for more information.

To upgrade from the old (eager) protocol, which always revokes all partitions before rebalancing, to cooperative rebalancing, you must follow a specific upgrade path to get all clients on the same `ConsumerPartitionAssignor` that supports the cooperative protocol. This can be done with two rolling bounces, using the `CooperativestickyAssignor` for the example: during the first one, add "cooperative-sticky" to the list of supported assignors for each member (without removing the previous assignor -- note that if previously using the default, you must include that explicitly as well). You then bounce and/or upgrade it. Once the entire group is on 2.4+ and all members have the "cooperative-sticky" among their supported assignors, remove the other assignor(s) and perform a second rolling bounce so that by the end all members support only the cooperative protocol. For further details on the cooperative rebalancing protocol and upgrade path, see [KIP-429](#).

- There are some behavioral changes to the `ConsumerRebalanceListener`, as well as a new API. Exceptions thrown during any of the listener's three callbacks will no longer be swallowed, and will instead be re-thrown all the way up to the `Consumer.poll()` call. The `onPartitionsLost` method has been added to allow users to react to abnormal circumstances where a consumer may have lost ownership of its partitions (such as a missed rebalance) and cannot commit offsets. By default, this will simply call the existing `onPartitionsRevoked` API to align with previous behavior. Note however that `onPartitionsLost` will not be called when the set of lost partitions is empty. This means that no callback will be invoked at the beginning of the first rebalance of a new consumer joining the group.

The semantics of the `ConsumerRebalanceListener`'s callbacks are further changed when following the cooperative rebalancing protocol described above. In addition to `onPartitionsLost`, `onPartitionsRevoked` will also never be called when the set of revoked partitions is empty. The callback will generally be invoked only at the end of a rebalance, and only on the set of partitions that are being moved to another consumer. The `onPartitionsAssigned` callback will however always be called, even with an empty set of partitions, as a way to notify users of a rebalance event (this is true for both cooperative and eager). For details on the new callback semantics, see the [ConsumerRebalanceListener javadocs](#).

- The Scala trait `kafka.security.auth.Authorizer` has been deprecated and replaced with a new Java API `org.apache.kafka.server.authorizer.Authorizer`. The authorizer implementation class `kafka.security.auth.simpleAclAuthorizer` has also been deprecated and replaced with a new implementation `kafka.security.authorizer.AclAuthorizer`. `AclAuthorizer` uses features

supported by the new API to improve authorization logging and is compatible with `SimpleAclAuthorizer`. For more details, see [KIP-504](#).

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, 1.0.x, 1.1.x, 2.0.x or 2.1.x or 2.2.x to 2.3.0](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.

`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.

`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.

Alternatively, if you are upgrading from a version prior to 0.11.0.x, then

`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1).
- o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from 0.11.0.x, 1.0.x, 1.1.x, 2.0.x, or 2.1.x, and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (0.11.0, 1.0, 1.1, 2.0, 2.1, 2.2).

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.

3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.3.

4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.

5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.3 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

### [Notable changes in 2.3.0](#)

- We are introducing a new rebalancing protocol for Kafka Connect based on [incremental cooperative rebalancing](#). The new protocol does not require stopping all the tasks during a rebalancing phase between Connect workers. Instead, only the tasks that need to be exchanged between workers are stopped and they are started in a follow up rebalance. The new Connect protocol is enabled by default beginning with 2.3.0. For more details on how it works and how to enable the old behavior of eager rebalancing, checkout [incremental cooperative rebalancing design](#).
- We are introducing static membership towards consumer user. This feature reduces unnecessary rebalances during normal application upgrades or rolling bounces. For more details on how to use it, checkout [static membership design](#).
- Kafka Streams DSL switches its used store types. While this change is mainly transparent to users, there are some corner cases that may require code changes. See the [Kafka Streams upgrade](#).

[section](#) for more details.

- Kafka Streams 2.3.0 requires 0.11 message format or higher and does not work with older message format.

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, 1.0.x, 1.1.x, 2.0.x or 2.1.x to 2.2.0](#)

If you are upgrading from a version prior to 2.1.x, please see the note below about the change to the schema used to store consumer offsets. Once you have changed the `inter.broker.protocol.version` to the latest version, it will not be possible to downgrade to a version prior to 2.1.

For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1).
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)
- If you are upgrading from 0.11.0.x, 1.0.x, 1.1.x, or 2.0.x and you have not overridden the message format, then you only need to override the inter-broker protocol version.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (0.11.0, 1.0, 1.1, 2.0).
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.2.
4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.2 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

### [Notable changes in 2.2.1](#)

- Kafka Streams 2.2.1 requires 0.11 message format or higher and does not work with older message format.

### [Notable changes in 2.2.0](#)

- The default consumer group id has been changed from the empty string ( `""` ) to `null`. Consumers who use the new default group id will not be able to subscribe to topics, and fetch or commit offsets. The empty string as consumer group id is deprecated but will be supported until a future major release. Old clients that rely on the empty string group id will now have to explicitly provide it as part of their consumer config. For more information see [KIP-289](#).

- The `bin/kafka-topics.sh` command line tool is now able to connect directly to brokers with `--bootstrap-server` instead of zookeeper. The old `--zookeeper` option is still available for now. Please read [KIP-377](#) for more information.
- Kafka Streams depends on a newer version of RocksDBs that requires MacOS 10.13 or higher.

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, 1.0.x, 1.1.x, or 2.0.0 to 2.1.0](#)

**Note that 2.1.x contains a change to the internal schema used to store consumer offsets. Once the upgrade is complete, it will not be possible to downgrade to previous versions. See the rolling upgrade notes below for more detail.**

### For a rolling upgrade:

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
 Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1).
  - o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from 0.11.0.x, 1.0.x, 1.1.x, or 2.0.x and you have not overridden the message format, then you only need to override the inter-broker protocol version.

- o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (0.11.0, 1.0, 1.1, 2.0).
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it. Once you have done so, the brokers will be running the latest version and you can verify that the cluster's behavior and performance meets expectations. It is still possible to downgrade at this point if there are any problems.
  3. Once the cluster's behavior and performance has been verified, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.1.
  4. Restart the brokers one by one for the new protocol version to take effect. Once the brokers begin using the latest protocol version, it will no longer be possible to downgrade the cluster to an older version.
  5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.1 on each broker and restart them one by one. Note that the older Scala clients, which are no longer maintained, do not support the message format introduced in 0.11, so to avoid conversion costs (or to take advantage of [exactly once semantics](#)), the newer Java clients must be used.

### Additional Upgrade Notes:

1. Offset expiration semantics has slightly changed in this version. According to the new semantics, offsets of partitions in a group will not be removed while the group is subscribed to the corresponding topic and is still active (has active consumers). If group becomes empty all its offsets will be removed after default offset retention period (or the one set by broker) has passed (unless the group becomes active again). Offsets associated with standalone (simple) consumers, that do not use Kafka group management, will be removed after default offset retention period (or the one set by broker) has passed since their last commit.
2. The default for console consumer's `enable.auto.commit` property when no `group.id` is provided is now set to `false`. This is to avoid polluting the consumer coordinator cache as the auto-generated group is not likely to be used by other consumers.

3. The default value for the producer's `retries` config was changed to `Integer.MAX_VALUE`, as we introduced `delivery.timeout.ms` in [KIP-91](#), which sets an upper bound on the total time between sending a record and receiving acknowledgement from the broker. By default, the delivery timeout is set to 2 minutes.
4. By default, MirrorMaker now overrides `delivery.timeout.ms` to `Integer.MAX_VALUE` when configuring the producer. If you have overridden the value of `retries` in order to fail faster, you will instead need to override `delivery.timeout.ms`.
5. The `ListGroup` API now expects, as a recommended alternative, `Describe Group` access to the groups a user should be able to list. Even though the old `Describe Cluster` access is still supported for backward compatibility, using it for this API is not advised.
6. [KIP-336](#) deprecates the ExtendedSerializer and ExtendedDeserializer interfaces and propagates the usage of Serializer and Deserializer. ExtendedSerializer and ExtendedDeserializer were introduced with [KIP-82](#) to provide record headers for serializers and deserializers in a Java 7 compatible fashion. Now we consolidated these interfaces as Java 7 support has been dropped since.

### [Notable changes in 2.1.0](#)

- Jetty has been upgraded to 9.4.12, which excludes TLS\_RSA\_\* ciphers by default because they do not support forward secrecy, see <https://github.com/eclipse/jetty.project/issues/2807> for more information.
- Unclean leader election is automatically enabled by the controller when `unclean.leader.election.enable` config is dynamically updated by using per-topic config override.
- The `AdminClient` has added a method `AdminClient#metrics()`. Now any application using the `AdminClient` can gain more information and insight by viewing the metrics captured from the `AdminClient`. For more information see [KIP-324](#)
- Kafka now supports Zstandard compression from [KIP-110](#). You must upgrade the broker as well as clients to make use of it. Consumers prior to 2.1.0 will not be able to read from topics which use Zstandard compression, so you should not enable it for a topic until all downstream consumers are upgraded. See the KIP for more detail.

### [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, 1.0.x, or 1.1.x to 2.0.0](#)

Kafka 2.0.0 introduces wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please review the [notable changes in 2.0.0](#) before upgrading.

#### **For a rolling upgrade:**

1. Update `server.properties` on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
 Alternatively, if you are upgrading from a version prior to 0.11.0.x, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1).
  - `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from 0.11.0.x, 1.0.x, or 1.1.x and you have not overridden the message format, then you only need to override the inter-broker protocol format.

- `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (0.11.0, 1.0, 1.1).
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.

3. Once the entire cluster is upgraded, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 2.0.
4. Restart the brokers one by one for the new protocol version to take effect.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 2.0 on each broker and restart them one by one. Note that the older Scala consumer does not support the new message format introduced in 0.11, so to avoid the performance cost of down-conversion (or to take advantage of [exactly once semantics](#)), the newer Java consumer must be used.

#### **Additional Upgrade Notes:**

1. If you are willing to accept downtime, you can simply take all the brokers down, update the code and start them back up. They will start with the new protocol by default.
2. Bumping the protocol version and restarting can be done any time after the brokers are upgraded. It does not have to be immediately after. Similarly for the message format version.
3. If you are using Java8 method references in your Kafka Streams code you might need to update your code to resolve method ambiguities. Hot-swapping the jar-file only might not work.
4. ACLs should not be added to prefixed resources, (added in

KIP-290

), until all brokers in the cluster have been updated.

**NOTE:** any prefixed ACLs added to a cluster, even after the cluster is fully upgraded, will be ignored should the cluster be downgraded again.

#### **Notable changes in 2.0.0**

- [KIP-186](#) increases the default offset retention time from 1 day to 7 days. This makes it less likely to "lose" offsets in an application that commits infrequently. It also increases the active set of offsets and therefore can increase memory usage on the broker. Note that the console consumer currently enables offset commit by default and can be the source of a large number of offsets which this change will now preserve for 7 days instead of 1. You can preserve the existing behavior by setting the broker config `offsets.retention.minutes` to 1440.
- Support for Java 7 has been dropped, Java 8 is now the minimum version required.
- The default value for `ssl.endpoint.identification.algorithm` was changed to `https`, which performs hostname verification (man-in-the-middle attacks are possible otherwise). Set `ssl.endpoint.identification.algorithm` to an empty string to restore the previous behaviour.
- [KAFKA-5674](#) extends the lower interval of `max.connections.per.ip` minimum to zero and therefore allows IP-based filtering of inbound connections.
- [KIP-272](#) added API version tag to the metric `kafka.network:type=RequestMetrics, name=RequestsPerSec, request={Produce|FetchConsumer|FetchFollower|...}`. This metric now becomes `kafka.network:type=RequestMetrics, name=RequestsPerSec, request={Produce|FetchConsumer|FetchFollower|...}, version={0|1|2|3|...}`. This will impact JMX monitoring tools that do not automatically aggregate. To get the total count for a specific request type, the tool needs to be updated to aggregate across different versions.
- [KIP-225](#) changed the metric "records.lag" to use tags for topic and partition. The original version with the name format "{topic}-{partition}.records-lag" has been removed.
- The Scala consumers, which have been deprecated since 0.11.0.0, have been removed. The Java consumer has been the recommended option since 0.10.0.0. Note that the Scala consumers in 1.1.0 (and older) will continue to work even if the brokers are upgraded to 2.0.0.

- The Scala producers, which have been deprecated since 0.10.0.0, have been removed. The Java producer has been the recommended option since 0.9.0.0. Note that the behaviour of the default partitioner in the Java producer differs from the default partitioner in the Scala producers. Users migrating should consider configuring a custom partitioner that retains the previous behaviour. Note that the Scala producers in 1.1.0 (and older) will continue to work even if the brokers are upgraded to 2.0.0.
- MirrorMaker and ConsoleConsumer no longer support the Scala consumer, they always use the Java consumer.
- The ConsoleProducer no longer supports the Scala producer, it always uses the Java producer.
- A number of deprecated tools that rely on the Scala clients have been removed: ReplayLogProducer, SimpleConsumerPerformance, SimpleConsumerShell, ExportZkOffsets, ImportZkOffsets, UpdateOffsetsInZK, VerifyConsumerRebalance.
- The deprecated kafka.tools.ProducerPerformance has been removed, please use org.apache.kafka.tools.ProducerPerformance.
- New Kafka Streams configuration parameter `upgrade.from` added that allows rolling bounce upgrade from older version.
- [KIP-284](#) changed the retention time for Kafka Streams repartition topics by setting its default value to `Long.MAX_VALUE`.
- Updated `ProcessorStateManager` APIs in Kafka Streams for registering state stores to the processor topology. For more details please read the Streams [Upgrade Guide](#).
- In earlier releases, Connect's worker configuration required the `internal.key.converter` and `internal.value.converter` properties. In 2.0, these are [no longer required](#) and default to the JSON converter. You may safely remove these properties from your Connect standalone and distributed worker configurations:
 

```
internal.key.converter=org.apache.kafka.connect.json.JsonConverter
internal.key.converter.schemas.enable=false
internal.value.converter=org.apache.kafka.connect.json.JsonConverter
internal.value.converter.schemas.enable=false
```
- [KIP-266](#) adds a new consumer configuration `default.api.timeout.ms` to specify the default timeout to use for `KafkaConsumer` APIs that could block. The KIP also adds overloads for such blocking APIs to support specifying a specific timeout to use for each of them instead of using the default timeout set by `default.api.timeout.ms`. In particular, a new `poll(Duration)` API has been added which does not block for dynamic partition assignment. The old `poll(long)` API has been deprecated and will be removed in a future version. Overloads have also been added for other `KafkaConsumer` methods like `partitionsFor`, `listTopics`, `offsetsForTimes`, `beginningOffsets`, `endOffsets` and `close` that take in a `Duration`.
- Also as part of KIP-266, the default value of `request.timeout.ms` has been changed to 30 seconds. The previous value was a little higher than 5 minutes to account for maximum time that a rebalance would take. Now we treat the JoinGroup request in the rebalance as a special case and use a value derived from `max.poll.interval.ms` for the request timeout. All other request types use the timeout defined by `request.timeout.ms`.
- The internal method `kafka.admin.AdminClient.deleteRecordsBefore` has been removed. Users are encouraged to migrate to `org.apache.kafka.clients.admin.AdminClient.deleteRecords`.
- The AclCommand tool `--producer` convenience option uses the [KIP-277](#) finer grained ACL on the given topic.
- [KIP-176](#) removes the `--new-consumer` option for all consumer based tools. This option is redundant since the new consumer is automatically used if `--bootstrap-server` is defined.
- [KIP-290](#) adds the ability to define ACLs on prefixed resources, e.g. any topic starting with 'foo'.
- KIP-283

improves message down-conversion handling on Kafka broker, which has typically been a memory-intensive operation. The KIP adds a mechanism by which the operation becomes less memory intensive by down-converting chunks of partition data at a time which helps put an upper bound on memory consumption. With this improvement, there is a change in

FetchResponse

protocol behavior where the broker could send an oversized message batch towards the end of the response with an invalid offset. Such oversized messages must be ignored by consumer clients, as is done by

KafkaConsumer

KIP-283 also adds new topic and broker configurations `message.downconversion.enable` and `log.message.downconversion.enable` respectively to control whether down-conversion is enabled. When disabled, broker does not perform any down-conversion and instead sends an `UNSUPPORTED_VERSION` error to the client.

- Dynamic broker configuration options can be stored in ZooKeeper using `kafka-configs.sh` before brokers are started. This option can be used to avoid storing clear passwords in `server.properties` as all password configs may be stored encrypted in ZooKeeper.
- ZooKeeper hosts are now re-resolved if connection attempt fails. But if your ZooKeeper host names resolve to multiple addresses and some of them are not reachable, then you may need to increase the connection timeout `zookeeper.connection.timeout.ms`.

### New Protocol Versions

- [KIP-279](#): `OffsetsForLeaderEpochResponse` v1 introduces a partition-level `leader_epoch` field.
- [KIP-219](#): Bump up the protocol versions of non-cluster action requests and responses that are throttled on quota violation.
- [KIP-290](#): Bump up the protocol versions ACL create, describe and delete requests and responses.

### Upgrading a 1.1 Kafka Streams Application

- Upgrading your Streams application from 1.1 to 2.0 does not require a broker upgrade. A Kafka Streams 2.0 application can connect to 2.0, 1.1, 1.0, 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- Note that in 2.0 we have removed the public APIs that are deprecated prior to 1.0; users leveraging on those deprecated APIs need to make code changes accordingly. See [Streams API changes in 2.0.0](#) for more details.

### Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x, 0.11.0.x, or 1.0.x to 1.1.x

Kafka 1.1.0 introduces wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please review the [notable changes in 1.1.0](#) before upgrading.

**For a rolling upgrade:**

1. Update server.properties on all brokers and add the following properties.  
 CURRENT\_KAFKA\_VERSION refers to the version you are upgrading from.  
 CURRENT\_MESSAGE\_FORMAT\_VERSION refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
 Alternatively, if you are upgrading from a version prior to 0.11.0.x, then CURRENT\_MESSAGE\_FORMAT\_VERSION should be set to match CURRENT\_KAFKA\_VERSION.
  - o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0).
  - o log.message.format.version=CURRENT\_MESSAGE\_FORMAT\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from 0.11.0.x or 1.0.x and you have not overridden the message format, then you only need to override the inter-broker protocol format.

- o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (0.11.0 or 1.0).

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.

3. Once the entire cluster is upgraded, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 1.1.

4. Restart the brokers one by one for the new protocol version to take effect.

5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 1.1 on each broker and restart them one by one. Note that the older Scala consumer does not support the new message format introduced in 0.11, so to avoid the performance cost of down-conversion (or to take advantage of [exactly.once semantics](#)), the newer Java consumer must be used.

#### **Additional Upgrade Notes:**

1. If you are willing to accept downtime, you can simply take all the brokers down, update the code and start them back up. They will start with the new protocol by default.
2. Bumping the protocol version and restarting can be done any time after the brokers are upgraded. It does not have to be immediately after. Similarly for the message format version.
3. If you are using Java8 method references in your Kafka Streams code you might need to update your code to resolve method ambiguities. Hot-swapping the jar-file only might not work.

#### **Notable changes in 1.1.1**

- New Kafka Streams configuration parameter `upgrade.from` added that allows rolling bounce upgrade from version 0.10.0.x
- See the [Kafka Streams upgrade guide](#) for details about this new config.

#### **Notable changes in 1.1.0**

- The kafka artifact in Maven no longer depends on log4j or slf4j-log4j12. Similarly to the kafka-clients artifact, users can now choose the logging back-end by including the appropriate slf4j module (slf4j-log4j12, logback, etc.). The release tarball still includes log4j and slf4j-log4j12.
- [KIP-225](#) changed the metric "records.lag" to use tags for topic and partition. The original version with the name format "{topic}-{partition}.records-lag" is deprecated and will be removed in 2.0.0.
- Kafka Streams is more robust against broker communication errors. Instead of stopping the Kafka Streams client with a fatal exception, Kafka Streams tries to self-heal and reconnect to the cluster. Using the new `AdminClient` you have better control of how often Kafka Streams retries and can [configure](#) fine-grained timeouts (instead of hard coded retries as in older version).
- Kafka Streams rebalance time was reduced further making Kafka Streams more responsive.
- Kafka Connect now supports message headers in both sink and source connectors, and to manipulate them via simple message transforms. Connectors must be changed to explicitly use them. A new `HeaderConverter` is introduced to control how headers are (de)serialized, and the new "SimpleHeaderConverter" is used by default to use string representations of values.

- kafka.tools.DumpLogSegments now automatically sets deep-iteration option if print-data-log is enabled explicitly or implicitly due to any of the other options like decoder.

## [New Protocol Versions](#)

- [KIP-226](#) introduced DescribeConfigs Request/Response v1.
- [KIP-227](#) introduced Fetch Request/Response v7.

## [Upgrading a 1.0 Kafka Streams Application](#)

- Upgrading your Streams application from 1.0 to 1.1 does not require a broker upgrade. A Kafka Streams 1.1 application can connect to 1.0, 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- See [Streams API changes in 1.1.0](#) for more details.

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x, 0.10.2.x or 0.11.0.x to 1.0.0](#)

Kafka 1.0.0 introduces wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please review the [notable changes in 1.0.0](#) before upgrading.

### **For a rolling upgrade:**

1. Update server.properties on all brokers and add the following properties.  
 CURRENT\_KAFKA\_VERSION refers to the version you are upgrading from.  
 CURRENT\_MESSAGE\_FORMAT\_VERSION refers to the message format version currently in use. If you have previously overridden the message format version, you should keep its current value.  
 Alternatively, if you are upgrading from a version prior to 0.11.0.x, then CURRENT\_MESSAGE\_FORMAT\_VERSION should be set to match CURRENT\_KAFKA\_VERSION.
  - o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1, 0.10.2, 0.11.0).
  - o log.message.format.version=CURRENT\_MESSAGE\_FORMAT\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)

If you are upgrading from 0.11.0.x and you have not overridden the message format, you must set both the message format version and the inter-broker protocol version to 0.11.0.

- o inter.broker.protocol.version=0.11.0
- o log.message.format.version=0.11.0

2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.
3. Once the entire cluster is upgraded, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 1.0.
4. Restart the brokers one by one for the new protocol version to take effect.
5. If you have overridden the message format version as instructed above, then you need to do one more rolling restart to upgrade it to its latest version. Once all (or most) consumers have been upgraded to 0.11.0 or later, change `log.message.format.version` to 1.0 on each broker and restart them one by one. If you are upgrading from 0.11.0 and `log.message.format.version` is set to 0.11.0, you can update the config and skip the rolling restart. Note that the older Scala consumer does not support the new message format introduced in 0.11, so to avoid the performance cost of down-conversion (or to take advantage of [exactly once semantics](#)), the newer Java consumer must be used.

### **Additional Upgrade Notes:**

1. If you are willing to accept downtime, you can simply take all the brokers down, update the code and start them back up. They will start with the new protocol by default.
2. Bumping the protocol version and restarting can be done any time after the brokers are upgraded. It does not have to be immediately after. Similarly for the message format version.

## [Notable changes in 1.0.2](#)

- New Kafka Streams configuration parameter `upgrade.from` added that allows rolling bounce upgrade from version 0.10.0.x
- See the [Kafka Streams upgrade guide](#) for details about this new config.

## [Notable changes in 1.0.1](#)

- Restored binary compatibility of AdminClient's Options classes (e.g. `CreateTopicsOptions`, `DeleteTopicsOptions`, etc.) with 0.11.0.x. Binary (but not source) compatibility had been broken inadvertently in 1.0.0.

## [Notable changes in 1.0.0](#)

- Topic deletion is now enabled by default, since the functionality is now stable. Users who wish to retain the previous behavior should set the broker config `delete.topic.enable` to `false`. Keep in mind that topic deletion removes data and the operation is not reversible (i.e. there is no "undelete" operation)
- For topics that support timestamp search if no offset can be found for a partition, that partition is now included in the search result with a null offset value. Previously, the partition was not included in the map. This change was made to make the search behavior consistent with the case of topics not supporting timestamp search.
- If the `inter.broker.protocol.version` is 1.0 or later, a broker will now stay online to serve replicas on live log directories even if there are offline log directories. A log directory may become offline due to `IOException` caused by hardware failure. Users need to monitor the per-broker metric `offlineLogDirectoryCount` to check whether there is offline log directory.
- Added `KafkaStorageException` which is a retriable exception. `KafkaStorageException` will be converted to `NotLeaderForPartitionException` in the response if the version of the client's `FetchRequest` or `ProducerRequest` does not support `KafkaStorageException`.
- `-XX:+DisableExplicitGC` was replaced by `-XX:+ExplicitGCIInvokesConcurrent` in the default JVM settings. This helps avoid out of memory exceptions during allocation of native memory by direct buffers in some cases.
- The overridden `handleError` method implementations have been removed from the following deprecated classes in the `kafka.api` package: `FetchRequest`, `GroupCoordinatorRequest`, `offsetCommitRequest`, `offsetFetchRequest`, `offsetRequest`, `ProducerRequest`, and `TopicMetadataRequest`. This was only intended for use on the broker, but it is no longer in use and the implementations have not been maintained. A stub implementation has been retained for binary compatibility.
- The Java clients and tools now accept any string as a client-id.
- The deprecated tool `kafka-consumer-offset-checker.sh` has been removed. Use `kafka-consumer-groups.sh` to get consumer group details.
- SimpleAclAuthorizer now logs access denials to the authorizer log by default.
- Authentication failures are now reported to clients as one of the subclasses of `AuthenticationException`. No retries will be performed if a client connection fails authentication.
- Custom `saslserver` implementations may throw `SaslAuthenticationException` to provide an error message to return to clients indicating the reason for authentication failure. Implementors should take care not to include any security-critical information in the exception message that should not be leaked to unauthenticated clients.
- The `app-info` mbean registered with JMX to provide version and commit id will be deprecated and replaced with metrics providing these attributes.
- Kafka metrics may now contain non-numeric values. `org.apache.kafka.common.Metric#value()` has been deprecated and will return `0.0` in such cases to minimise the probability of breaking users who read the value of every client metric (via a `MetricsReporter` implementation or by calling the `metrics()` method). `org.apache.kafka.common.Metric#metricValue()` can be used to retrieve numeric and non-numeric metric values.

- Every Kafka rate metric now has a corresponding cumulative count metric with the suffix `-total` to simplify downstream processing. For example, `records-consumed-rate` has a corresponding metric named `records-consumed-total`.
- Mx4j will only be enabled if the system property `kafka_mx4jenable` is set to `true`. Due to a logic inversion bug, it was previously enabled by default and disabled if `kafka_mx4jenable` was set to `true`.
- The package `org.apache.kafka.common.security.auth` in the clients jar has been made public and added to the javadocs. Internal classes which had previously been located in this package have been moved elsewhere.
- When using an Authorizer and a user doesn't have required permissions on a topic, the broker will return `TOPIC_AUTHORIZATION_FAILED` errors to requests irrespective of topic existence on broker. If the user have required permissions and the topic doesn't exists, then the `UNKNOWN_TOPIC_OR_PARTITION` error code will be returned.
- config/consumer.properties file updated to use new consumer config properties.

### New Protocol Versions

- [KIP-112](#): LeaderAndIsrRequest v1 introduces a partition-level `is_new` field.
- [KIP-112](#): UpdateMetadataRequest v4 introduces a partition-level `offline_replicas` field.
- [KIP-112](#): MetadataResponse v5 introduces a partition-level `offline_replicas` field.
- [KIP-112](#): ProduceResponse v4 introduces error code for KafkaStorageException.
- [KIP-112](#): FetchResponse v6 introduces error code for KafkaStorageException.
- [KIP-152](#): SaslAuthenticate request has been added to enable reporting of authentication failures. This request will be used if the SaslHandshake request version is greater than 0.

### Upgrading a 0.11.0 Kafka Streams Application

- Upgrading your Streams application from 0.11.0 to 1.0 does not require a broker upgrade. A Kafka Streams 1.0 application can connect to 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though). However, Kafka Streams 1.0 requires 0.10 message format or newer and does not work with older message formats.
- If you are monitoring on streams metrics, you will need make some changes to the metrics names in your reporting and monitoring code, because the metrics sensor hierarchy was changed.
- There are a few public APIs including `ProcessorContext#schedule()`, `Processor#punctuate()` and `KstreamBuilder`, `TopologyBuilder` are being deprecated by new APIs. We recommend making corresponding code changes, which should be very minor since the new APIs look quite similar, when you upgrade.
- See [Streams API changes in 1.0.0](#) for more details.

### Upgrading a 0.10.2 Kafka Streams Application

- Upgrading your Streams application from 0.10.2 to 1.0 does not require a broker upgrade. A Kafka Streams 1.0 application can connect to 1.0, 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- If you are monitoring on streams metrics, you will need make some changes to the metrics names in your reporting and monitoring code, because the metrics sensor hierarchy was changed.
- There are a few public APIs including `ProcessorContext#schedule()`, `Processor#punctuate()` and `KstreamBuilder`, `TopologyBuilder` are being deprecated by new APIs. We recommend making corresponding code changes, which should be very minor since the new APIs look quite similar, when you upgrade.
- If you specify customized `key.serde`, `value.serde` and `timestamp.extractor` in configs, it is recommended to use their replaced configure parameter as these configs are deprecated.
- See [Streams API changes in 0.11.0](#) for more details.

## [Upgrading a 0.10.1 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.1 to 1.0 does not require a broker upgrade. A Kafka Streams 1.0 application can connect to 1.0, 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- You need to recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- If you are monitoring on streams metrics, you will need make some changes to the metrics names in your reporting and monitoring code, because the metrics sensor hierarchy was changed.
- There are a few public APIs including `ProcessorContext#schedule()`, `Processor#punctuate()` and `KStreamBuilder`, `TopologyBuilder` are being deprecated by new APIs. We recommend making corresponding code changes, which should be very minor since the new APIs look quite similar, when you upgrade.
- If you specify customized `key.serde`, `value.serde` and `timestamp.extractor` in configs, it is recommended to use their replaced configure parameter as these configs are deprecated.
- If you use a custom (i.e., user implemented) timestamp extractor, you will need to update this code, because the `TimestampExtractor` interface was changed.
- If you register custom metrics, you will need to update this code, because the `StreamsMetric` interface was changed.
- See [Streams API changes in 1.0.0](#), [Streams API changes in 0.11.0](#) and [Streams API changes in 0.10.2](#) for more details.

## [Upgrading a 0.10.0 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.0 to 1.0 does require a [broker upgrade](#) because a Kafka Streams 1.0 application can only connect to 0.1, 0.11.0, 0.10.2, or 0.10.1 brokers.
- There are couple of API changes, that are not backward compatible (cf. [Streams API changes in 1.0.0](#), [Streams API changes in 0.11.0](#), [Streams API changes in 0.10.2](#), and [Streams API changes in 0.10.1](#) for more details). Thus, you need to update and recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- Upgrading from 0.10.0.x to 1.0.2 requires two rolling bounces with config

```
upgrade.from="0.10.0"
```

set for first upgrade phase (cf.

KIP-268

- ). As an alternative, an offline upgrade is also possible.
  - prepare your application instances for a rolling bounce and make sure that config `upgrade.from` is set to `"0.10.0"` for new version 0.11.0.3
  - bounce each instance of your application once
  - prepare your newly deployed 1.0.2 application instances for a second round of rolling bounces; make sure to remove the value for config `upgrade.from`
  - bounce each instance of your application once more to complete the upgrade
- Upgrading from 0.10.0.x to 1.0.0 or 1.0.1 requires an offline upgrade (rolling bounce upgrade is not supported)
  - stop all old (0.10.0.x) application instances
  - update your code and swap old code and jar file with new code and new jar file
  - restart all new (1.0.0 or 1.0.1) application instances

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x, 0.10.1.x or 0.10.2.x to 0.11.0.0](#)

Kafka 0.11.0.0 introduces a new message format version as well as wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please review the [notable changes in 0.11.0](#) before upgrading.

Starting with version 0.10.2, Java clients (producer and consumer) have acquired the ability to communicate with older brokers. Version 0.11.0 clients can talk to version 0.10.0 or newer brokers. However, if your brokers are older than 0.10.0, you must upgrade all the brokers in the Kafka cluster before upgrading your clients. Version 0.11.0 brokers support 0.8.x and newer clients.

### **For a rolling upgrade:**

1. Update server.properties on all brokers and add the following properties.  
`CURRENT_KAFKA_VERSION` refers to the version you are upgrading from.  
`CURRENT_MESSAGE_FORMAT_VERSION` refers to the current message format version currently in use. If you have not overridden the message format previously, then  
`CURRENT_MESSAGE_FORMAT_VERSION` should be set to match `CURRENT_KAFKA_VERSION`.
  - o `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2, 0.9.0, 0.10.0, 0.10.1 or 0.10.2).
  - o `log.message.format.version=CURRENT_MESSAGE_FORMAT_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.
3. Once the entire cluster is upgraded, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 0.11.0, but do not change `log.message.format.version` yet.
4. Restart the brokers one by one for the new protocol version to take effect.
5. Once all (or most) consumers have been upgraded to 0.11.0 or later, then change `log.message.format.version` to 0.11.0 on each broker and restart them one by one. Note that the older Scala consumer does not support the new message format, so to avoid the performance cost of down-conversion (or to take advantage of [exactly once semantics](#)), the new Java consumer must be used.

### **Additional Upgrade Notes:**

1. If you are willing to accept downtime, you can simply take all the brokers down, update the code and start them back up. They will start with the new protocol by default.
2. Bumping the protocol version and restarting can be done any time after the brokers are upgraded. It does not have to be immediately after. Similarly for the message format version.
3. It is also possible to enable the 0.11.0 message format on individual topics using the topic admin tool (`bin/kafka-topics.sh`) prior to updating the global setting `log.message.format.version`.
4. If you are upgrading from a version prior to 0.10.0, it is NOT necessary to first update the message format to 0.10.0 before you switch to 0.11.0.

## [Upgrading a 0.10.2 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.2 to 0.11.0 does not require a broker upgrade. A Kafka Streams 0.11.0 application can connect to 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- If you specify customized `key.serde`, `value.serde` and `timestamp.extractor` in configs, it is recommended to use their replaced configure parameter as these configs are deprecated.
- See [Streams API changes in 0.11.0](#) for more details.

## [Upgrading a 0.10.1 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.1 to 0.11.0 does not require a broker upgrade. A Kafka Streams 0.11.0 application can connect to 0.11.0, 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- You need to recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- If you specify customized `key.serde`, `value.serde` and `timestamp.extractor` in configs, it is recommended to use their replaced configure parameter as these configs are deprecated.
- If you use a custom (i.e., user implemented) timestamp extractor, you will need to update this code, because the `TimestampExtractor` interface was changed.
- If you register custom metrics, you will need to update this code, because the `streamsMetric` interface was changed.
- See [Streams API changes in 0.11.0](#) and [Streams API changes in 0.10.2](#) for more details.

## [Upgrading a 0.10.0 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.0 to 0.11.0 does require a [broker upgrade](#) because a Kafka Streams 0.11.0 application can only connect to 0.11.0, 0.10.2, or 0.10.1 brokers.
- There are couple of API changes, that are not backward compatible (cf. [Streams API changes in 0.11.0](#), [Streams API changes in 0.10.2](#), and [Streams API changes in 0.10.1](#) for more details). Thus, you need to update and recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- Upgrading from 0.10.0.x to 0.11.0.3 requires two rolling bounces with config

```
upgrade.from="0.10.0"
```

set for first upgrade phase (cf.

KIP-268

- ). As an alternative, an offline upgrade is also possible.
  - prepare your application instances for a rolling bounce and make sure that config `upgrade.from` is set to `"0.10.0"` for new version 0.11.0.3
  - bounce each instance of your application once
  - prepare your newly deployed 0.11.0.3 application instances for a second round of rolling bounces; make sure to remove the value for config `upgrade.from`
  - bounce each instance of your application once more to complete the upgrade
- Upgrading from 0.10.0.x to 0.11.0.0, 0.11.0.1, or 0.11.0.2 requires an offline upgrade (rolling bounce upgrade is not supported)
  - stop all old (0.10.0.x) application instances
  - update your code and swap old code and jar file with new code and new jar file
  - restart all new (0.11.0.0, 0.11.0.1, or 0.11.0.2) application instances

## [Notable changes in 0.11.0.3](#)

- New Kafka Streams configuration parameter `upgrade.from` added that allows rolling bounce upgrade from version 0.10.0.x
- See the [Kafka Streams upgrade guide](#) for details about this new config.

## [Notable changes in 0.11.0.0](#)

- Unclean leader election is now disabled by default. The new default favors durability over availability. Users who wish to retain the previous behavior should set the broker config `unclean.leader.election.enable` to `true`.
- Producer configs `block.on.buffer.full`, `metadata.fetch.timeout.ms` and `timeout.ms` have been removed. They were initially deprecated in Kafka 0.9.0.0.
- The `offsets.topic.replication.factor` broker config is now enforced upon auto topic creation. Internal auto topic creation will fail with a GROUP\_COORDINATOR\_NOT\_AVAILABLE error until the cluster size meets this replication factor requirement.
- When compressing data with snappy, the producer and broker will use the compression scheme's default block size (2 x 32 KB) instead of 1 KB in order to improve the compression ratio. There have been reports of data compressed with the smaller block size being 50% larger than when compressed with the larger block size. For the snappy case, a producer with 5000 partitions will require an additional 315 MB of JVM heap.
- Similarly, when compressing data with gzip, the producer and broker will use 8 KB instead of 1 KB as the buffer size. The default for gzip is excessively low (512 bytes).
- The broker configuration `max.message.bytes` now applies to the total size of a batch of messages. Previously the setting applied to batches of compressed messages, or to non-compressed messages individually. A message batch may consist of only a single message, so in most cases, the limitation on the size of individual messages is only reduced by the overhead of the batch format. However, there are some subtle implications for message format conversion (see [below](#) for more detail). Note also that while previously the broker would ensure that at least one message is returned in each fetch request (regardless of the total and partition-level fetch sizes), the same behavior now applies to one message batch.
- GC log rotation is enabled by default, see KAFKA-3754 for details.
- Deprecated constructors of RecordMetadata, MetricName and Cluster classes have been removed.
- Added user headers support through a new Headers interface providing user headers read and write access.
- ProducerRecord and ConsumerRecord expose the new Headers API via `Headers headers()` method call.
- ExtendedSerializer and ExtendedDeserializer interfaces are introduced to support serialization and deserialization for headers. Headers will be ignored if the configured serializer and deserializer are not the above classes.
- A new config, `group.initial.rebalance.delay.ms`, was introduced. This config specifies the time, in milliseconds, that the GroupCoordinator will delay the initial consumer rebalance. The rebalance will be further delayed by the value of `group.initial.rebalance.delay.ms` as new members join the group, up to a maximum of `max.poll.interval.ms`. The default value for this is 3 seconds. During development and testing it might be desirable to set this to 0 in order to not delay test execution time.
- `org.apache.kafka.common.Cluster#partitionsForTopic`, `partitionsForNode` and `availablePartitionsForTopic` methods will return an empty list instead of `null` (which is considered a bad practice) in case the metadata for the required topic does not exist.
- Streams API configuration parameters `timestamp.extractor`, `key.serde`, and `value.serde` were deprecated and replaced by `default.timestamp.extractor`, `default.key.serde`, and `default.value.serde`, respectively.
- For offset commit failures in the Java consumer's `commitAsync` APIs, we no longer expose the underlying cause when instances of `RetriableCommitFailedException` are passed to the commit callback. See [KAFKA-5052](#) for more detail.

## [New Protocol Versions](#)

- [KIP-107](#): FetchRequest v5 introduces a partition-level `log_start_offset` field.
- [KIP-107](#): FetchResponse v5 introduces a partition-level `log_start_offset` field.
- [KIP-82](#): ProduceRequest v3 introduces an array of `header` in the message protocol, containing `key` field and `value` field.
- [KIP-82](#): FetchResponse v5 introduces an array of `header` in the message protocol, containing `key` field and `value` field.

## [Notes on Exactly Once Semantics](#)

Kafka 0.11.0 includes support for idempotent and transactional capabilities in the producer. Idempotent delivery ensures that messages are delivered exactly once to a particular topic partition during the lifetime of a single producer. Transactional delivery allows producers to send data to multiple partitions such that either all messages are successfully delivered, or none of them are. Together, these capabilities enable "exactly once semantics" in Kafka. More details on these features are available in the user guide, but below we add a few specific notes on enabling them in an upgraded cluster. Note that enabling EoS is not required and there is no impact on the broker's behavior if unused.

1. Only the new Java producer and consumer support exactly once semantics.
2. These features depend crucially on the [0.11.0 message format](#). Attempting to use them on an older format will result in unsupported version errors.
3. Transaction state is stored in a new internal topic `__transaction_state`. This topic is not created until the first attempt to use a transactional request API. Similar to the consumer offsets topic, there are several settings to control the topic's configuration. For example, `transaction.state.log.min_isr` controls the minimum ISR for this topic. See the configuration section in the user guide for a full list of options.
4. For secure clusters, the transactional APIs require new ACLs which can be turned on with the `bin/kafka-acls.sh` tool.
5. EoS in Kafka introduces new request APIs and modifies several existing ones. See [KIP-98](#) for the full details

## [Notes on the new message format in 0.11.0](#)

The 0.11.0 message format includes several major enhancements in order to support better delivery semantics for the producer (see [KIP-98](#)) and improved replication fault tolerance (see [KIP-101](#)). Although the new format contains more information to make these improvements possible, we have made the batch format much more efficient. As long as the number of messages per batch is more than 2, you can expect lower overall overhead. For smaller batches, however, there may be a small performance impact. See [here](#) for the results of our initial performance analysis of the new message format. You can also find more detail on the message format in the [KIP-98](#) proposal.

One of the notable differences in the new message format is that even uncompressed messages are stored together as a single batch. This has a few implications for the broker configuration `max.message.bytes`, which limits the size of a single batch. First, if an older client produces messages to a topic partition using the old format, and the messages are individually smaller than `max.message.bytes`, the broker may still reject them after they are merged into a single batch during the up-conversion process. Generally this can happen when the aggregate size of the individual messages is larger than `max.message.bytes`. There is a similar effect for older consumers reading messages down-converted from the new format: if the fetch size is not set at least as large as `max.message.bytes`, the consumer may not be able to make progress even if the individual uncompressed messages are smaller than the configured fetch size. This behavior does not impact the Java client for 0.10.1.0 and later since it uses an updated fetch protocol which ensures that at least one message can be returned even if it exceeds the fetch size. To get around these problems, you should ensure 1) that the producer's batch size is not set larger than `max.message.bytes`, and 2) that the consumer's fetch size is set at least as large as `max.message.bytes`.

Most of the discussion on the performance impact of [upgrading to the 0.10.0 message format](#) remains pertinent to the 0.11.0 upgrade. This mainly affects clusters that are not secured with TLS since "zero-copy" transfer is already not possible in that case. In order to avoid the cost of down-conversion, you should ensure that consumer applications are upgraded to the latest 0.11.0 client. Significantly, since the old consumer has been deprecated in 0.11.0.0, it does not support the new message format. You must upgrade to use the new consumer to use the new message format without the cost of down-conversion. Note that 0.11.0 consumers support backwards compatibility with 0.10.0 brokers and upward, so it is possible to upgrade the clients first before the brokers.

## [Upgrading from 0.8.x, 0.9.x, 0.10.0.x or 0.10.1.x to 0.10.2.0](#)

0.10.2.0 has wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please review the [notable changes in 0.10.2.0](#) before upgrading.

Starting with version 0.10.2, Java clients (producer and consumer) have acquired the ability to communicate with older brokers. Version 0.10.2 clients can talk to version 0.10.0 or newer brokers. However, if your brokers are older than 0.10.0, you must upgrade all the brokers in the Kafka cluster before upgrading your clients. Version 0.10.2 brokers support 0.8.x and newer clients.

### **For a rolling upgrade:**

1. Update server.properties file on all brokers and add the following properties:
  - o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 0.8.2, 0.9.0, 0.10.0 or 0.10.1).
  - o log.message.format.version=CURRENT\_KAFKA\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.
3. Once the entire cluster is upgraded, bump the protocol version by editing inter.broker.protocol.version and setting it to 0.10.2.
4. If your previous message format is 0.10.0, change log.message.format.version to 0.10.2 (this is a no-op as the message format is the same for 0.10.0, 0.10.1 and 0.10.2). If your previous message format version is lower than 0.10.0, do not change log.message.format.version yet - this parameter should only change once all consumers have been upgraded to 0.10.0.0 or later.
5. Restart the brokers one by one for the new protocol version to take effect.
6. If log.message.format.version is still lower than 0.10.0 at this point, wait until all consumers have been upgraded to 0.10.0 or later, then change log.message.format.version to 0.10.2 on each broker and restart them one by one.

**Note:** If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

**Note:** Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

## [Upgrading a 0.10.1 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.1 to 0.10.2 does not require a broker upgrade. A Kafka Streams 0.10.2 application can connect to 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- You need to recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- If you use a custom (i.e., user implemented) timestamp extractor, you will need to update this code, because the `TimestampExtractor` interface was changed.
- If you register custom metrics, you will need to update this code, because the `streamsMetric` interface was changed.
- See [Streams API changes in 0.10.2](#) for more details.

## [Upgrading a 0.10.0 Kafka Streams Application](#)

- Upgrading your Streams application from 0.10.0 to 0.10.2 does require a [broker upgrade](#) because a Kafka Streams 0.10.2 application can only connect to 0.10.2 or 0.10.1 brokers.
- There are couple of API changes, that are not backward compatible (cf. [Streams API changes in 0.10.2](#) for more details). Thus, you need to update and recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- Upgrading from 0.10.0.x to 0.10.2.2 requires two rolling bounces with config

```
upgrade.from="0.10.0"
```

set for first upgrade phase (cf.

KIP-268

- ). As an alternative, an offline upgrade is also possible.
  - prepare your application instances for a rolling bounce and make sure that config `upgrade.from` is set to `"0.10.0"` for new version 0.10.2.2
  - bounce each instance of your application once
  - prepare your newly deployed 0.10.2.2 application instances for a second round of rolling bounces; make sure to remove the value for config `upgrade.from`
  - bounce each instance of your application once more to complete the upgrade
- Upgrading from 0.10.0.x to 0.10.2.0 or 0.10.2.1 requires an offline upgrade (rolling bounce upgrade is not supported)
  - stop all old (0.10.0.x) application instances
  - update your code and swap old code and jar file with new code and new jar file
  - restart all new (0.10.2.0 or 0.10.2.1) application instances

## [Notable changes in 0.10.2.2](#)

- New configuration parameter `upgrade.from` added that allows rolling bounce upgrade from version 0.10.0.x

## [Notable changes in 0.10.2.1](#)

- The default values for two configurations of the StreamsConfig class were changed to improve the resiliency of Kafka Streams applications. The internal Kafka Streams producer `retries` default value was changed from 0 to 10. The internal Kafka Streams consumer `max.poll.interval.ms` default value was changed from 300000 to `Integer.MAX_VALUE`.

## [Notable changes in 0.10.2.0](#)

- The Java clients (producer and consumer) have acquired the ability to communicate with older brokers. Version 0.10.2 clients can talk to version 0.10.0 or newer brokers. Note that some features are not available or are limited when older brokers are used.
- Several methods on the Java consumer may now throw `InterruptedException` if the calling thread is interrupted. Please refer to the `KafkaConsumer` Javadoc for a more in-depth explanation of this change.
- Java consumer now shuts down gracefully. By default, the consumer waits up to 30 seconds to complete pending requests. A new close API with timeout has been added to `KafkaConsumer` to control the maximum wait time.
- Multiple regular expressions separated by commas can be passed to MirrorMaker with the new Java consumer via the `--whitelist` option. This makes the behaviour consistent with MirrorMaker when

used the old Scala consumer.

- Upgrading your Streams application from 0.10.1 to 0.10.2 does not require a broker upgrade. A Kafka Streams 0.10.2 application can connect to 0.10.2 and 0.10.1 brokers (it is not possible to connect to 0.10.0 brokers though).
- The Zookeeper dependency was removed from the Streams API. The Streams API now uses the Kafka protocol to manage internal topics instead of modifying Zookeeper directly. This eliminates the need for privileges to access Zookeeper directly and "StreamsConfig.ZOOKEEPER\_CONFIG" should not be set in the Streams app any more. If the Kafka cluster is secured, Streams apps must have the required security privileges to create new topics.
- Several new fields including "security.protocol", "connections.max.idle.ms", "retry.backoff.ms", "reconnect.backoff.ms" and "request.timeout.ms" were added to StreamsConfig class. User should pay attention to the default values and set these if needed. For more details please refer to [3.5 Kafka Streams Configs](#).

## [New Protocol Versions](#)

- [KIP-88](#): OffsetFetchRequest v2 supports retrieval of offsets for all topics if the `topics` array is set to `null`.
- [KIP-88](#): OffsetFetchResponse v2 introduces a top-level `error_code` field.
- [KIP-103](#): UpdateMetadataRequest v3 introduces a `listener_name` field to the elements of the `end_points` array.
- [KIP-108](#): CreateTopicsRequest v1 introduces a `validate_only` field.
- [KIP-108](#): CreateTopicsResponse v1 introduces an `error_message` field to the elements of the `topic_errors` array.

## [Upgrading from 0.8.x, 0.9.x or 0.10.0.X to 0.10.1.0](#)

0.10.1.0 has wire protocol changes. By following the recommended rolling upgrade plan below, you guarantee no downtime during the upgrade. However, please notice the [Potential breaking changes in 0.10.1.0](#) before upgrade.

Note: Because new protocols are introduced, it is important to upgrade your Kafka clusters before upgrading your clients (i.e. 0.10.1.x clients only support 0.10.1.x or later brokers while 0.10.1.x brokers also support older clients).

### **For a rolling upgrade:**

1. Update server.properties file on all brokers and add the following properties:
  - `inter.broker.protocol.version=CURRENT_KAFKA_VERSION` (e.g. 0.8.2.0, 0.9.0.0 or 0.10.0.0).
  - `log.message.format.version=CURRENT_KAFKA_VERSION` (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)
2. Upgrade the brokers one at a time: shut down the broker, update the code, and restart it.
3. Once the entire cluster is upgraded, bump the protocol version by editing `inter.broker.protocol.version` and setting it to 0.10.1.0.
4. If your previous message format is 0.10.0, change `log.message.format.version` to 0.10.1 (this is a no-op as the message format is the same for both 0.10.0 and 0.10.1). If your previous message format version is lower than 0.10.0, do not change `log.message.format.version` yet - this parameter should only change once all consumers have been upgraded to 0.10.0.0 or later.
5. Restart the brokers one by one for the new protocol version to take effect.
6. If `log.message.format.version` is still lower than 0.10.0 at this point, wait until all consumers have been upgraded to 0.10.0 or later, then change `log.message.format.version` to 0.10.1 on each broker and restart them one by one.

**Note:** If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

**Note:** Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

### Potential breaking changes in 0.10.1.0

- The log retention time is no longer based on last modified time of the log segments. Instead it will be based on the largest timestamp of the messages in a log segment.
- The log rolling time is no longer depending on log segment create time. Instead it is now based on the timestamp in the messages. More specifically, if the timestamp of the first message in the segment is T, the log will be rolled out when a new message has a timestamp greater than or equal to  $T + \text{log.roll.ms}$ .
- The open file handlers of 0.10.0 will increase by ~33% because of the addition of time index files for each segment.
- The time index and offset index share the same index size configuration. Since each time index entry is 1.5x the size of offset index entry. User may need to increase `log.index.size.max.bytes` to avoid potential frequent log rolling.
- Due to the increased number of index files, on some brokers with large amount the log segments (e.g. >15K), the log loading process during the broker startup could be longer. Based on our experiment, setting the `num.recovery.threads.per.data.dir` to one may reduce the log loading time.

### Upgrading a 0.10.0 Kafka Streams Application

- Upgrading your Streams application from 0.10.0 to 0.10.1 does require a [broker upgrade](#) because a Kafka Streams 0.10.1 application can only connect to 0.10.1 brokers.
- There are couple of API changes, that are not backward compatible (cf. [Streams API changes in 0.10.1](#) for more details). Thus, you need to update and recompile your code. Just swapping the Kafka Streams library jar file will not work and will break your application.
- Upgrading from 0.10.0.x to 0.10.1.2 requires two rolling bounces with config

```
upgrade.from="0.10.0"
```

set for first upgrade phase (cf.

KIP-268

- ). As an alternative, an offline upgrade is also possible.
  - prepare your application instances for a rolling bounce and make sure that config `upgrade.from` is set to `"0.10.0"` for new version 0.10.1.2
  - bounce each instance of your application once
  - prepare your newly deployed 0.10.1.2 application instances for a second round of rolling bounces; make sure to remove the value for config `upgrade.from`
  - bounce each instance of your application once more to complete the upgrade
- Upgrading from 0.10.0.x to 0.10.1.0 or 0.10.1.1 requires an offline upgrade (rolling bounce upgrade is not supported)
  - stop all old (0.10.0.x) application instances
  - update your code and swap old code and jar file with new code and new jar file
  - restart all new (0.10.1.0 or 0.10.1.1) application instances

## Notable changes in 0.10.1.0

- The new Java consumer is no longer in beta and we recommend it for all new development. The old Scala consumers are still supported, but they will be deprecated in the next release and will be removed in a future major release.
- The `--new-consumer / --new.consumer` switch is no longer required to use tools like MirrorMaker and the Console Consumer with the new consumer; one simply needs to pass a Kafka broker to connect to instead of the ZooKeeper ensemble. In addition, usage of the Console Consumer with the old consumer has been deprecated and it will be removed in a future major release.
- Kafka clusters can now be uniquely identified by a cluster id. It will be automatically generated when a broker is upgraded to 0.10.1.0. The cluster id is available via the `kafka.server:type=KafkaServer,name=ClusterId` metric and it is part of the Metadata response. Serializers, client interceptors and metric reporters can receive the cluster id by implementing the `ClusterResourceListener` interface.
- The BrokerState "RunningAsController" (value 4) has been removed. Due to a bug, a broker would only be in this state briefly before transitioning out of it and hence the impact of the removal should be minimal. The recommended way to detect if a given broker is the controller is via the `kafka.controller:type=KafkaController,name=ActiveControllerCount` metric.
- The new Java Consumer now allows users to search offsets by timestamp on partitions.
- The new Java Consumer now supports heartbeating from a background thread. There is a new configuration `max.poll.interval.ms` which controls the maximum time between poll invocations before the consumer will proactively leave the group (5 minutes by default). The value of the configuration `request.timeout.ms` (default to 30 seconds) must always be smaller than `max.poll.interval.ms` (default to 5 minutes), since that is the maximum time that a `JoinGroup` request can block on the server while the consumer is rebalance. Finally, the default value of `session.timeout.ms` has been adjusted down to 10 seconds, and the default value of `max.poll.records` has been changed to 500.
- When using an Authorizer and a user doesn't have **Describe** authorization on a topic, the broker will no longer return `TOPIC_AUTHORIZATION_FAILED` errors to requests since this leaks topic names. Instead, the `UNKNOWN_TOPIC_OR_PARTITION` error code will be returned. This may cause unexpected timeouts or delays when using the producer and consumer since Kafka clients will typically retry automatically on unknown topic errors. You should consult the client logs if you suspect this could be happening.
- Fetch responses have a size limit by default (50 MB for consumers and 10 MB for replication). The existing per partition limits also apply (1 MB for consumers and replication). Note that neither of these limits is an absolute maximum as explained in the next point.
- Consumers and replicas can make progress if a message larger than the response/partition size limit is found. More concretely, if the first message in the first non-empty partition of the fetch is larger than either or both limits, the message will still be returned.
- Overloaded constructors were added to `kafka.api.FetchRequest` and `kafka.javaapi.FetchRequest` to allow the caller to specify the order of the partitions (since order is significant in v3). The previously existing constructors were deprecated and the partitions are shuffled before the request is sent to avoid starvation issues.

## New Protocol Versions

- `ListOffsetRequest` v1 supports accurate offset search based on timestamps.
- `MetadataResponse` v2 introduces a new field: "cluster\_id".
- `FetchRequest` v3 supports limiting the response size (in addition to the existing per partition limit), it returns messages bigger than the limits if required to make progress and the order of partitions in the request is now significant.
- `JoinGroup` v1 introduces a new field: "rebalance\_timeout".

## Upgrading from 0.8.x or 0.9.x to 0.10.0.0

0.10.0.0 has [potential breaking changes](#) (please review before upgrading) and possible [performance impact following the upgrade](#). By following the recommended rolling upgrade plan below, you guarantee no downtime and no performance impact during and following the upgrade.

Note: Because new protocols are introduced, it is important to upgrade your Kafka clusters before upgrading your clients.

**Notes to clients with version 0.9.0.0:** Due to a bug introduced in 0.9.0.0, clients that depend on ZooKeeper (old Scala high-level Consumer and MirrorMaker if used with the old consumer) will not work with 0.10.0.x brokers. Therefore, 0.9.0.0 clients should be upgraded to 0.9.0.1 **before** brokers are upgraded to 0.10.0.x. This step is not necessary for 0.8.X or 0.9.0.1 clients.

### **For a rolling upgrade:**

1. Update server.properties file on all brokers and add the following properties:
  - o inter.broker.protocol.version=CURRENT\_KAFKA\_VERSION (e.g. 0.8.2 or 0.9.0.0).
  - o log.message.format.version=CURRENT\_KAFKA\_VERSION (See [potential performance impact following the upgrade](#) for the details on what this configuration does.)
2. Upgrade the brokers. This can be done a broker at a time by simply bringing it down, updating the code, and restarting it.
3. Once the entire cluster is upgraded, bump the protocol version by editing inter.broker.protocol.version and setting it to 0.10.0.0. NOTE: You shouldn't touch log.message.format.version yet - this parameter should only change once all consumers have been upgraded to 0.10.0.0
4. Restart the brokers one by one for the new protocol version to take effect.
5. Once all consumers have been upgraded to 0.10.0, change log.message.format.version to 0.10.0 on each broker and restart them one by one.

**Note:** If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

**Note:** Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

### Potential performance impact following upgrade to 0.10.0.0

The message format in 0.10.0 includes a new timestamp field and uses relative offsets for compressed messages. The on disk message format can be configured through log.message.format.version in the server.properties file. The default on-disk message format is 0.10.0. If a consumer client is on a version before 0.10.0.0, it only understands message formats before 0.10.0. In this case, the broker is able to convert messages from the 0.10.0 format to an earlier format before sending the response to the consumer on an older version. However, the broker can't use zero-copy transfer in this case. Reports from the Kafka community on the performance impact have shown CPU utilization going from 20% before to 100% after an upgrade, which forced an immediate upgrade of all clients to bring performance back to normal. To avoid such message conversion before consumers are upgraded to 0.10.0.0, one can set log.message.format.version to 0.8.2 or 0.9.0 when upgrading the broker to 0.10.0.0. This way, the broker can still use zero-copy transfer to send the data to the old consumers. Once consumers are upgraded, one can change the message format to 0.10.0 on the broker and enjoy the new message format that includes new timestamp and improved compression. The conversion is supported to ensure compatibility and can be useful to support a few apps that have not updated to newer clients yet, but is impractical to support all consumer traffic on even an overprovisioned cluster. Therefore, it is critical to avoid the message conversion as much as possible when brokers have been upgraded but the majority of clients have not.

For clients that are upgraded to 0.10.0.0, there is no performance impact.

**Note:** By setting the message format version, one certifies that all existing messages are on or below that message format version. Otherwise consumers before 0.10.0.0 might break. In particular, after the message format is set to 0.10.0, one should not change it back to an earlier format as it may break consumers on versions before 0.10.0.0.

**Note:** Due to the additional timestamp introduced in each message, producers sending small messages may see a message throughput degradation because of the increased overhead. Likewise, replication now transmits an additional 8 bytes per message. If you're running close to the network capacity of your cluster, it's possible that you'll overwhelm the network cards and see failures and performance issues due to the overload.

**Note:** If you have enabled compression on producers, you may notice reduced producer throughput and/or lower compression rate on the broker in some cases. When receiving compressed messages, 0.10.0 brokers avoid recompressing the messages, which in general reduces the latency and improves the throughput. In certain cases, however, this may reduce the batching size on the producer, which could lead to worse throughput. If this happens, users can tune linger.ms and batch.size of the producer for better throughput. In addition, the producer buffer used for compressing messages with snappy is smaller than the one used by the broker, which may have a negative impact on the compression ratio for the messages on disk. We intend to make this configurable in a future Kafka release.

### Potential breaking changes in 0.10.0.0

- Starting from Kafka 0.10.0.0, the message format version in Kafka is represented as the Kafka version. For example, message format 0.9.0 refers to the highest message version supported by Kafka 0.9.0.
- Message format 0.10.0 has been introduced and it is used by default. It includes a timestamp field in the messages and relative offsets are used for compressed messages.
- ProduceRequest/Response v2 has been introduced and it is used by default to support message format 0.10.0
- FetchRequest/Response v2 has been introduced and it is used by default to support message format 0.10.0
- MessageFormatter interface was changed from `def writeTo(key: Array[Byte], value: Array[Byte], output: PrintStream)` to `def writeTo(consumerRecord: ConsumerRecord[Array[Byte], Array[Byte]], output: PrintStream)`
- MessageReader interface was changed from `def readMessage(): KeyedMessage[Array[Byte], Array[Byte]]` to `def readMessage(): ProducerRecord[Array[Byte], Array[Byte]]`
- MessageFormatter's package was changed from `kafka.tools` to `kafka.common`
- MessageReader's package was changed from `kafka.tools` to `kafka.common`
- MirrorMakerMessageHandler no longer exposes the `handle(record: MessageAndMetadata[Array[Byte], Array[Byte]])` method as it was never called.
- The 0.7 KafkaMigrationTool is no longer packaged with Kafka. If you need to migrate from 0.7 to 0.10.0, please migrate to 0.8 first and then follow the documented upgrade process to upgrade from 0.8 to 0.10.0.
- The new consumer has standardized its APIs to accept `java.util.collection` as the sequence type for method parameters. Existing code may have to be updated to work with the 0.10.0 client library.
- LZ4-compressed message handling was changed to use an interoperable framing specification (LZ4f v1.5.1). To maintain compatibility with old clients, this change only applies to Message format 0.10.0 and later. Clients that Produce/Fetch LZ4-compressed messages using v0/v1 (Message format 0.9.0) should continue to use the 0.9.0 framing implementation. Clients that use Produce/Fetch protocols v2 or later should use interoperable LZ4f framing. A list of interoperable LZ4 libraries is available at <https://www.lz4.org/>

## [Notable changes in 0.10.0.0](#)

- Starting from Kafka 0.10.0.0, a new client library named **Kafka Streams** is available for stream processing on data stored in Kafka topics. This new client library only works with 0.10.x and upward versioned brokers due to message format changes mentioned above. For more information please read [Streams documentation](#).
- The default value of the configuration parameter `receive.buffer.bytes` is now 64K for the new consumer.
- The new consumer now exposes the configuration parameter `exclude.internal.topics` to restrict internal topics (such as the consumer offsets topic) from accidentally being included in regular expression subscriptions. By default, it is enabled.
- The old Scala producer has been deprecated. Users should migrate their code to the Java producer included in the kafka-clients JAR as soon as possible.
- The new consumer API has been marked stable.

## [Upgrading from 0.8.0, 0.8.1.X, or 0.8.2.X to 0.9.0.0](#)

0.9.0.0 has [potential breaking changes](#) (please review before upgrading) and an inter-broker protocol change from previous versions. This means that upgraded brokers and clients may not be compatible with older versions. It is important that you upgrade your Kafka cluster before upgrading your clients. If you are using MirrorMaker downstream clusters should be upgraded first as well.

### **For a rolling upgrade:**

1. Update server.properties file on all brokers and add the following property:  
`inter.broker.protocol.version=0.8.2.X`
2. Upgrade the brokers. This can be done a broker at a time by simply bringing it down, updating the code, and restarting it.
3. Once the entire cluster is upgraded, bump the protocol version by editing  
`inter.broker.protocol.version` and setting it to 0.9.0.0.
4. Restart the brokers one by one for the new protocol version to take effect

**Note:** If you are willing to accept downtime, you can simply take all the brokers down, update the code and start all of them. They will start with the new protocol by default.

**Note:** Bumping the protocol version and restarting can be done any time after the brokers were upgraded. It does not have to be immediately after.

## [Potential breaking changes in 0.9.0.0](#)

- Java 1.6 is no longer supported.
- Scala 2.9 is no longer supported.
- Broker IDs above 1000 are now reserved by default to automatically assigned broker IDs. If your cluster has existing broker IDs above that threshold make sure to increase the `reserved.broker.max.id` broker configuration property accordingly.
- Configuration parameter `replica.lag.max.messages` was removed. Partition leaders will no longer consider the number of lagging messages when deciding which replicas are in sync.
- Configuration parameter `replica.lag.time.max.ms` now refers not just to the time passed since last fetch request from replica, but also to time since the replica last caught up. Replicas that are still fetching messages from leaders but did not catch up to the latest messages in `replica.lag.time.max.ms` will be considered out of sync.
- Compacted topics no longer accept messages without key and an exception is thrown by the producer if this is attempted. In 0.8.x, a message without key would cause the log compaction thread to subsequently complain and quit (and stop compacting all compacted topics).
- MirrorMaker no longer supports multiple target clusters. As a result it will only accept a single -- `consumer.config` parameter. To mirror multiple source clusters, you will need at least one MirrorMaker instance per source cluster, each with its own consumer configuration.
- Tools packaged under `*org.apache.kafka.clients.tools.**` have been moved to `*org.apache.kafka.tools.**`. All included scripts will still function as usual, only custom code directly

importing these classes will be affected.

- The default Kafka JVM performance options (KAFKA\_JVM\_PERFORMANCE\_OPTS) have been changed in kafka-run-class.sh.
- The kafka-topics.sh script (kafka.admin.TopicCommand) now exits with non-zero exit code on failure.
- The kafka-topics.sh script (kafka.admin.TopicCommand) will now print a warning when topic names risk metric collisions due to the use of a '.' or '\_' in the topic name, and error in the case of an actual collision.
- The kafka-console-producer.sh script (kafka.tools.ConsoleProducer) will use the Java producer instead of the old Scala producer by default, and users have to specify 'old-producer' to use the old producer.
- By default, all command line tools will print all logging messages to stderr instead of stdout.

### [Notable changes in 0.9.0.1](#)

- The new broker id generation feature can be disabled by setting broker.id.generation.enable to false.
- Configuration parameter log.cleaner.enable is now true by default. This means topics with a cleanup.policy=compact will now be compacted by default, and 128 MB of heap will be allocated to the cleaner process via log.cleaner.dedupe.buffer.size. You may want to review log.cleaner.dedupe.buffer.size and the other log.cleaner configuration values based on your usage of compacted topics.
- Default value of configuration parameter fetch.min.bytes for the new consumer is now 1 by default.

### **Deprecations in 0.9.0.0**

- Altering topic configuration from the kafka-topics.sh script (kafka.admin.TopicCommand) has been deprecated. Going forward, please use the kafka-configs.sh script (kafka.admin.ConfigCommand) for this functionality.
- The kafka-consumer-offset-checker.sh (kafka.tools.ConsumerOffsetChecker) has been deprecated. Going forward, please use kafka-consumer-groups.sh (kafka.admin.ConsumerGroupCommand) for this functionality.
- The kafka.tools.ProducerPerformance class has been deprecated. Going forward, please use org.apache.kafka.tools.ProducerPerformance for this functionality (kafka-producer-perf-test.sh will also be changed to use the new class).
- The producer config block.on.buffer.full has been deprecated and will be removed in future release. Currently its default value has been changed to false. The KafkaProducer will no longer throw BufferExhaustedException but instead will use max.block.ms value to block, after which it will throw a TimeoutException. If block.on.buffer.full property is set to true explicitly, it will set the max.block.ms to Long.MAX\_VALUE and metadata.fetch.timeout.ms will not be honoured

### [Upgrading from 0.8.1 to 0.8.2](#)

0.8.2 is fully compatible with 0.8.1. The upgrade can be done one broker at a time by simply bringing it down, updating the code, and restarting it.

### [Upgrading from 0.8.0 to 0.8.1](#)

0.8.1 is fully compatible with 0.8. The upgrade can be done one broker at a time by simply bringing it down, updating the code, and restarting it.

### [Upgrading from 0.7](#)

Release 0.7 is incompatible with newer releases. Major changes were made to the API, ZooKeeper data structures, and protocol, and configuration in order to add replication (Which was missing in 0.7). The upgrade from 0.7 to later versions requires a [special tool](#) for migration. This migration can be done without downtime.

## 2. APIs

Kafka includes five core apis:

1. The [Producer](#) API allows applications to send streams of data to topics in the Kafka cluster.
2. The [Consumer](#) API allows applications to read streams of data from topics in the Kafka cluster.
3. The [Streams](#) API allows transforming streams of data from input topics to output topics.
4. The [Connect](#) API allows implementing connectors that continually pull from some source system or application into Kafka or push from Kafka into some sink system or application.
5. The [Admin](#) API allows managing and inspecting topics, brokers, and other Kafka objects.

Kafka exposes all its functionality over a language independent protocol which has clients available in many programming languages. However only the Java clients are maintained as part of the main Kafka project, the others are available as independent open source projects. A list of non-Java clients is available [here](#).

### 2.1 Producer API

The Producer API allows applications to send streams of data to topics in the Kafka cluster.

Examples showing how to use the producer are given in the [javadocs](#).

To use the producer, you can use the following maven dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.4.0</version>
</dependency>
```

### 2.2 Consumer API

The Consumer API allows applications to read streams of data from topics in the Kafka cluster.

Examples showing how to use the consumer are given in the [javadocs](#).

To use the consumer, you can use the following maven dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.4.0</version>
</dependency>
```

### 2.3 Streams API

The [Streams](#) API allows transforming streams of data from input topics to output topics.

Examples showing how to use this library are given in the [javadocs](#)

Additional documentation on using the Streams API is available [here](#).

To use Kafka Streams you can use the following maven dependency:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>3.4.0</version>
</dependency>
```

When using Scala you may optionally include the `kafka-streams-scala` library. Additional documentation on using the Kafka Streams DSL for Scala is available [in the developer guide](#).

To use Kafka Streams DSL for Scala 2.13 you can use the following maven dependency:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams-scala_2.13</artifactId>
  <version>3.4.0</version>
</dependency>
```

## 2.4 Connect API

The Connect API allows implementing connectors that continually pull from some source data system into Kafka or push from Kafka into some sink data system.

Many users of Connect won't need to use this API directly, though, they can use pre-built connectors without needing to write any code. Additional information on using Connect is available [here](#).

Those who want to implement custom connectors can see the [javadoc](#).

## 2.5 Admin API

The Admin API supports managing and inspecting topics, brokers, acls, and other Kafka objects.

To use the Admin API, add the following Maven dependency:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>3.4.0</version>
</dependency>
```

For more information about the Admin APIs, see the [javadoc](#).

# 3. CONFIGURATION

Kafka uses key-value pairs in the [property file format](#) for configuration. These values can be supplied either from a file or programmatically.

## 3.1 Broker Configs

The essential configurations are the following:

- `broker.id`
- `log.dirs`
- `zookeeper.connect`

Topic-level configurations and defaults are discussed in more detail [below](#).

### • [\*\*advertised.listeners\*\*](#)

Listeners to publish to ZooKeeper for clients to use, if different than the `listeners` config property. In IaaS environments, this may need to be different from the interface to which the broker binds. If this is not set, the value for `listeners` will be used. Unlike `listeners`, it is not valid to advertise the 0.0.0.0 meta-address.

Also unlike `listeners`, there can be duplicated ports in this property, so that one listener can be configured to advertise another listener's address. This can be useful in some cases where external load balancers are used.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	per-broker

- **[auto.create.topics.enable](#)**

Enable auto creation of topic on the server

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	high
Update Mode:	read-only

- **[auto.leader.rebalance.enable](#)**

Enables auto leader balancing. A background thread checks the distribution of partition leaders at regular intervals, configurable by `leader.imbalance.check.interval.seconds`. If the leader imbalance exceeds `leader.imbalance.per.broker.percentage`, leader rebalance to the preferred leader for partitions is triggered.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	high
Update Mode:	read-only

- **[background.threads](#)**

The number of threads to use for various background processing tasks

Type:	<b>int</b>
Default:	10
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- [\*\*broker.id\*\*](#)

The broker id for this server. If unset, a unique broker id will be generated. To avoid conflicts between zookeeper generated broker id's and user configured broker id's, generated broker ids start from reserved.broker.max.id + 1.

	<b>Type:</b>	int
	Default:	-1
	Valid Values:	
	Importance:	high
	Update Mode:	read-only

- [\*\*compression.type\*\*](#)

Specify the final compression type for a given topic. This configuration accepts the standard compression codecs ('gzip', 'snappy', 'lz4', 'zstd'). It additionally accepts 'uncompressed' which is equivalent to no compression; and 'producer' which means retain the original compression codec set by the producer.

	<b>Type:</b>	string
	Default:	producer
	Valid Values:	[uncompressed, zstd, lz4, snappy, gzip, producer]
	Importance:	high
	Update Mode:	cluster-wide

- [\*\*control.plane.listener.name\*\*](#)

Name of listener used for communication between controller and brokers. Broker will use the control.plane.listener.name to locate the endpoint in listeners list, to listen for connections from the controller. For example, if a broker's config is :

```
listeners = INTERNAL://192.1.1.8:9092, EXTERNAL://10.1.1.5:9093, CONTROLLER://192.1.1.8:9094
listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL
control.plane.listener.name = CONTROLLER
```

On startup, the broker will start listening on "192.1.1.8:9094" with security protocol "SSL".

On controller side, when it discovers a broker's published endpoints through zookeeper, it will use the control.plane.listener.name to find the endpoint, which it will use to establish connection to the broker.

For example, if the broker's published endpoints on zookeeper are :

"endpoints" :

```
["INTERNAL://broker1.example.com:9092", "EXTERNAL://broker1.example.com:9093", "CONTROLLER://broker1.example.com:9094"]
```

and the controller's config is :

```
listener.security.protocol.map = INTERNAL:PLAINTEXT, EXTERNAL:SSL, CONTROLLER:SSL
control.plane.listener.name = CONTROLLER
```

then controller will use "broker1.example.com:9094" with security protocol "SSL" to connect to the broker.

If not explicitly configured, the default value will be null and there will be no dedicated endpoints for controller connections.

If explicitly configured, the value cannot be the same as the value of `inter.broker.listener.name`.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*controller.listener.names\*\*](#)

A comma-separated list of the names of the listeners used by the controller. This is required if running in KRaft mode. When communicating with the controller quorum, the broker will always use the first listener in this list.

Note: The ZK-based controller should not set this configuration.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*controller.quorum.election.backoff.max.ms\*\*](#)

Maximum time in milliseconds before starting new elections. This is used in the binary exponential backoff mechanism that helps prevent gridlocked elections

Type:	<b>int</b>
Default:	1000 (1 second)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*controller.quorum.election.timeout.ms\*\*](#)

Maximum time in milliseconds to wait without being able to fetch from the leader before triggering a new election

Type:	<b>int</b>
Default:	1000 (1 second)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [controller.quorum.fetch.timeout.ms](#)

Maximum time without a successful fetch from the current leader before becoming a candidate and triggering an election for voters; Maximum time without receiving fetch from a majority of the quorum before asking around to see if there's a new epoch for leader

<b>Type:</b>	<b>int</b>
Default:	2000 (2 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [controller.quorum.voters](#)

Map of id/endpoint information for the set of voters in a comma-separated list of `{id}@{host}:{port}` entries. For example: `1@localhost:9092,2@localhost:9093,3@localhost:9094`

<b>Type:</b>	<b>list</b>
Default:	""
Valid Values:	non-empty list
Importance:	high
Update Mode:	read-only

- [delete.topic.enable](#)

Enables delete topic. Delete topic through the admin tool will have no effect if this config is turned off

<b>Type:</b>	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	high
Update Mode:	read-only

- [early.start.listeners](#)

A comma-separated list of listener names which may be started before the authorizer has finished initialization. This is useful when the authorizer is dependent on the cluster itself for bootstrapping, as is the case for the StandardAuthorizer (which stores ACLs in the metadata log.) By default, all listeners included in controller.listener.names will also be early start listeners. A listener should not appear in this list if it accepts external traffic.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- **[leader.imbalance.check.interval.seconds](#)**

The frequency with which the partition rebalance check is triggered by the controller

Type:	<b>long</b>
Default:	300
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- **[leader.imbalance.per.broker.percentage](#)**

The ratio of leader imbalance allowed per broker. The controller would trigger a leader balance if it goes above this value per broker. The value is specified in percentage.

Type:	<b>int</b>
Default:	10
Valid Values:	
Importance:	high
Update Mode:	read-only

- **[listeners](#)**

Listener List - Comma-separated list of URIs we will listen on and the listener names. If the listener name is not a security protocol, `listener.security.protocol.map` must also be set.

Listener names and port numbers must be unique.

Specify hostname as 0.0.0.0 to bind to all interfaces.

Leave hostname empty to bind to default interface.

Examples of legal listener lists:

PLAINTEXT://myhost:9092,SSL://:9091

CLIENT://0.0.0.0:9092,REPLICATION://localhost:9093

Type:	<b>string</b>
Default:	PLAINTEXT://:9092
Valid Values:	
Importance:	high
Update Mode:	per-broker

- **log.dir**

The directory in which the log data is kept (supplemental for log.dirs property)

Type:	<b>string</b>
Default:	/tmp/kafka-logs
Valid Values:	
Importance:	high
Update Mode:	read-only

- **log.dirs**

A comma-separated list of the directories where the log data is stored. If not set, the value in log.dir is used.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- **log.flush.interval.messages**

The number of messages accumulated on a log partition before messages are flushed to disk

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- **log.flush.interval.ms**

The maximum time in ms that a message in any topic is kept in memory before flushed to disk. If not set, the value in log.flush.scheduler.interval.ms is used

Type:	<b>long</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [log.flush.offset.checkpoint.interval.ms](#)

The frequency with which we update the persistent record of the last flush which acts as the log recovery point

<b>Type:</b>	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	high
Update Mode:	read-only

- [log.flush.scheduler.interval.ms](#)

The frequency in ms that the log flusher checks whether any log needs to be flushed to disk

<b>Type:</b>	<b>long</b>
Default:	9223372036854775807
Valid Values:	
Importance:	high
Update Mode:	read-only

- [log.flush.start.offset.checkpoint.interval.ms](#)

The frequency with which we update the persistent record of log start offset

<b>Type:</b>	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	high
Update Mode:	read-only

- [log.retention.bytes](#)

The maximum size of the log before deleting it

<b>Type:</b>	<b>long</b>
Default:	-1
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [log.retention.hours](#)

The number of hours to keep a log file before deleting it (in hours), tertiary to log.retention.ms property

Type:	<b>int</b>
Default:	168
Valid Values:	
Importance:	high
Update Mode:	read-only

- [log.retention.minutes](#)

The number of minutes to keep a log file before deleting it (in minutes), secondary to log.retention.ms property. If not set, the value in log.retention.hours is used

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [log.retention.ms](#)

The number of milliseconds to keep a log file before deleting it (in milliseconds), If not set, the value in log.retention.minutes is used. If set to -1, no time limit is applied.

Type:	<b>long</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [log.roll.hours](#)

The maximum time before a new log segment is rolled out (in hours), secondary to log.roll.ms property

Type:	<b>int</b>
Default:	168
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [log.roll.jitter.hours](#)

The maximum jitter to subtract from logRollTimeMillis (in hours), secondary to log.roll.jitter.ms property

Type:	<b>int</b>
Default:	0
Valid Values:	[0,...]
Importance:	high
Update Mode:	read-only

- [log.roll.jitter.ms](#)

The maximum jitter to subtract from logRollTimeMillis (in milliseconds). If not set, the value in log.roll.jitter.hours is used

Type:	<b>long</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [log.roll.ms](#)

The maximum time before a new log segment is rolled out (in milliseconds). If not set, the value in log.roll.hours is used

Type:	<b>long</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [log.segment.bytes](#)

The maximum size of a single log file

Type:	<b>int</b>
Default:	1073741824 (1 gibibyte)
Valid Values:	[14,...]
Importance:	high
Update Mode:	cluster-wide

- [\*\*log.segment.delete.delay.ms\*\*](#)

The amount of time to wait before deleting a file from the filesystem

Type:	<b>long</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	high
Update Mode:	cluster-wide

- [\*\*message.max.bytes\*\*](#)

The largest record batch size allowed by Kafka (after compression if compression is enabled). If this is increased and there are consumers older than 0.10.2, the consumers' fetch size must also be increased so that they can fetch record batches this large. In the latest message format version, records are always grouped into batches for efficiency. In previous message format versions, uncompressed records are not grouped into batches and this limit only applies to a single record in that case. This can be set per topic with the topic level `max.message.bytes` config.

Type:	<b>int</b>
Default:	1048588
Valid Values:	[0,...]
Importance:	high
Update Mode:	cluster-wide

- [\*\*metadata.log.dir\*\*](#)

This configuration determines where we put the metadata log for clusters in KRaft mode. If it is not set, the metadata log is placed in the first log directory from log.dirs.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*metadata.log.max.record.bytes.between.snapshots\*\*](#)

This is the maximum number of bytes in the log between the latest snapshot and the high-watermark needed before generating a new snapshot. The default value is 20971520. To generate snapshots based on the time elapsed, see the `metadata.log.max.snapshot.interval.ms` configuration. The Kafka node will generate a snapshot when either the maximum time interval is reached or the maximum bytes limit is reached.

Type:	<b>long</b>
Default:	20971520
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*metadata.log.max.snapshot.interval.ms\*\*](#)

This is the maximum number of milliseconds to wait to generate a snapshot if there are committed records in the log that are not included in the latest snapshot. A value of zero disables time based snapshot generation. The default value is 3600000. To generate snapshots based on the number of metadata bytes, see the `metadata.log.max.record.bytes.between.snapshots` configuration. The Kafka node will generate a snapshot when either the maximum time interval is reached or the maximum bytes limit is reached.

Type:	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	[0,...]
Importance:	high
Update Mode:	read-only

- [\*\*metadata.log.segment.bytes\*\*](#)

The maximum size of a single metadata log file.

Type:	<b>int</b>
Default:	1073741824 (1 gibibyte)
Valid Values:	[12,...]
Importance:	high
Update Mode:	read-only

- [\*\*metadata.log.segment.ms\*\*](#)

The maximum time before a new metadata log file is rolled out (in milliseconds).

Type:	<b>long</b>
Default:	604800000 (7 days)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*metadata.max.retention.bytes\*\*](#)

The maximum combined size of the metadata log and snapshots before deleting old snapshots and log files. Since at least one snapshot must exist before any logs can be deleted, this is a soft limit.

Type:	<b>long</b>
Default:	104857600 (100 mebibytes)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*metadata.max.retention.ms\*\*](#)

The number of milliseconds to keep a metadata log file or snapshot before deleting it. Since at least one snapshot must exist before any logs can be deleted, this is a soft limit.

Type:	<b>long</b>
Default:	604800000 (7 days)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*min.insync.replicas\*\*](#)

When a producer sets acks to "all" (or "-1"), min.insync.replicas specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).

When used together, min.insync.replicas and acks allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of 3, set min.insync.replicas to 2, and produce with acks of "all". This will ensure that the producer raises an exception if a majority of replicas do not receive a write.

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- [\*\*node.id\*\*](#)

The node ID associated with the roles this process is playing when `process.roles` is non-empty. This is required configuration when running in KRaft mode.

Type:	<b>int</b>
Default:	-1
Valid Values:	
Importance:	high
Update Mode:	read-only

- **[num.io.threads](#)**

The number of threads that the server uses for processing requests, which may include disk I/O

Type:	<b>int</b>
Default:	8
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- **[num.network.threads](#)**

The number of threads that the server uses for receiving requests from the network and sending responses to the network

Type:	<b>int</b>
Default:	3
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- **[num.recovery.threads.per.data.dir](#)**

The number of threads per data directory to be used for log recovery at startup and flushing at shutdown

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	high
Update Mode:	cluster-wide

- **[num.replica.alter.log.dirs.threads](#)**

The number of threads that can move replicas between log directories, which may include disk I/O

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [num.replica.fetchers](#)

Number of fetcher threads used to replicate records from each source broker. The total number of fetchers on each broker is bound by `num.replica.fetchers` multiplied by the number of brokers in the cluster. Increasing this value can increase the degree of I/O parallelism in the follower and leader broker at the cost of higher CPU and memory utilization.

Type:	<b>int</b>
Default:	1
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [offset.metadata.max.bytes](#)

The maximum size for a metadata entry associated with an offset commit

Type:	<b>int</b>
Default:	4096 (4 kibibytes)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [offsets.commit.required.acks](#)

The required acks before the commit can be accepted. In general, the default (-1) should not be overridden

Type:	<b>short</b>
Default:	-1
Valid Values:	
Importance:	high
Update Mode:	read-only

- [offsets.commit.timeout.ms](#)

Offset commit will be delayed until all replicas for the offsets topic receive the commit or this timeout is reached. This is similar to the producer request timeout.

<b>Type:</b>	<b>int</b>
Default:	5000 (5 seconds)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*offsets.load.buffer.size\*\*](#)

Batch size for reading from the offsets segments when loading offsets into the cache (soft-limit, overridden if records are too large).

<b>Type:</b>	<b>int</b>
Default:	5242880
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*offsets.retention.check.interval.ms\*\*](#)

Frequency at which to check for stale offsets

<b>Type:</b>	<b>long</b>
Default:	600000 (10 minutes)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*offsets.retention.minutes\*\*](#)

For subscribed consumers, committed offset of a specific partition will be expired and discarded when 1) this retention period has elapsed after the consumer group loses all its consumers (i.e. becomes empty); 2) this retention period has elapsed since the last time an offset is committed for the partition and the group is no longer subscribed to the corresponding topic. For standalone consumers (using manual assignment), offsets will be expired after this retention period has elapsed since the time of last commit. Note that when a group is deleted via the delete-group request, its committed offsets will also be deleted without extra retention period; also when a topic is deleted via the delete-topic request, upon propagated metadata update any group's committed offsets for that topic will also be deleted without extra retention period.

<b>Type:</b>	<b>int</b>
Default:	10080
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [offsets.topic.compression.codec](#)

Compression codec for the offsets topic - compression may be used to achieve "atomic" commits

Type:	<b>int</b>
Default:	0
Valid Values:	
Importance:	high
Update Mode:	read-only

- [offsets.topic.num.partitions](#)

The number of partitions for the offset commit topic (should not change after deployment)

Type:	<b>int</b>
Default:	50
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [offsets.topic.replication.factor](#)

The replication factor for the offsets topic (set higher to ensure availability). Internal topic creation will fail until the cluster size meets this replication factor requirement.

Type:	<b>short</b>
Default:	3
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [offsets.topic.segment.bytes](#)

The offsets topic segment bytes should be kept relatively small in order to facilitate faster log compaction and cache loads

Type:	<b>int</b>
Default:	104857600 (100 mebibytes)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [process.roles](#)

The roles that this process plays: 'broker', 'controller', or 'broker,controller' if it is both. This configuration is only applicable for clusters in KRaft (Kafka Raft) mode (instead of ZooKeeper). Leave this config undefined or empty for Zookeeper clusters.

<b>Type:</b>	list
Default:	""
Valid Values:	[broker, controller]
Importance:	high
Update Mode:	read-only

- [queued.max.requests](#)

The number of queued requests allowed for data-plane, before blocking the network threads

<b>Type:</b>	int
Default:	500
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [replica.fetch.min.bytes](#)

Minimum bytes expected for each fetch response. If not enough bytes, wait up to `replica.fetch.wait.max.ms` (broker config).

<b>Type:</b>	int
Default:	1
Valid Values:	
Importance:	high
Update Mode:	read-only

- [replica.fetch.wait.max.ms](#)

The maximum wait time for each fetcher request issued by follower replicas. This value should always be less than the `replica.lag.time.max.ms` at all times to prevent frequent shrinking of ISR for low throughput topics

<b>Type:</b>	int
Default:	500
Valid Values:	
Importance:	high
Update Mode:	read-only

- [replica.high.watermark.checkpoint.interval.ms](#)

The frequency with which the high watermark is saved out to disk

Type:	<b>long</b>
Default:	5000 (5 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [replica.lag.time.max.ms](#)

If a follower hasn't sent any fetch requests or hasn't consumed up to the leaders log end offset for at least this time, the leader will remove the follower from isr

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [replica.socket.receive.buffer.bytes](#)

The socket receive buffer for network requests

Type:	<b>int</b>
Default:	65536 (64 kibibytes)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [replica.socket.timeout.ms](#)

The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms

Type:	<b>int</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [request.timeout.ms](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

<b>Type:</b>	<b>int</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [sasl.mechanism.controller.protocol](#)

SASL mechanism used for communication with controllers. Default is GSSAPI.

<b>Type:</b>	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	high
Update Mode:	read-only

- [socket.receive.buffer.bytes](#)

The SO\_RCVBUF buffer of the socket server sockets. If the value is -1, the OS default will be used.

<b>Type:</b>	<b>int</b>
Default:	102400 (100 kibibytes)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [socket.request.max.bytes](#)

The maximum number of bytes in a socket request

<b>Type:</b>	<b>int</b>
Default:	104857600 (100 mebibytes)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*socket.send.buffer.bytes\*\*](#)

The SO\_SNDBUF buffer of the socket server sockets. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	102400 (100 kibibytes)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*transaction.max.timeout.ms\*\*](#)

The maximum allowed timeout for transactions. If a client's requested transaction time exceed this, then the broker will return an error in InitProducerIdRequest. This prevents a client from too large of a timeout, which can stall consumers reading from topics included in the transaction.

Type:	<b>int</b>
Default:	900000 (15 minutes)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transaction.state.log.load.buffer.size\*\*](#)

Batch size for reading from the transaction log segments when loading producer ids and transactions into the cache (soft-limit, overridden if records are too large).

Type:	<b>int</b>
Default:	5242880
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transaction.state.log.min\\_isr\*\*](#)

Overridden min.insync.replicas config for the transaction topic.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transaction.state.log.num.partitions\*\*](#)

The number of partitions for the transaction topic (should not change after deployment).

Type:	<b>int</b>
Default:	50
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transaction.state.log.replication.factor\*\*](#)

The replication factor for the transaction topic (set higher to ensure availability). Internal topic creation will fail until the cluster size meets this replication factor requirement.

Type:	<b>short</b>
Default:	3
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transaction.state.log.segment.bytes\*\*](#)

The transaction topic segment bytes should be kept relatively small in order to facilitate faster log compaction and cache loads

Type:	<b>int</b>
Default:	104857600 (100 mebibytes)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*transactional.id.expiration.ms\*\*](#)

The time in ms that the transaction coordinator will wait without receiving any transaction status updates for the current transaction before expiring its transactional id. Transactional IDs will not expire while a the transaction is still ongoing.

Type:	<b>int</b>
Default:	604800000 (7 days)
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [unclean.leader.election.enable](#)

Indicates whether to enable replicas not in the ISR set to be elected as leader as a last resort, even though doing so may result in data loss

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	high
Update Mode:	cluster-wide

- [zookeeper.connect](#)

Specifies the ZooKeeper connection string in the form `hostname:port` where host and port are the host and port of a ZooKeeper server. To allow connecting through other ZooKeeper nodes when that ZooKeeper machine is down you can also specify multiple hosts in the form `hostname1:port1,hostname2:port2,hostname3:port3`.

The server can also have a ZooKeeper chroot path as part of its ZooKeeper connection string which puts its data under some path in the global ZooKeeper namespace. For example to give a chroot path of `/chroot/path` you would give the connection string as

`hostname1:port1,hostname2:port2,hostname3:port3/chroot/path`.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [zookeeper.connection.timeout.ms](#)

The max time that the client waits to establish a connection to zookeeper. If not set, the value in `zookeeper.session.timeout.ms` is used

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	high
Update Mode:	read-only

- [zookeeper.max.in.flight.requests](#)

The maximum number of unacknowledged requests the client will send to Zookeeper before blocking.

Type:	<b>int</b>
Default:	10
Valid Values:	[1,...]
Importance:	high
Update Mode:	read-only

- [\*\*zookeeper.metadata.migration.enable\*\*](#)

Enable ZK to KRaft migration

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*zookeeper.session.timeout.ms\*\*](#)

Zookeeper session timeout

Type:	<b>int</b>
Default:	18000 (18 seconds)
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*zookeeper.set.acl\*\*](#)

Set client to use secure ACLs

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	high
Update Mode:	read-only

- [\*\*broker.heartbeat.interval.ms\*\*](#)

The length of time in milliseconds between broker heartbeats. Used when running in KRaft mode.

<b>Type:</b>	<b>int</b>
Default:	2000 (2 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **[broker.id.generation.enable](#)**

Enable automatic broker id generation on the server. When enabled the value configured for reserved.broker.max.id should be reviewed.

<b>Type:</b>	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **[broker.rack](#)**

Rack of the broker. This will be used in rack aware replication assignment for fault tolerance.

Examples: `RACK1`, `us-east-1d`

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **[broker.session.timeout.ms](#)**

The length of time in milliseconds that a broker lease lasts if no heartbeats are made. Used when running in KRaft mode.

<b>Type:</b>	<b>int</b>
Default:	9000 (9 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **[connections.max.idle.ms](#)**

Idle connections timeout: the server socket processor threads close the connections that idle more than this

Type:	<b>long</b>
Default:	600000 (10 minutes)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*connections.max.reauth.ms\*\*](#)

When explicitly set to a positive number (the default is 0, not a positive number), a session lifetime that will not exceed the configured value will be communicated to v2.2.0 or later clients when they authenticate. The broker will disconnect any such connection that is not re-authenticated within the session lifetime and that is then subsequently used for any purpose other than re-authentication. Configuration names can optionally be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.oauthbearer.connections.max.reauth.ms=3600000

Type:	<b>long</b>
Default:	0
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*controlled.shutdown.enable\*\*](#)

Enable controlled shutdown of the server

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*controlled.shutdown.max.retries\*\*](#)

Controlled shutdown can fail for multiple reasons. This determines the number of retries when such failure happens

Type:	<b>int</b>
Default:	3
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [controlled.shutdown.retry.backoff.ms](#)

Before each retry, the system needs time to recover from the state that caused the previous failure (Controller fail over, replica lag etc). This config determines the amount of time to wait before retrying.

Type:	<b>long</b>
Default:	5000 (5 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [controller.quorum.append.linger.ms](#)

The duration in milliseconds that the leader will wait for writes to accumulate before flushing them to disk.

Type:	<b>int</b>
Default:	25
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [controller.quorum.request.timeout.ms](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

Type:	<b>int</b>
Default:	2000 (2 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [controller.socket.timeout.ms](#)

The socket timeout for controller-to-broker channels

Type:	<b>int</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*default.replication.factor\*\*](#)

The default replication factors for automatically created topics

Type:	<b>int</b>
Default:	1
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*delegation.token.expiry.time.ms\*\*](#)

The token validity time in miliseconds before the token needs to be renewed. Default value 1 day.

Type:	<b>long</b>
Default:	86400000 (1 day)
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*delegation.token.master.key\*\*](#)

DEPRECATED: An alias for delegation.token.secret.key, which should be used instead of this config.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*delegation.token.max.lifetime.ms\*\*](#)

The token has a maximum lifetime beyond which it cannot be renewed anymore. Default value 7 days.

Type:	<b>long</b>
Default:	604800000 (7 days)
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*delegation.token.secret.key\*\*](#)

Secret key to generate and verify delegation tokens. The same key must be configured across all the brokers. If the key is not set or set to empty string, brokers will disable the delegation token support.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*delete.records.purgatory.purge.interval.requests\*\*](#)

The purge interval (in number of requests) of the delete records request purgatory

Type:	<b>int</b>
Default:	1
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*fetch.max.bytes\*\*](#)

The maximum number of bytes we will return for a fetch request. Must be at least 1024.

Type:	<b>int</b>
Default:	57671680 (55 mebibytes)
Valid Values:	[1024,...]
Importance:	medium
Update Mode:	read-only

- [\*\*fetch.purgatory.purge.interval.requests\*\*](#)

The purge interval (in number of requests) of the fetch request purgatory

Type:	<b>int</b>
Default:	1000
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*group.initial.rebalance.delay.ms\*\*](#)

The amount of time the group coordinator will wait for more consumers to join a new group before performing the first rebalance. A longer delay means potentially fewer rebalances, but increases the time until processing begins.

Type:	<b>int</b>
Default:	3000 (3 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*group.max.session.timeout.ms\*\*](#)

The maximum allowed session timeout for registered consumers. Longer timeouts give consumers more time to process messages in between heartbeats at the cost of a longer time to detect failures.

Type:	<b>int</b>
Default:	1800000 (30 minutes)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*group.max.size\*\*](#)

The maximum number of consumers that a single consumer group can accommodate.

Type:	<b>int</b>
Default:	2147483647
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*group.min.session.timeout.ms\*\*](#)

The minimum allowed session timeout for registered consumers. Shorter timeouts result in quicker failure detection at the cost of more frequent consumer heartbeating, which can overwhelm broker resources.

Type:	<b>int</b>
Default:	6000 (6 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [initial.broker.registration.timeout.ms](#)

When initially registering with the controller quorum, the number of milliseconds to wait before declaring failure and exiting the broker process.

<b>Type:</b>	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [inter.broker.listener.name](#)

Name of listener used for communication between brokers. If this is unset, the listener name is defined by security.inter.broker.protocol. It is an error to set this and security.inter.broker.protocol properties at the same time.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [inter.broker.protocol.version](#)

Specify which version of the inter-broker protocol will be used.

This is typically bumped after all brokers were upgraded to a new version.

Example of some valid values are: 0.8.0, 0.8.1, 0.8.1.1, 0.8.2, 0.8.2.0, 0.8.2.1, 0.9.0.0, 0.9.0.1 Check MetadataVersion for the full list.

<b>Type:</b>	<b>string</b>
Default:	3.4-IV0
Valid Values:	[0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV0, 2.1-IV1, 2.1-IV2, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.3-IV1, 2.4-IV0, 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1, 3.0-IV0, 3.0-IV1, 3.1-IV0, 3.2-IV0, 3.3-IV0, 3.3-IV1, 3.3-IV2, 3.3-IV3, 3.4-IV0]
Importance:	medium
Update Mode:	read-only

- [log.cleaner.backoff.ms](#)

The amount of time to sleep when there are no logs to clean

<b>Type:</b>	<b>long</b>
Default:	15000 (15 seconds)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- **[log.cleaner.dedupe.buffer.size](#)**

The total memory used for log deduplication across all cleaner threads

<b>Type:</b>	<b>long</b>
Default:	134217728
Valid Values:	
Importance:	medium
Update Mode:	cluster-wide

- **[log.cleaner.delete.retention.ms](#)**

The amount of time to retain delete tombstone markers for log compacted topics. This setting also gives a bound on the time in which a consumer must complete a read if they begin from offset 0 to ensure that they get a valid snapshot of the final stage (otherwise delete tombstones may be collected before they complete their scan).

<b>Type:</b>	<b>long</b>
Default:	86400000 (1 day)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- **[log.cleaner.enable](#)**

Enable the log cleaner process to run on the server. Should be enabled if using any topics with a cleanup.policy=compact including the internal offsets topic. If disabled those topics will not be compacted and continually grow in size.

<b>Type:</b>	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [log.cleaner.io.buffer.load.factor](#)

Log cleaner dedupe buffer load factor. The percentage full the dedupe buffer can become. A higher value will allow more log to be cleaned at once but will lead to more hash collisions

Type:	<b>double</b>
Default:	0.9
Valid Values:	
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.io.buffer.size](#)

The total memory used for log cleaner I/O buffers across all cleaner threads

Type:	<b>int</b>
Default:	524288
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.io.max.bytes.per.second](#)

The log cleaner will be throttled so that the sum of its read and write i/o will be less than this value on average

Type:	<b>double</b>
Default:	1.7976931348623157E308
Valid Values:	
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.max.compaction.lag.ms](#)

The maximum time a message will remain ineligible for compaction in the log. Only applicable for logs that are being compacted.

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[1,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.min.cleanable.ratio](#)

The minimum ratio of dirty log to total log for a log to be eligible for cleaning. If the log.cleaner.max.compaction.lag.ms or the log.cleaner.min.compaction.lag.ms configurations are also specified, then the log compactor considers the log eligible for compaction as soon as either: (i) the dirty ratio threshold has been met and the log has had dirty (uncompacted) records for at least the log.cleaner.min.compaction.lag.ms duration, or (ii) if the log has had dirty (uncompacted) records for at most the log.cleaner.max.compaction.lag.ms period.

<b>Type:</b>	<b>double</b>
Default:	0.5
Valid Values:	[0,...,1]
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.min.compaction.lag.ms](#)

The minimum time a message will remain uncompacted in the log. Only applicable for logs that are being compacted.

<b>Type:</b>	<b>long</b>
Default:	0
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.cleaner.threads](#)

The number of background threads to use for log cleaning

<b>Type:</b>	<b>int</b>
Default:	1
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.cleanup.policy](#)

The default cleanup policy for segments beyond the retention window. A comma separated list of valid policies. Valid policies are: "delete" and "compact"

<b>Type:</b>	<b>list</b>
Default:	delete
Valid Values:	[compact, delete]
Importance:	medium
Update Mode:	cluster-wide

- [log.index.interval.bytes](#)

The interval with which we add an entry to the offset index

Type:	<b>int</b>
Default:	4096 (4 kibibytes)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.index.size.max.bytes](#)

The maximum size in bytes of the offset index

Type:	<b>int</b>
Default:	10485760 (10 mebibytes)
Valid Values:	[4,...]
Importance:	medium
Update Mode:	cluster-wide

- [log.message.format.version](#)

Specify the message format version the broker will use to append messages to the logs. The value should be a valid MetadataVersion. Some examples are: 0.8.2, 0.9.0.0, 0.10.0, check MetadataVersion for more details. By setting a particular message format version, the user is certifying that all the existing messages on disk are smaller or equal than the specified version. Setting this value incorrectly will cause consumers with older versions to break as they will receive messages with a format that they don't understand.

Type:	<b>string</b>
Default:	3.0-IV1
Valid Values:	[0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV0, 2.1-IV1, 2.1-IV2, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.3-IV1, 2.4-IV0, 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1, 3.0-IV0, 3.0-IV1, 3.1-IV0, 3.2-IV0, 3.3-IV0, 3.3-IV1, 3.3-IV2, 3.3-IV3, 3.4-IV0]
Importance:	medium
Update Mode:	read-only

- [log.message.timestamp.difference.max.ms](#)

The maximum difference allowed between the timestamp when a broker receives a message and the timestamp specified in the message. If log.message.timestamp.type=CreateTime, a message will be rejected if the difference in timestamp exceeds this threshold. This configuration is ignored if log.message.timestamp.type=LogAppendTime. The maximum timestamp difference allowed should be no greater than log.retention.ms to avoid unnecessarily frequent log rolling.

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [\*\*log.message.timestamp.type\*\*](#)

Define whether the timestamp in the message is message create time or log append time. The value should be either `createTime` or `LogAppendTime`

Type:	<b>string</b>
Default:	CreateTime
Valid Values:	[CreateTime, LogAppendTime]
Importance:	medium
Update Mode:	cluster-wide

- [\*\*log.preallocate\*\*](#)

Should pre allocate file when create new segment? If you are using Kafka on Windows, you probably need to set it to true.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium
Update Mode:	cluster-wide

- [\*\*log.retention.check.interval.ms\*\*](#)

The frequency in milliseconds that the log cleaner checks whether any log is eligible for deletion

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*max.connection.creation.rate\*\*](#)

The maximum connection creation rate we allow in the broker at any time. Listener-level limits may also be configured by prefixing the config name with the listener prefix, for example, `listener.name.internal.max.connection.creation.rate`. Broker-wide connection rate limit should be configured based on broker capacity while listener limits should be configured based on application requirements. New connections will be throttled if either the listener or the broker limit

is reached, with the exception of inter-broker listener. Connections on the inter-broker listener will be throttled only when the listener-level rate limit is reached.

Type:	int
Default:	2147483647
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [\*\*max.connections\*\*](#)

The maximum number of connections we allow in the broker at any time. This limit is applied in addition to any per-ip limits configured using max.connections.per.ip. Listener-level limits may also be configured by prefixing the config name with the listener prefix, for example, `listener.name.internal.max.connections`. Broker-wide limit should be configured based on broker capacity while listener limits should be configured based on application requirements. New connections are blocked if either the listener or broker limit is reached. Connections on the inter-broker listener are permitted even if broker-wide limit is reached. The least recently used connection on another listener will be closed in this case.

Type:	int
Default:	2147483647
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [\*\*max.connections.per.ip\*\*](#)

The maximum number of connections we allow from each ip address. This can be set to 0 if there are overrides configured using max.connections.per.ip.overrides property. New connections from the ip address are dropped if the limit is reached.

Type:	int
Default:	2147483647
Valid Values:	[0,...]
Importance:	medium
Update Mode:	cluster-wide

- [\*\*max.connections.per.ip.overrides\*\*](#)

A comma-separated list of per-ip or hostname overrides to the default maximum number of connections. An example value is "hostName:100,127.0.0.1:200"

<b>Type:</b>	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium
Update Mode:	cluster-wide

- [\*\*max.incremental.fetch.session.cache.slots\*\*](#)

The maximum number of incremental fetch sessions that we will maintain.

<b>Type:</b>	<b>int</b>
Default:	1000
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

- [\*\*num.partitions\*\*](#)

The default number of log partitions per topic

<b>Type:</b>	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*password.encoder.old.secret\*\*](#)

The old secret that was used for encoding dynamically configured passwords. This is required only when the secret is updated. If specified, all dynamically encoded passwords are decoded using this old secret and re-encoded using password.encoder.secret when broker starts up.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*password.encoder.secret\*\*](#)

The secret used for encoding dynamically configured passwords for this broker.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*principal.builder.class\*\*](#)

The fully qualified name of a class that implements the KafkaPrincipalBuilder interface, which is used to build the KafkaPrincipal object used during authorization. If no principal builder is defined, the default behavior depends on the security protocol in use. For SSL authentication, the principal will be derived using the rules defined by `ssl.principal.mapping.rules` applied on the distinguished name from the client certificate if one is provided; otherwise, if client authentication is not required, the principal name will be ANONYMOUS. For SASL authentication, the principal will be derived using the rules defined by `sasl.kerberos.principal.to.local.rules` if GSSAPI is in use, and the SASL authentication ID for other mechanisms. For PLAINTEXT, the principal will be ANONYMOUS.

Type:	<b>class</b>
Default:	org.apache.kafka.common.security.authenticator.DefaultKafkaPrincipalBuilder
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*producer.purgatory.purge.interval.requests\*\*](#)

The purge interval (in number of requests) of the producer request purgatory

Type:	<b>int</b>
Default:	1000
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*queued.max.request.bytes\*\*](#)

The number of queued bytes allowed before no more requests are read

Type:	<b>long</b>
Default:	-1
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [replica.fetch.backoff.ms](#)

The amount of time to sleep when fetch partition error occurs.

Type:	int
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

- [replica.fetch.max.bytes](#)

The number of bytes of messages to attempt to fetch for each partition. This is not an absolute maximum, if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that progress can be made. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config).

Type:	int
Default:	1048576 (1 mebibyte)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

- [replica.fetch.response.max.bytes](#)

Maximum bytes expected for the entire fetch response. Records are fetched in batches, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that progress can be made. As such, this is not an absolute maximum. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config).

Type:	int
Default:	10485760 (10 mebibytes)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

- [replica.selector.class](#)

The fully qualified class name that implements ReplicaSelector. This is used by the broker to find the preferred read replica. By default, we use an implementation that returns the leader.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **reserved.broker.max.id**

Max number that can be used for a broker.id

Type:	<b>int</b>
Default:	1000
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

- **sasl.client.callback.handler.class**

The fully qualified name of a SASL client callback handler class that implements the AuthenticateCallbackHandler interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **sasl.enabled.mechanisms**

The list of SASL mechanisms enabled in the Kafka server. The list may contain any mechanism for which a security provider is available. Only GSSAPI is enabled by default.

Type:	<b>list</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **sasl.jaas.config**

JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described [here](#). The format for the value is: `loginModuleClass controlFlag (optionName=optionValue)*;`. For brokers, the config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;`

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.kinit.cmd\*\*](#)

Kerberos kinit command path.

Type:	<b>string</b>
Default:	/usr/bin/kinit
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.min.time.before.relogin\*\*](#)

Login thread sleep time between refresh attempts.

Type:	<b>long</b>
Default:	60000
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.principal.to.local.rules\*\*](#)

A list of rules for mapping from principal names to short names (typically operating system usernames). The rules are evaluated in order and the first rule that matches a principal name is used to map it to a short name. Any later rules in the list are ignored. By default, principal names of the form `{username}/{hostname}@{REALM}` are mapped to `{username}`. For more details on the format please see [security authorization and acls](#). Note that this configuration is ignored if an extension of `kafkaPrincipalBuilder` is provided by the `principal.builder.class` configuration.

Type:	<b>list</b>
Default:	DEFAULT
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.service.name\*\*](#)

The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.ticket.renew.jitter\*\*](#)

Percentage of random jitter added to the renewal time.

Type:	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.kerberos.ticket.renew.window.factor\*\*](#)

Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*sasl.login.callback.handler.class\*\*](#)

The fully qualified name of a SASL login callback handler class that implements the AuthenticateCallbackHandler interface. For brokers, login callback handler config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [sasl.login.class](#)

The fully qualified name of a class that implements the Login interface. For brokers, login config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [sasl.login.refresh.buffer.seconds](#)

The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and sasl.login.refresh.min.period.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	300
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [sasl.login.refresh.min.period.seconds](#)

The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and sasl.login.refresh.buffer.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	60
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [sasl.login.refresh.window.factor](#)

Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **sasl.login.refresh.window.jitter**

The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **sasl.mechanism.inter.broker.protocol**

SASL mechanism used for inter-broker communication. Default is GSSAPI.

Type:	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **sasl.oauthbearer.jwks.endpoint.url**

The OAuth/OIDC provider URL from which the provider's [JWKS \(JSON Web Key Set\)](#) can be retrieved. The URL can be HTTP(S)-based or file-based. If the URL is HTTP(S)-based, the JWKS data will be retrieved from the OAuth/OIDC provider via the configured URL on broker startup. All then-current keys will be cached on the broker for incoming requests. If an authentication request is received for a JWT that includes a "kid" header claim value that isn't yet in the cache, the JWKS endpoint will be queried again on demand. However, the broker polls the URL every `sasl.oauthbearer.jwks.endpoint.refresh.ms` milliseconds to refresh the cache with any forthcoming keys before any JWT requests that include them are received. If the URL is file-based, the broker will load the JWKS file from a configured location on startup. In the event that the JWT includes a "kid" header value that isn't in the JWKS file, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **sasl.oauthbearer.token.endpoint.url**

The URL for the OAuth/OIDC identity provider. If the URL is HTTP(S)-based, it is the issuer's token endpoint URL to which requests will be made to login based on the configuration in sasl.jaas.config. If the URL is file-based, it specifies a file containing an access token (in JWT serialized form) issued by the OAuth/OIDC identity provider to use for authorization.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **sasl.server.callback.handler.class**

The fully qualified name of a SASL server callback handler class that implements the AuthenticateCallbackHandler interface. Server callback handlers must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.plain.sasl.server.callback.handler.class=com.example.CustomPlainCallbackHandler.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- **sasl.server.max.receive.size**

The maximum receive size allowed before and during initial SASL authentication. Default receive size is 512KB. GSSAPI limits requests to 64K, but we allow upto 512KB by default for custom SASL mechanisms. In practice, PLAIN, SCRAM and OAUTH mechanisms can use much smaller limits.

Type:	<b>int</b>
Default:	524288
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*security.inter.broker.protocol\*\*](#)

Security protocol used to communicate between brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL. It is an error to set this and inter.broker.listener.name properties at the same time.

<b>Type:</b>	<b>string</b>
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium
Update Mode:	read-only

- [\*\*socket.connection.setup.timeout.max.ms\*\*](#)

The maximum amount of time the client will wait for the socket connection to be established. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the timeout resulting in a random range between 20% below and 20% above the computed value.

<b>Type:</b>	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*socket.connection.setup.timeout.ms\*\*](#)

The amount of time the client will wait for the socket connection to be established. If the connection is not built before the timeout elapses, clients will close the socket channel.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*socket.listen.backlog.size\*\*](#)

The maximum number of pending connections on the socket. In Linux, you may also need to configure `somaxconn` and `tcp_max_syn_backlog` kernel parameters accordingly to make the configuration takes effect.

Type:	<b>int</b>
Default:	50
Valid Values:	[1,...]
Importance:	medium
Update Mode:	read-only

- [\*\*ssl.cipher.suites\*\*](#)

A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [\*\*ssl.client.auth\*\*](#)

Configures kafka broker to request client authentication. The following settings are common:

- `ssl.client.auth=required` If set to required client authentication is required.
- `ssl.client.auth=requested` This means client authentication is optional. unlike required, if this option is set client can choose not to provide authentication information about itself
- `ssl.client.auth=None` This means client authentication is not needed.

Type:	<b>string</b>
Default:	none
Valid Values:	[required, requested, none]
Importance:	medium
Update Mode:	per-broker

- [\*\*ssl.enabled.protocols\*\*](#)

The list of protocols enabled for SSL connections. The default is 'TLSv1.2,TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. With the default value for Java 11, clients and servers will prefer TLSv1.3 if both support it and fallback to TLSv1.2 otherwise (assuming both support at least TLSv1.2). This default should be fine for most cases. Also see the config documentation for `ssl.protocol`.

<b>Type:</b>	<b>list</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **ssl.key.password**

The password of the private key in the key store file or the PEM key specified in 'ssl.keystore.key'.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **ssl.keymanager.algorithm**

The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.

<b>Type:</b>	<b>string</b>
Default:	SunX509
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **ssl.keystore.certificate.chain**

Certificate chain in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with a list of X.509 certificates

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **ssl.keystore.key**

Private key in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with PKCS#8 keys. If the key is encrypted, key password must be specified using 'ssl.key.password'

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.keystore.location](#)**

The location of the key store file. This is optional for client and can be used for two-way authentication for client.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.keystore.password](#)**

The store password for the key store file. This is optional for client and only needed if 'ssl.keystore.location' is configured. Key store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.keystore.type](#)**

The file format of the key store file. This is optional for client. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.protocol](#)**

The SSL protocol used to generate the SSLContext. The default is 'TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. This value should be fine for most use cases. Allowed values in recent JVMs are 'TLSv1.2' and 'TLSv1.3'. 'TLS', 'TLSv1.1', 'SSL', 'SSLv2' and 'SSLv3' may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. With the default value for this config and 'ssl.enabled.protocols', clients will downgrade to 'TLSv1.2' if the server does

not support 'TLSv1.3'. If this config is set to 'TLSv1.2', clients will not use 'TLSv1.3' even if it is one of the values in `ssl.enabled.protocols` and the server only supports 'TLSv1.3'.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.provider](#)**

The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.trustmanager.algorithm](#)**

The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

Type:	<b>string</b>
Default:	PKIX
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- **[ssl.truststore.certificates](#)**

Trusted certificates in the format specified by 'ssl.truststore.type'. Default SSL engine factory supports only PEM format with X.509 certificates.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [ssl.truststore.location](#)

The location of the trust store file.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [ssl.truststore.password](#)

The password for the trust store file. If a password is not set, trust store file configured will still be used, but integrity checking is disabled. Trust store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [ssl.truststore.type](#)

The file format of the trust store file. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium
Update Mode:	per-broker

- [zookeeper.clientCnxnSocket](#)

Typically set to `org.apache.zookeeper.ClientCnxnSocketNetty` when using TLS connectivity to ZooKeeper. Overrides any explicit value set via the same-named `zookeeper.clientCnxnSocket` system property.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.client.enable](#)

Set client to use TLS when connecting to ZooKeeper. An explicit value overrides any value set via the `zookeeper.client.secure` system property (note the different name). Defaults to false if neither is set; when true, `zookeeper.clientCnxnSocket` must be set (typically to `org.apache.zookeeper.ClientCnxnSocketNetty`); other values to set may include `zookeeper.ssl.cipher.suites`, `zookeeper.ssl.crl.enable`, `zookeeper.ssl.enabled.protocols`, `zookeeper.ssl.endpoint.identification.algorithm`, `zookeeper.ssl.keystore.location`, `zookeeper.ssl.keystore.password`, `zookeeper.ssl.keystore.type`, `zookeeper.ssl.ocsp.enable`, `zookeeper.ssl.protocol`, `zookeeper.ssl.truststore.location`, `zookeeper.ssl.truststore.password`, `zookeeper.ssl.truststore.type`

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.keystore.location](#)

Keystore location when using a client-side certificate with TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.keystore.location` system property (note the camelCase).

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.keystore.password](#)

Keystore password when using a client-side certificate with TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.keystore.password` system property (note the camelCase). Note that ZooKeeper does not support a key password different from the keystore password, so be sure to set the key password in the keystore to be identical to the keystore password; otherwise the connection attempt to Zookeeper will fail.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.keystore.type](#)

Keystore type when using a client-side certificate with TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.keystore.type` system property (note the camelCase). The default value of `null` means the type will be auto-detected based on the filename extension of the keystore.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.truststore.location](#)

Truststore location when using TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.truststore.location` system property (note the camelCase).

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.truststore.password](#)

Truststore password when using TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.trustStore.password` system property (note the camelCase).

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [zookeeper.ssl.truststore.type](#)

Truststore type when using TLS connectivity to ZooKeeper. Overrides any explicit value set via the `zookeeper.ssl.trustStore.type` system property (note the camelCase). The default value of `null` means the type will be auto-detected based on the filename extension of the truststore.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium
Update Mode:	read-only

- [\*\*alter.config.policy.class.name\*\*](#)

The alter configs policy class that should be used for validation. The class should implement the `org.apache.kafka.server.policy.AlterConfigPolicy` interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*alter.log.dirs.replication.quota.window.num\*\*](#)

The number of samples to retain in memory for alter log dirs replication quotas

Type:	<b>int</b>
Default:	11
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*alter.log.dirs.replication.quota.window.size.seconds\*\*](#)

The time span of each sample for alter log dirs replication quotas

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*authorizer.class.name\*\*](#)

The fully qualified name of a class that implements `org.apache.kafka.server.authorizer.Authorizer` interface, which is used by the broker for authorization.

Type:	<b>string</b>
Default:	""
Valid Values:	non-null string
Importance:	low
Update Mode:	read-only

- [\*\*auto.include.jmx.reporter\*\*](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*client.quota.callback.class\*\*](#)

The fully qualified name of a class that implements the ClientQuotaCallback interface, which is used to determine quota limits applied to client requests. By default, the `and` quotas that are stored in ZooKeeper are applied. For any given request, the most specific quota that matches the user principal of the session and the client-id of the request is applied.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*connection.failed.authentication.delay.ms\*\*](#)

Connection close delay on failed authentication: this is the time (in milliseconds) by which connection close will be delayed on authentication failure. This must be configured to be less than `connections.max.idle.ms` to prevent connection timeout.

Type:	<b>int</b>
Default:	100
Valid Values:	[0,...]
Importance:	low
Update Mode:	read-only

- [\*\*controller.quorum.retry.backoff.ms\*\*](#)

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

Type:	<b>int</b>
Default:	20
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*controller.quota.window.num\*\*](#)

The number of samples to retain in memory for controller mutation quotas

Type:	<b>int</b>
Default:	11
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*controller.quota.window.size.seconds\*\*](#)

The time span of each sample for controller mutations quotas

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*create.topic.policy.class.name\*\*](#)

The create topic policy class that should be used for validation. The class should implement the `org.apache.kafka.server.policy.CreateTopicPolicy` interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*delegation.token.expiry.check.interval.ms\*\*](#)

Scan interval to remove expired delegation tokens.

Type:	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*kafka.metrics.polling.interval.secs\*\*](#)

The metrics polling interval (in seconds) which can be used in kafka.metrics.reporters implementations.

Type:	<b>int</b>
Default:	10
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*kafka.metrics.reporters\*\*](#)

A list of classes to use as Yammer metrics custom reporters. The reporters should implement `kafka.metrics.KafkaMetricsReporter` trait. If a client wants to expose JMX operations on a custom reporter, the custom reporter needs to additionally implement an MBean trait that extends `kafka.metrics.KafkaMetricsReporterMBean` trait so that the registered MBean is compliant with the standard MBean convention.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*listener.security.protocol.map\*\*](#)

Map between listener names and security protocols. This must be defined for the same security protocol to be usable in more than one port or IP. For example, internal and external traffic can be separated even if SSL is required for both. Concretely, the user could define listeners with names INTERNAL and EXTERNAL and this property as: `INTERNAL:SSL,EXTERNAL:SSL`. As shown, key and value are separated by a colon and map entries are separated by commas. Each listener name should only appear once in the map. Different security (SSL and SASL) settings can be configured for each listener by adding a normalised prefix (the listener name is lowercased) to the config name. For example, to set a different keystore for the INTERNAL listener, a config with name `listener.name.internal.ssl.keystore.location` would be set. If the config for the listener name is not set, the config will fallback to the generic config (i.e. `ssl.keystore.location`). Note

that in KRaft a default mapping from the listener names defined by `controller.listener.names` to PLAINTEXT is assumed if no explicit mapping is provided and no other security protocol is in use.

Type:	<b>string</b>
Default:	PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
Valid Values:	
Importance:	low
Update Mode:	per-broker

- [\*\*log.message.downconversion.enable\*\*](#)

This configuration controls whether down-conversion of message formats is enabled to satisfy consume requests. When set to `false`, broker will not perform down-conversion for consumers expecting an older message format. The broker responds with `UNSUPPORTED_VERSION` error for consume requests from such older clients. This configuration does not apply to any message format conversion that might be required for replication to followers.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low
Update Mode:	cluster-wide

- [\*\*metadata.max.idle.interval.ms\*\*](#)

This configuration controls how often the active controller should write no-op records to the metadata partition. If the value is 0, no-op records are not appended to the metadata partition. The default value is 500

Type:	<b>int</b>
Default:	500
Valid Values:	[0,...]
Importance:	low
Update Mode:	read-only

- [\*\*metric.reporters\*\*](#)

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	low
Update Mode:	cluster-wide

- [metrics.num.samples](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [metrics.recording.level](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	
Importance:	low
Update Mode:	read-only

- [metrics.sample.window.ms](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [password.encoder.cipher.algorithm](#)

The Cipher algorithm used for encoding dynamically configured passwords.

Type:	<b>string</b>
Default:	AES/CBC/PKCS5Padding
Valid Values:	
Importance:	low
Update Mode:	read-only

- [password.encoder.iterations](#)

The iteration count used for encoding dynamically configured passwords.

Type:	<b>int</b>
Default:	4096
Valid Values:	[1024,...]
Importance:	low
Update Mode:	read-only

- **[password.encoder.key.length](#)**

The key length used for encoding dynamically configured passwords.

Type:	<b>int</b>
Default:	128
Valid Values:	[8,...]
Importance:	low
Update Mode:	read-only

- **[password.encoder.keyfactory.algorithm](#)**

The SecretKeyFactory algorithm used for encoding dynamically configured passwords. Default is PBKDF2WithHmacSHA512 if available and PBKDF2WithHmacSHA1 otherwise.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- **[producer.id.expiration.ms](#)**

The time in ms that a topic partition leader will wait before expiring producer IDs. Producer IDs will not expire while a transaction associated to them is still ongoing. Note that producer IDs may expire sooner if the last write from the producer ID is deleted due to the topic's retention settings. Setting this value the same or higher than `delivery.timeout.ms` can help prevent expiration during retries and protect against message duplication, but the default should be reasonable for most use cases.

Type:	<b>int</b>
Default:	86400000 (1 day)
Valid Values:	[1,...]
Importance:	low
Update Mode:	cluster-wide

- [quota.window.num](#)

The number of samples to retain in memory for client quotas

Type:	int
Default:	11
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [quota.window.size.seconds](#)

The time span of each sample for client quotas

Type:	int
Default:	1
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [replication.quota.window.num](#)

The number of samples to retain in memory for replication quotas

Type:	int
Default:	11
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [replication.quota.window.size.seconds](#)

The time span of each sample for replication quotas

Type:	int
Default:	1
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [sasl.login.connect.timeout.ms](#)

The (optional) value in milliseconds for the external authentication provider connection timeout.  
Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.login.read.timeout.ms\*\*](#)

The (optional) value in milliseconds for the external authentication provider read timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.login.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.login.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low
Update Mode:	read-only

- **sasl.oauthbearer.clock.skew.seconds**

The (optional) value in seconds to allow for differences between the time of the OAuth/OIDC identity provider and the broker.

Type:	int
Default:	30
Valid Values:	
Importance:	low
Update Mode:	read-only

- **sasl.oauthbearer.expected.audience**

The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth "aud" claim and if this value is set, the broker will match the value from JWT's "aud" claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	list
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- **sasl.oauthbearer.expected.issuer**

The (optional) setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth "iss" claim and if this value is set, the broker will match it exactly against what is in the JWT's "iss" claim. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	string
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- **sasl.oauthbearer.jwks.endpoint.refresh.ms**

The (optional) value in milliseconds for the broker to wait between refreshing its JWKS (JSON Web Key Set) cache that contains the keys to verify the signature of the JWT.

Type:	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between attempts to retrieve the JWKS (JSON Web Key Set) from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between JWKS (JSON Web Key Set) retrieval attempts from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*sasl.oauthbearer.scope.claim.name\*\*](#)

The OAuth claim for the scope is often named "scope", but this (optional) setting can provide a different name to use for the scope included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	scope
Valid Values:	
Importance:	low
Update Mode:	read-only

- [sasl.oauthbearer.sub.claim.name](#)

The OAuth claim for the subject is often named "sub", but this (optional) setting can provide a different name to use for the subject included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	sub
Valid Values:	
Importance:	low
Update Mode:	read-only

- [security.providers](#)

A list of configurable creator classes each returning a provider implementing security algorithms. These classes should implement the

`org.apache.kafka.common.security.auth.SecurityProviderCreator` interface.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [ssl.endpoint.identification.algorithm](#)

The endpoint identification algorithm to validate server hostname using server certificate.

Type:	<b>string</b>
Default:	https
Valid Values:	
Importance:	low
Update Mode:	per-broker

- [ssl.engine.factory.class](#)

The class of type `org.apache.kafka.common.security.auth.SslEngineFactory` to provide SSLEngine objects. Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	per-broker

- [\*\*ssl.principal.mapping.rules\*\*](#)

A list of rules for mapping from distinguished name from the client certificate to short name. The rules are evaluated in order and the first rule that matches a principal name is used to map it to a short name. Any later rules in the list are ignored. By default, distinguished name of the X.500 certificate will be the principal. For more details on the format please see [security\\_authorization\\_and\\_acls](#). Note that this configuration is ignored if an extension of KafkaPrincipalBuilder is provided by the `principal.builder.class` configuration.

Type:	<b>string</b>
Default:	DEFAULT
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*ssl.secure.random.implementation\*\*](#)

The SecureRandom PRNG implementation to use for SSL cryptography operations.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low
Update Mode:	per-broker

- [\*\*transaction.abort.timed.out.transaction.cleanup.interval.ms\*\*](#)

The interval at which to rollback transactions that have timed out

Type:	<b>int</b>
Default:	10000 (10 seconds)
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [\*\*transaction.remove.expired.transaction.cleanup.interval.ms\*\*](#)

The interval at which to remove transactions that have expired due to `transactional.id.expiration.ms` passing

Type:	<b>int</b>
Default:	3600000 (1 hour)
Valid Values:	[1,...]
Importance:	low
Update Mode:	read-only

- [zookeeper.ssl.cipher.suites](#)

Specifies the enabled cipher suites to be used in ZooKeeper TLS negotiation (csv). Overrides any explicit value set via the `zookeeper.ssl.ciphersuites` system property (note the single word "ciphersuites"). The default value of `null` means the list of enabled cipher suites is determined by the Java runtime being used.

Type:	list
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [zookeeper.ssl.crl.enable](#)

Specifies whether to enable Certificate Revocation List in the ZooKeeper TLS protocols. Overrides any explicit value set via the `zookeeper.ssl.crl` system property (note the shorter name).

Type:	boolean
Default:	false
Valid Values:	
Importance:	low
Update Mode:	read-only

- [zookeeper.ssl.enabled.protocols](#)

Specifies the enabled protocol(s) in ZooKeeper TLS negotiation (csv). Overrides any explicit value set via the `zookeeper.ssl.enabledProtocols` system property (note the camelCase). The default value of `null` means the enabled protocol will be the value of the `zookeeper.ssl.protocol` configuration property.

Type:	list
Default:	null
Valid Values:	
Importance:	low
Update Mode:	read-only

- [zookeeper.ssl.endpoint.identification.algorithm](#)

Specifies whether to enable hostname verification in the ZooKeeper TLS negotiation process, with (case-insensitively) "https" meaning ZooKeeper hostname verification is enabled and an explicit blank value meaning it is disabled (disabling it is only recommended for testing purposes). An explicit value overrides any "true" or "false" value set via the `zookeeper.ssl.hostnameverification` system property (note the different name and values; true implies https and false implies blank).

Type:	<b>string</b>
Default:	HTTPS
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*zookeeper.ssl.ocsp.enable\*\*](#)

Specifies whether to enable Online Certificate Status Protocol in the ZooKeeper TLS protocols. Overrides any explicit value set via the `zookeeper.ssl.ocsp` system property (note the shorter name).

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	low
Update Mode:	read-only

- [\*\*zookeeper.ssl.protocol\*\*](#)

Specifies the protocol to be used in ZooKeeper TLS negotiation. An explicit value overrides any value set via the same-named `zookeeper.ssl.protocol` system property.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	low
Update Mode:	read-only

More details about broker configuration can be found in the scala class `kafka.server.KafkaConfig`.

### [\*\*3.1.1 Updating Broker Configs\*\*](#)

From Kafka version 1.1 onwards, some of the broker configs can be updated without restarting the broker. See the `Dynamic Update Mode` column in [Broker Configs](#) for the update mode of each broker config.

- `read-only`: Requires a broker restart for update
- `per-broker`: May be updated dynamically for each broker
- `cluster-wide`: May be updated dynamically as a cluster-wide default. May also be updated as a per-broker value for testing.

To alter the current broker configs for broker id 0 (for example, the number of log cleaner threads):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --
  entity-name 0 --alter --add-config log.cleaner.threads=2
```

To describe the current dynamic broker configs for broker id 0:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --describe
```

To delete a config override and revert to the statically configured or default value for broker id 0 (for example, the number of log cleaner threads):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --alter --delete-config log.cleaner.threads
```

Some configs may be configured as a cluster-wide default to maintain consistent values across the whole cluster. All brokers in the cluster will process the cluster default update. For example, to update log cleaner threads on all brokers:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-default --alter --add-config log.cleaner.threads=2
```

To describe the currently configured dynamic cluster-wide default configs:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-default --describe
```

All configs that are configurable at cluster level may also be configured at per-broker level (e.g. for testing). If a config value is defined at different levels, the following order of precedence is used:

- Dynamic per-broker config stored in ZooKeeper
- Dynamic cluster-wide default config stored in ZooKeeper
- Static broker config from `server.properties`
- Kafka default, see [broker configs](#)

## Updating Password Configs Dynamically

Password config values that are dynamically updated are encrypted before storing in ZooKeeper. The broker config `password.encoder.secret` must be configured in `server.properties` to enable dynamic update of password configs. The secret may be different on different brokers.

The secret used for password encoding may be rotated with a rolling restart of brokers. The old secret used for encoding passwords currently in ZooKeeper must be provided in the static broker config `password.encoder.old.secret` and the new secret must be provided in `password.encoder.secret`. All dynamic password configs stored in ZooKeeper will be re-encoded with the new secret when the broker starts up.

In Kafka 1.1.x, all dynamically updated password configs must be provided in every alter request when updating configs using `kafka-configs.sh` even if the password config is not being altered. This constraint will be removed in a future release.

## Updating Password Configs in ZooKeeper Before Starting Brokers

From Kafka 2.0.0 onwards, `kafka-configs.sh` enables dynamic broker configs to be updated using ZooKeeper before starting brokers for bootstrapping. This enables all password configs to be stored in encrypted form, avoiding the need for clear passwords in `server.properties`. The broker config `password.encoder.secret` must also be specified if any password configs are included in the alter command. Additional encryption parameters may also be specified. Password encoder configs will not be persisted in ZooKeeper. For example, to store SSL key password for listener `INTERNAL` on broker 0:

```
> bin/kafka-configs.sh --zookeeper localhost:2182 --zk-tls-config-file zk_tls_config.properties --entity-type brokers --entity-name 0 --alter --add-config 'listener.name.internal.ssl.key.password=key-password,password.encoder.secret=secret,password.encoder.iterations=8192'
```

The configuration `listener.name.internal.ssl.key.password` will be persisted in ZooKeeper in encrypted form using the provided encoder configs. The encoder secret and iterations are not persisted in ZooKeeper.

### Updating SSL Keystore of an Existing Listener

Brokers may be configured with SSL keystores with short validity periods to reduce the risk of compromised certificates. Keystores may be updated dynamically without restarting the broker. The config name must be prefixed with the listener prefix `listener.name.{listenerName}`, so that only the keystore config of a specific listener is updated. The following configs may be updated in a single alter request at per-broker level:

- `ssl.keystore.type`
- `ssl.keystore.location`
- `ssl.keystore.password`
- `ssl.key.password`

If the listener is the inter-broker listener, the update is allowed only if the new keystore is trusted by the truststore configured for that listener. For other listeners, no trust validation is performed on the keystore by the broker. Certificates must be signed by the same certificate authority that signed the old certificate to avoid any client authentication failures.

### Updating SSL Truststore of an Existing Listener

Broker truststores may be updated dynamically without restarting the broker to add or remove certificates. Updated truststore will be used to authenticate new client connections. The config name must be prefixed with the listener prefix `listener.name.{listenerName}`, so that only the truststore config of a specific listener is updated. The following configs may be updated in a single alter request at per-broker level:

- `ssl.truststore.type`
- `ssl.truststore.location`
- `ssl.truststore.password`

If the listener is the inter-broker listener, the update is allowed only if the existing keystore for that listener is trusted by the new truststore. For other listeners, no trust validation is performed by the broker before the update. Removal of CA certificates used to sign client certificates from the new truststore can lead to client authentication failures.

### Updating Default Topic Configuration

Default topic configuration options used by brokers may be updated without broker restart. The configs are applied to topics without a topic config override for the equivalent per-topic config. One or more of these configs may be overridden at cluster-default level used by all brokers.

- `log.segment.bytes`
- `log.roll.ms`
- `log.roll.hours`
- `log.roll.jitter.ms`
- `log.roll.jitter.hours`
- `log.index.size.max.bytes`
- `log.flush.interval.messages`
- `log.flush.interval.ms`
- `log.retention.bytes`

- `log.retention.ms`
- `log.retention.minutes`
- `log.retention.hours`
- `log.index.interval.bytes`
- `log.cleaner.delete.retention.ms`
- `log.cleaner.min.compaction.lag.ms`
- `log.cleaner.max.compaction.lag.ms`
- `log.cleaner.min.cleanable.ratio`
- `log.cleanup.policy`
- `log.segment.delete.delay.ms`
- `unclean.leader.election.enable`
- `min.insync.replicas`
- `max.message.bytes`
- `compression.type`
- `log.preallocate`
- `log.message.timestamp.type`
- `log.message.timestamp.difference.max.ms`

From Kafka version 2.0.0 onwards, unclean leader election is automatically enabled by the controller when the config `unclean.leader.election.enable` is dynamically updated. In Kafka version 1.1.x, changes to `unclean.leader.election.enable` take effect only when a new controller is elected.

Controller re-election may be forced by running:

```
> bin/zookeeper-shell.sh localhost
      rmr /controller
```

## Updating Log Cleaner Configs

Log cleaner configs may be updated dynamically at cluster-default level used by all brokers. The changes take effect on the next iteration of log cleaning. One or more of these configs may be updated:

- `log.cleaner.threads`
- `log.cleaner.io.max.bytes.per.second`
- `log.cleaner.dedupe.buffer.size`
- `log.cleaner.io.buffer.size`
- `log.cleaner.io.buffer.load.factor`
- `log.cleaner.backoff.ms`

## Updating Thread Configs

The size of various thread pools used by the broker may be updated dynamically at cluster-default level used by all brokers. Updates are restricted to the range `currentSize / 2` to `currentSize * 2` to ensure that config updates are handled gracefully.

- `num.network.threads`
- `num.io.threads`
- `num.replica.fetchers`
- `num.recovery.threads.per.data.dir`
- `log.cleaner.threads`
- `background.threads`

## Updating ConnectionQuota Configs

The maximum number of connections allowed for a given IP/host by the broker may be updated dynamically at cluster-default level used by all brokers. The changes will apply for new connection creations and the existing connections count will be taken into account by the new limits.

- `max.connections.per.ip`
- `max.connections.per.ip.overrides`

## Adding and Removing Listeners

Listeners may be added or removed dynamically. When a new listener is added, security configs of the listener must be provided as listener configs with the listener prefix `listener.name.{listenerName}`. If the new listener uses SASL, the JAAS configuration of the listener must be provided using the JAAS configuration property `sasl.jaas.config` with the listener and mechanism prefix. See [JAAS configuration for Kafka brokers](#) for details.

In Kafka version 1.1.x, the listener used by the inter-broker listener may not be updated dynamically. To update the inter-broker listener to a new listener, the new listener may be added on all brokers without restarting the broker. A rolling restart is then required to update `inter.broker.listener.name`.

In addition to all the security configs of new listeners, the following configs may be updated dynamically at per-broker level:

- `listeners`
- `advertised.listeners`
- `listener.security.protocol.map`

Inter-broker listener must be configured using the static broker configuration `inter.broker.listener.name` or `security.inter.broker.protocol`.

## 3.2 Topic-Level Configs

Configurations pertinent to topics have both a server default as well an optional per-topic override. If no per-topic configuration is given the server default is used. The override can be set at topic creation time by giving one or more `--config` options. This example creates a topic named *my-topic* with a custom max message size and flush rate:

```
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-topic --  
partitions 1 \  
--replication-factor 1 --config max.message.bytes=64000 --config flush.messages=1
```

Overrides can also be changed or set later using the alter config command. This example updates the max message size for *my-topic*:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --  
entity-name my-topic  
--alter --add-config max.message.bytes=128000
```

To check overrides set on the topic you can do

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --  
entity-name my-topic --describe
```

To remove an override you can do

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --
  entity-name my-topic
    --alter --delete-config max.message.bytes
```

The following are the topic-level configurations. The server's default configuration for this property is given under the Server Default Property heading. A given server default config value only applies to a topic if it does not have an explicit topic config override.

- [\*\*cleanup.policy\*\*](#)

This config designates the retention policy to use on log segments. The "delete" policy (which is the default) will discard old segments when their retention time or size limit has been reached. The "compact" policy will enable [log compaction](#), which retains the latest value for each key. It is also possible to specify both policies in a comma-separated list (e.g. "delete,compact"). In this case, old segments will be discarded per the retention time and size configuration, while retained segments will be compacted.

Type:	list
Default:	delete
Valid Values:	[compact, delete]
Server Default Property:	log.cleanup.policy
Importance:	medium

- [\*\*compression.type\*\*](#)

Specify the final compression type for a given topic. This configuration accepts the standard compression codecs ('gzip', 'snappy', 'lz4', 'zstd'). It additionally accepts 'uncompressed' which is equivalent to no compression; and 'producer' which means retain the original compression codec set by the producer.

Type:	string
Default:	producer
Valid Values:	[uncompressed, zstd, lz4, snappy, gzip, producer]
Server Default Property:	compression.type
Importance:	medium

- [\*\*delete.retention.ms\*\*](#)

The amount of time to retain delete tombstone markers for [log compacted](#) topics. This setting also gives a bound on the time in which a consumer must complete a read if they begin from offset 0 to ensure that they get a valid snapshot of the final stage (otherwise delete tombstones may be collected before they complete their scan).

Type:	long
Default:	86400000 (1 day)
Valid Values:	[0,...]
Server Default Property:	log.cleaner.delete.retention.ms
Importance:	medium

- [\*\*file.delete.delay.ms\*\*](#)

The time to wait before deleting a file from the filesystem

Type:	<b>long</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Server Default Property:	log.segment.delete.delay.ms
Importance:	medium

- [\*\*flush.messages\*\*](#)

This setting allows specifying an interval at which we will force an fsync of data written to the log. For example if this was set to 1 we would fsync after every message; if it were 5 we would fsync after every five messages. In general we recommend you not set this and use replication for durability and allow the operating system's background flush capabilities as it is more efficient. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](#)).

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[1,...]
Server Default Property:	log.flush.interval.messages
Importance:	medium

- [\*\*flush.ms\*\*](#)

This setting allows specifying a time interval at which we will force an fsync of data written to the log. For example if this was set to 1000 we would fsync after 1000 ms had passed. In general we recommend you not set this and use replication for durability and allow the operating system's background flush capabilities as it is more efficient.

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[0,...]
Server Default Property:	log.flush.interval.ms
Importance:	medium

- [\*\*follower.replication.throttled.replicas\*\*](#)

A list of replicas for which log replication should be throttled on the follower side. The list should describe a set of replicas in the form [PartitionId]:[BrokerId],[PartitionId]:[BrokerId]:... or alternatively the wildcard '\*' can be used to throttle all replicas for this topic.

Type:	list
Default:	""
Valid Values:	[partitionId]:[brokerId],[partitionId]:[brokerId],...
Server Default Property:	follower.replication.throttled.replicas
Importance:	medium

- [\*\*index.interval.bytes\*\*](#)

This setting controls how frequently Kafka adds an index entry to its offset index. The default setting ensures that we index a message roughly every 4096 bytes. More indexing allows reads to jump closer to the exact position in the log but makes the index larger. You probably don't need to change this.

Type:	int
Default:	4096 (4 kibibytes)
Valid Values:	[0,...]
Server Default Property:	log.index.interval.bytes
Importance:	medium

- [\*\*leader.replication.throttled.replicas\*\*](#)

A list of replicas for which log replication should be throttled on the leader side. The list should describe a set of replicas in the form [PartitionId]:[BrokerId],[PartitionId]:[BrokerId]... or alternatively the wildcard '\*' can be used to throttle all replicas for this topic.

Type:	list
Default:	""
Valid Values:	[partitionId]:[brokerId],[partitionId]:[brokerId],...
Server Default Property:	leader.replication.throttled.replicas
Importance:	medium

- [\*\*max.compaction.lag.ms\*\*](#)

The maximum time a message will remain ineligible for compaction in the log. Only applicable for logs that are being compacted.

Type:	long
Default:	9223372036854775807
Valid Values:	[1,...]
Server Default Property:	log.cleaner.max.compaction.lag.ms
Importance:	medium

- [\*\*max.message.bytes\*\*](#)

The largest record batch size allowed by Kafka (after compression if compression is enabled). If this is increased and there are consumers older than 0.10.2, the consumers' fetch size must also be increased so that they can fetch record batches this large. In the latest message format version, records are always grouped into batches for efficiency. In previous message format versions, uncompressed records are not grouped into batches and this limit only applies to a single record in that case.

<b>Type:</b>	<b>int</b>
Default:	1048588
Valid Values:	[0,...]
Server Default Property:	message.max.bytes
Importance:	medium

- [\*\*message.format.version\*\*](#)

[DEPRECATED] Specify the message format version the broker will use to append messages to the logs. The value of this config is always assumed to be `3.0` if `inter.broker.protocol.version` is 3.0 or higher (the actual config value is ignored). Otherwise, the value should be a valid ApiVersion. Some examples are: 0.10.0, 1.1, 2.8, 3.0. By setting a particular message format version, the user is certifying that all the existing messages on disk are smaller or equal than the specified version. Setting this value incorrectly will cause consumers with older versions to break as they will receive messages with a format that they don't understand.

<b>Type:</b>	<b>string</b>
Default:	3.0-IV1
Valid Values:	[0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.10.0-IV0, 0.10.0-IV1, 0.10.1-IV0, 0.10.1-IV1, 0.10.1-IV2, 0.10.2-IV0, 0.11.0-IV0, 0.11.0-IV1, 0.11.0-IV2, 1.0-IV0, 1.1-IV0, 2.0-IV0, 2.0-IV1, 2.1-IV0, 2.1-IV1, 2.1-IV2, 2.2-IV0, 2.2-IV1, 2.3-IV0, 2.3-IV1, 2.4-IV0, 2.4-IV1, 2.5-IV0, 2.6-IV0, 2.7-IV0, 2.7-IV1, 2.7-IV2, 2.8-IV0, 2.8-IV1, 3.0-IV0, 3.0-IV1, 3.1-IV0, 3.2-IV0, 3.3-IV0, 3.3-IV1, 3.3-IV2, 3.3-IV3, 3.4-IV0]
Server Default Property:	log.message.format.version
Importance:	medium

- [\*\*message.timestamp.difference.max.ms\*\*](#)

The maximum difference allowed between the timestamp when a broker receives a message and the timestamp specified in the message. If `message.timestamp.type=CreateTime`, a message will be rejected if the difference in timestamp exceeds this threshold. This configuration is ignored if `message.timestamp.type=LogAppendTime`.

Type:	<b>long</b>
Default:	9223372036854775807
Valid Values:	[0,...]
Server Default Property:	log.message.timestamp.difference.max.ms
Importance:	medium

- [\*\*message.timestamp.type\*\*](#)

Define whether the timestamp in the message is message create time or log append time. The value should be either `CreateTime` or `LogAppendTime`

Type:	<b>string</b>
Default:	CreateTime
Valid Values:	[CreateTime, LogAppendTime]
Server Default Property:	log.message.timestamp.type
Importance:	medium

- [\*\*min.cleanable.dirty.ratio\*\*](#)

This configuration controls how frequently the log compactor will attempt to clean the log (assuming [log compaction](#) is enabled). By default we will avoid cleaning a log where more than 50% of the log has been compacted. This ratio bounds the maximum space wasted in the log by duplicates (at 50% at most 50% of the log could be duplicates). A higher ratio will mean fewer, more efficient cleanings but will mean more wasted space in the log. If the `max.compaction.lag.ms` or the `min.compaction.lag.ms` configurations are also specified, then the log compactor considers the log to be eligible for compaction as soon as either: (i) the dirty ratio threshold has been met and the log has had dirty (uncompacted) records for at least the `min.compaction.lag.ms` duration, or (ii) if the log has had dirty (uncompacted) records for at most the `max.compaction.lag.ms` period.

Type:	<b>double</b>
Default:	0.5
Valid Values:	[0,...,1]
Server Default Property:	log.cleaner.min.cleanable.ratio
Importance:	medium

- [\*\*min.compaction.lag.ms\*\*](#)

The minimum time a message will remain uncompacted in the log. Only applicable for logs that are being compacted.

Type:	<b>long</b>
Default:	0
Valid Values:	[0,...]
Server Default Property:	log.cleaner.min.compaction.lag.ms
Importance:	medium

- [min.insync.replicas](#)

When a producer sets acks to "all" (or "-1"), this configuration specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).

When used together, `min.insync.replicas` and `acks` allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of 3, set `min.insync.replicas` to 2, and produce with `acks` of "all". This will ensure that the producer raises an exception if a majority of replicas do not receive a write.

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Server Default Property:	min.insync.replicas
Importance:	medium

- [preallocate](#)

True if we should preallocate the file on disk when creating a new log segment.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Server Default Property:	log.preallocate
Importance:	medium

- [retention.bytes](#)

This configuration controls the maximum size a partition (which consists of log segments) can grow to before we will discard old log segments to free up space if we are using the "delete" retention policy. By default there is no size limit only a time limit. Since this limit is enforced at the partition level, multiply it by the number of partitions to compute the topic retention in bytes.

Type:	<b>long</b>
Default:	-1
Valid Values:	
Server Default Property:	log.retention.bytes
Importance:	medium

- [retention.ms](#)

This configuration controls the maximum time we will retain a log before we will discard old log segments to free up space if we are using the "delete" retention policy. This represents an SLA on how soon consumers must read their data. If set to -1, no time limit is applied.

Type:	<b>long</b>
Default:	604800000 (7 days)
Valid Values:	[-1,...]
Server Default Property:	log.retention.ms
Importance:	medium

- **segment.bytes**

This configuration controls the segment file size for the log. Retention and cleaning is always done a file at a time so a larger segment size means fewer files but less granular control over retention.

Type:	<b>int</b>
Default:	1073741824 (1 gibibyte)
Valid Values:	[14,...]
Server Default Property:	log.segment.bytes
Importance:	medium

- **segment.index.bytes**

This configuration controls the size of the index that maps offsets to file positions. We preallocate this index file and shrink it only after log rolls. You generally should not need to change this setting.

Type:	<b>int</b>
Default:	10485760 (10 mebibytes)
Valid Values:	[4,...]
Server Default Property:	log.index.size.max.bytes
Importance:	medium

- **segment.jitter.ms**

The maximum random jitter subtracted from the scheduled segment roll time to avoid thundering herds of segment rolling

Type:	<b>long</b>
Default:	0
Valid Values:	[0,...]
Server Default Property:	log.roll.jitter.ms
Importance:	medium

- **segment.ms**

This configuration controls the period of time after which Kafka will force the log to roll even if the segment file isn't full to ensure that retention can delete or compact old data.

Type:	<b>long</b>
Default:	604800000 (7 days)
Valid Values:	[1,...]
Server Default Property:	log.roll.ms
Importance:	medium

- [\*\*unclean.leader.election.enable\*\*](#)

Indicates whether to enable replicas not in the ISR set to be elected as leader as a last resort, even though doing so may result in data loss.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Server Default Property:	unclean.leader.election.enable
Importance:	medium

- [\*\*message.downconversion.enable\*\*](#)

This configuration controls whether down-conversion of message formats is enabled to satisfy consume requests. When set to `false`, broker will not perform down-conversion for consumers expecting an older message format. The broker responds with `UNsupported_version` error for consume requests from such older clients. This configuration does not apply to any message format conversion that might be required for replication to followers.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Server Default Property:	log.message.downconversion.enable
Importance:	low

### 3.3 Producer Configs

Below is the configuration of the producer:

- [\*\*key.serializer\*\*](#)

Serializer class for key that implements the `org.apache.kafka.common.serialization.Serializer` interface.

Type:	<b>class</b>
Default:	
Valid Values:	
Importance:	high

- [value.serializer](#)

Serializer class for value that implements the `org.apache.kafka.common.serialization.Serializer` interface.

Type:	class
Default:	
Valid Values:	
Importance:	high

- [bootstrap.servers](#)

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

Type:	list
Default:	""
Valid Values:	non-null string
Importance:	high

- [buffer.memory](#)

The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will block for `max.block.ms` after which it will throw an exception.

This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.

Type:	long
Default:	33554432
Valid Values:	[0,...]
Importance:	high

- [compression.type](#)

The compression type for all data generated by the producer. The default is none (i.e. no compression). Valid values are `none`, `gzip`, `snappy`, `lz4`, or `zstd`. Compression is of full batches of data, so the efficacy of batching will also impact the compression ratio (more batching means better compression).

<b>Type:</b>	<b>string</b>
Default:	none
Valid Values:	[none, gzip, snappy, lz4, zstd]
Importance:	high

- [\*\*retries\*\*](#)

Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Produce requests will be failed before the number of retries has been exhausted if the timeout configured by `delivery.timeout.ms` expires first before successful acknowledgement. Users should generally prefer to leave this config unset and instead use `delivery.timeout.ms` to control retry behavior.

Enabling idempotence requires this config value to be greater than 0. If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled.

Allowing retries while setting `enable.idempotence` to `false` and `max.in.flight.requests.per.connection` to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first.

<b>Type:</b>	<b>int</b>
Default:	2147483647
Valid Values:	[0,...,2147483647]
Importance:	high

- [\*\*ssl.key.password\*\*](#)

The password of the private key in the key store file or the PEM key specified in 'ssl.keystore.key'.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.certificate.chain\*\*](#)

Certificate chain in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with a list of X.509 certificates

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.key\*\*](#)

Private key in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with PKCS#8 keys. If the key is encrypted, key password must be specified using 'ssl.key.password'

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.location\*\*](#)

The location of the key store file. This is optional for client and can be used for two-way authentication for client.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.password\*\*](#)

The store password for the key store file. This is optional for client and only needed if 'ssl.keystore.location' is configured. Key store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.certificates\*\*](#)

Trusted certificates in the format specified by 'ssl.truststore.type'. Default SSL engine factory supports only PEM format with X.509 certificates.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.location\*\*](#)

The location of the trust store file.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- **[ssl.truststore.password](#)**

The password for the trust store file. If a password is not set, trust store file configured will still be used, but integrity checking is disabled. Trust store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- **[batch.size](#)**

The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes.

No attempt will be made to batch records larger than this size.

Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.

A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.

Note: This setting gives the upper bound of the batch size to be sent. If we have fewer than this many bytes accumulated for this partition, we will 'linger' for the `linger.ms` time waiting for more records to show up. This `linger.ms` setting defaults to 0, which means we'll immediately send out a record even the accumulated batch size is under this `batch.size` setting.

Type:	<b>int</b>
Default:	16384
Valid Values:	[0,...]
Importance:	medium

- **[client.dns.lookup](#)**

Controls how the client uses DNS lookups. If set to `use_all_dns_ips`, connect to each returned IP address in sequence until a successful connection is established. After a disconnection, the next IP is used. Once all IPs have been used once, the client resolves the IP(s) from the hostname again (both the JVM and the OS cache DNS name lookups, however). If set to `resolve_canonical_bootstrap_servers_only`, resolve each bootstrap address into a list of canonical names. After the bootstrap phase, this behaves the same as `use_all_dns_ips`.

Type:	<b>string</b>
Default:	use_all_dns_ips
Valid Values:	[use_all_dns_ips, resolve_canonical_bootstrap_servers_only]
Importance:	medium

- [\*\*client.id\*\*](#)

An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium

- [\*\*connections.max.idle.ms\*\*](#)

Close idle connections after the number of milliseconds specified by this config.

Type:	<b>long</b>
Default:	540000 (9 minutes)
Valid Values:	
Importance:	medium

- [\*\*delivery.timeout.ms\*\*](#)

An upper bound on the time to report success or failure after a call to `send()` returns. This limits the total time that a record will be delayed prior to sending, the time to await acknowledgement from the broker (if expected), and the time allowed for retriable send failures. The producer may report failure to send a record earlier than this config if either an unrecoverable error is encountered, the retries have been exhausted, or the record is added to a batch which reached an earlier delivery expiration deadline. The value of this config should be greater than or equal to the sum of `request.timeout.ms` and `linger.ms`.

Type:	<b>int</b>
Default:	120000 (2 minutes)
Valid Values:	[0,...]
Importance:	medium

- [\*\*linger.ms\*\*](#)

The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding a small amount of artificial delay—that is, rather than immediately sending out a record, the producer will wait for up to the given delay to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on

the delay for batching: once we get `batch.size` worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting `linger.ms=5`, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absence of load.

Type:	<b>long</b>
Default:	0
Valid Values:	[0,...]
Importance:	medium

- [\*\*max.block.ms\*\*](#)

The configuration controls how long the `KafkaProducer`'s `send()`, `partitionsFor()`, `initTransactions()`, `sendOffsetsToTransaction()`, `commitTransaction()` and `abortTransaction()` methods will block. For `send()` this timeout bounds the total time waiting for both metadata fetch and buffer allocation (blocking in the user-supplied serializers or partitioner is not counted against this timeout). For `partitionsFor()` this timeout bounds the time spent waiting for metadata if it is unavailable. The transaction-related methods always block, but may timeout if the transaction coordinator could not be discovered or did not respond within the timeout.

Type:	<b>long</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	medium

- [\*\*max.request.size\*\*](#)

The maximum size of a request in bytes. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests. This is also effectively a cap on the maximum uncompressed record batch size. Note that the server has its own cap on the record batch size (after compression if compression is enabled) which may be different from this.

Type:	<b>int</b>
Default:	1048576
Valid Values:	[0,...]
Importance:	medium

- [\*\*partitioner.class\*\*](#)

A class to use to determine which partition to be send to when produce the records. Available options are:

- If not set, the default partitioning logic is used. This strategy will try sticking to a partition until at least `batch.size` bytes is produced to the partition. It works with the strategy:
  - If no partition is specified but a key is present, choose a partition based on a hash of the key

- If no partition or key is present, choose the sticky partition that changes when at least `batch.size` bytes are produced to the partition.
- `org.apache.kafka.clients.producer.RoundRobinPartitioner`: This partitioning strategy is that each record in a series of consecutive records will be sent to a different partition (no matter if the 'key' is provided or not), until we run out of partitions and start over again. Note: There's a known issue that will cause uneven distribution when new batch is created. Please check KAFKA-9965 for more detail.

Implementing the `org.apache.kafka.clients.producer.Partitioner` interface allows you to plug in a custom partitioner.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*partitioner.ignore.keys\*\*](#)

When set to 'true' the producer won't use record keys to choose a partition. If 'false', producer would choose a partition based on a hash of the key when a key is present. Note: this setting has no effect if a custom partitioner is used.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

- [\*\*receive.buffer.bytes\*\*](#)

The size of the TCP receive buffer (SO\_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	32768 (32 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- [\*\*request.timeout.ms\*\*](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted. This should be larger than `replica.lag.time.max.ms` (a broker configuration) to reduce the possibility of message duplication due to unnecessary producer retries.

Type:	<b>int</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	medium

- **sasl.client.callback.handler.class**

The fully qualified name of a SASL client callback handler class that implements the AuthenticateCallbackHandler interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.jaas.config**

JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described [here](#). The format for the value is: `loginModuleClass controlFlag (optionName=optionValue)*;`. For brokers, the config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule` required;

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.kerberos.service.name**

The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.login.callback.handler.class**

The fully qualified name of a SASL login callback handler class that implements the AuthenticateCallbackHandler interface. For brokers, login callback handler config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.login.class**

The fully qualified name of a class that implements the Login interface. For brokers, login config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.mechanism**

SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.

Type:	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.jwks.endpoint.url**

The OAuth/OIDC provider URL from which the provider's [JWKS \(JSON Web Key Set\)](#) can be retrieved. The URL can be HTTP(S)-based or file-based. If the URL is HTTP(S)-based, the JWKS data will be retrieved from the OAuth/OIDC provider via the configured URL on broker startup. All then-current keys will be cached on the broker for incoming requests. If an authentication request is received for a JWT that includes a "kid" header claim value that isn't yet in the cache, the JWKS endpoint will be queried again on demand. However, the broker polls the URL every sasl.oauthbearer.jwks.endpoint.refresh.ms milliseconds to refresh the cache with any forthcoming keys before any JWT requests that include them are received. If the URL is file-based, the broker will load the JWKS file from a configured location on startup. In the event that the JWT includes a "kid" header value that isn't in the JWKS file, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.token.endpoint.url**

The URL for the OAuth/OIDC identity provider. If the URL is HTTP(S)-based, it is the issuer's token endpoint URL to which requests will be made to login based on the configuration in sasl.jaas.config. If the URL is file-based, it specifies a file containing an access token (in JWT serialized form) issued by the OAuth/OIDC identity provider to use for authorization.

Type:	string
Default:	null
Valid Values:	
Importance:	medium

- **security.protocol**

Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL.

Type:	string
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium

- **send.buffer.bytes**

The size of the TCP send buffer (SO\_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.

Type:	int
Default:	131072 (128 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- **socket.connection.setup.timeout.max.ms**

The maximum amount of time the client will wait for the socket connection to be established. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the timeout resulting in a random range between 20% below and 20% above the computed value.

Type:	long
Default:	30000 (30 seconds)
Valid Values:	
Importance:	medium

- [socket.connection.setup.timeout.ms](#)

The amount of time the client will wait for the socket connection to be established. If the connection is not built before the timeout elapses, clients will close the socket channel.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	medium

- [ssl.enabled.protocols](#)

The list of protocols enabled for SSL connections. The default is 'TLSv1.2,TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. With the default value for Java 11, clients and servers will prefer TLSv1.3 if both support it and fallback to TLSv1.2 otherwise (assuming both support at least TLSv1.2). This default should be fine for most cases. Also see the config documentation for `ssl.protocol`.

Type:	<b>list</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [ssl.keystore.type](#)

The file format of the key store file. This is optional for client. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [ssl.protocol](#)

The SSL protocol used to generate the SSLContext. The default is 'TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. This value should be fine for most use cases. Allowed values in recent JVMs are 'TLSv1.2' and 'TLSv1.3'. 'TLS', 'TLSv1.1', 'SSL', 'SSLv2' and 'SSLv3' may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. With the default value for this config and 'ssl.enabled.protocols', clients will downgrade to 'TLSv1.2' if the server does not support 'TLSv1.3'. If this config is set to 'TLSv1.2', clients will not use 'TLSv1.3' even if it is one of the values in `ssl.enabled.protocols` and the server only supports 'TLSv1.3'.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [\*\*ssl.provider\*\*](#)

The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*ssl.truststore.type\*\*](#)

The file format of the trust store file. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [\*\*acks\*\*](#)

The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are allowed:

- `acks=0` If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the `retries` configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to `-1`.
- `acks=1` This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.
- `acks=all` This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee. This is equivalent to the `acks=-1` setting.

Note that enabling idempotence requires this config value to be 'all'. If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled.

Type:	<b>string</b>
Default:	all
Valid Values:	[all, -1, 0, 1]
Importance:	low

- [auto.include.jmx.reporter](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [enable.idempotence](#)

When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream. If 'false', producer retries due to broker failures, etc., may write duplicates of the retried message in the stream. Note that enabling idempotence requires `max.in.flight.requests.per.connection` to be less than or equal to 5 (with message ordering preserved for any allowable value), `retries` to be greater than 0, and `acks` must be 'all'.

Idempotence is enabled by default if no conflicting configurations are set. If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled. If idempotence is explicitly enabled and conflicting configurations are set, a `ConfigException` is thrown.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [interceptor.classes](#)

A list of classes to use as interceptors. Implementing the `org.apache.kafka.clients.producer.ProducerInterceptor` interface allows you to intercept (and possibly mutate) the records received by the producer before they are published to the Kafka cluster. By default, there are no interceptors.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string
Importance:	low

- [max.in.flight.requests.per.connection](#)

The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this configuration is set to be greater than 1 and `enable.idempotence` is set to false, there is a risk of message reordering after a failed send due to retries (i.e., if retries are enabled); if retries are disabled or if `enable.idempotence` is set to true, ordering will be preserved. Additionally, enabling idempotence requires the value of this configuration to be less than or equal to 5. If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled.

Type:	<b>int</b>
Default:	5
Valid Values:	[1,...]
Importance:	low

- [\*\*metadata.max.age.ms\*\*](#)

The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	low

- [\*\*metadata.max.idle.ms\*\*](#)

Controls how long the producer will cache metadata for a topic that's idle. If the elapsed time since a topic was last produced to exceeds the metadata idle duration, then the topic's metadata is forgotten and the next access to it will force a metadata fetch request.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[5000,...]
Importance:	low

- [\*\*metric.reporters\*\*](#)

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string
Importance:	low

- [\*\*metrics.num.samples\*\*](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low

- [\*\*metrics.recording.level\*\*](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	[INFO, DEBUG, TRACE]
Importance:	low

- [\*\*metrics.sample.window.ms\*\*](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [\*\*partitioner.adaptive.partitioning.enable\*\*](#)

When set to 'true', the producer will try to adapt to broker performance and produce more messages to partitions hosted on faster brokers. If 'false', producer will try to distribute messages uniformly. Note: this setting has no effect if a custom partitioner is used

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [\*\*partitioner.availability.timeout.ms\*\*](#)

If a broker cannot process produce requests from a partition for `partitioner.availability.timeout.ms` time, the partitioner treats that partition as not available. If the value is 0, this logic is disabled. Note: this setting has no effect if a custom partitioner is used or `partitioner.adaptive.partitioning.enable` is set to 'false'

Type:	<b>long</b>
Default:	0
Valid Values:	[0,...]
Importance:	low

- [\*\*reconnect.backoff.max.ms\*\*](#)

The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.

Type:	<b>long</b>
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	low

- [\*\*reconnect.backoff.ms\*\*](#)

The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.

Type:	<b>long</b>
Default:	50
Valid Values:	[0,...]
Importance:	low

- [\*\*retry.backoff.ms\*\*](#)

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

Type:	<b>long</b>
Default:	100
Valid Values:	[0,...]
Importance:	low

- [\*\*sasl.kerberos.kinit.cmd\*\*](#)

Kerberos kinit command path.

Type:	<b>string</b>
Default:	/usr/bin/kinit
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.min.time.before.relogin\*\*](#)

Login thread sleep time between refresh attempts.

Type:	<b>long</b>
Default:	60000
Valid Values:	
Importance:	low

- [sasl.kerberos.ticket.renew.jitter](#)

Percentage of random jitter added to the renewal time.

Type:	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	low

- [sasl.kerberos.ticket.renew.window.factor](#)

Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	low

- [sasl.login.connect.timeout.ms](#)

The (optional) value in milliseconds for the external authentication provider connection timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.login.read.timeout.ms](#)

The (optional) value in milliseconds for the external authentication provider read timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.login.refresh.buffer.seconds](#)

The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and sasl.login.refresh.min.period.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	300
Valid Values:	[0,...,3600]
Importance:	low

- [sasl.login.refresh.min.period.seconds](#)

The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and sasl.login.refresh.buffer.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	60
Valid Values:	[0,...,900]
Importance:	low

- [sasl.login.refresh.window.factor](#)

Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.8
Valid Values:	[0.5,...,1.0]
Importance:	low

- [sasl.login.refresh.window.jitter](#)

The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.05
Valid Values:	[0.0,...,0.25]
Importance:	low

- [sasl.login.retry.backoff.max.ms](#)

The (optional) value in milliseconds for the maximum wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [\*\*sasl.login.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.clock.skew.seconds\*\*](#)

The (optional) value in seconds to allow for differences between the time of the OAuth/OIDC identity provider and the broker.

<b>Type:</b>	<b>int</b>
Default:	30
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.expected.audience\*\*](#)

The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth "aud" claim and if this value is set, the broker will match the value from JWT's "aud" claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

<b>Type:</b>	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.expected.issuer\*\*](#)

The (optional) setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth "iss" claim and if this value is set, the broker will match it exactly against what is in the JWT's "iss" claim. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.refresh.ms](#)**

The (optional) value in milliseconds for the broker to wait between refreshing its JWKS (JSON Web Key Set) cache that contains the keys to verify the signature of the JWT.

Type:	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms](#)**

The (optional) value in milliseconds for the maximum wait between attempts to retrieve the JWKS (JSON Web Key Set) from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.retry.backoff.ms](#)**

The (optional) value in milliseconds for the initial wait between JWKS (JSON Web Key Set) retrieval attempts from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.scope.claim.name](#)**

The OAuth claim for the scope is often named "scope", but this (optional) setting can provide a different name to use for the scope included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	scope
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.sub.claim.name](#)**

The OAuth claim for the subject is often named "sub", but this (optional) setting can provide a different name to use for the subject included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	sub
Valid Values:	
Importance:	low

- **[security.providers](#)**

A list of configurable creator classes each returning a provider implementing security algorithms. These classes should implement the `org.apache.kafka.common.security.auth.SecurityProviderCreator` interface.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.cipher.suites](#)**

A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.endpoint.identification.algorithm](#)**

The endpoint identification algorithm to validate server hostname using server certificate.

Type:	<b>string</b>
Default:	https
Valid Values:	
Importance:	low

- [ssl.engine.factory.class](#)

The class of type org.apache.kafka.common.security.auth.SslEngineFactory to provide SSLEngine objects. Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [ssl.keymanager.algorithm](#)

The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.

Type:	<b>string</b>
Default:	SunX509
Valid Values:	
Importance:	low

- [ssl.secure.random.implementation](#)

The SecureRandom PRNG implementation to use for SSL cryptography operations.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [ssl.trustmanager.algorithm](#)

The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

Type:	<b>string</b>
Default:	PKIX
Valid Values:	
Importance:	low

- [transaction.timeout.ms](#)

The maximum amount of time in ms that the transaction coordinator will wait for a transaction status update from the producer before proactively aborting the ongoing transaction. If this value is larger than the transaction.max.timeout.ms setting in the broker, the request will fail with a `InvalidTxnTimeoutException` error.

<b>Type:</b>	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	
Importance:	low

- [\*\*transactional.id\*\*](#)

The TransactionalId to use for transactional delivery. This enables reliability semantics which span multiple producer sessions since it allows the client to guarantee that transactions using the same TransactionalId have been completed prior to starting any new transactions. If no TransactionalId is provided, then the producer is limited to idempotent delivery. If a TransactionalId is configured, `enable.idempotence` is implied. By default the TransactionId is not configured, which means transactions cannot be used. Note that, by default, transactions require a cluster of at least three brokers which is the recommended setting for production; for development you can change this, by adjusting broker setting `transaction.state.log.replication.factor`.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	non-empty string
Importance:	low

## [\*\*3.4 Consumer Configs\*\*](#)

Below is the configuration for the consumer:

- [\*\*key.deserializer\*\*](#)

Deserializer class for key that implements the `org.apache.kafka.common.serialization.Deserializer` interface.

<b>Type:</b>	<b>class</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*value.deserializer\*\*](#)

Deserializer class for value that implements the `org.apache.kafka.common.serialization.Deserializer` interface.

<b>Type:</b>	<b>class</b>
Default:	
Valid Values:	
Importance:	high

- **[bootstrap.servers](#)**

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

Type:	list
Default:	""
Valid Values:	non-null string
Importance:	high

- **[fetch.min.bytes](#)**

The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as a single byte of data is available or the fetch request times out waiting for data to arrive. Setting this to something greater than 1 will cause the server to wait for larger amounts of data to accumulate which can improve server throughput a bit at the cost of some additional latency.

Type:	int
Default:	1
Valid Values:	[0,...]
Importance:	high

- **[group.id](#)**

A unique string that identifies the consumer group this consumer belongs to. This property is required if the consumer uses either the group management functionality by using `subscribe(topic)` or the Kafka-based offset management strategy.

Type:	string
Default:	null
Valid Values:	
Importance:	high

- **[heartbeat.interval.ms](#)**

The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.

Type:	<b>int</b>
Default:	3000 (3 seconds)
Valid Values:	
Importance:	high

- [\*\*max.partition.fetch.bytes\*\*](#)

The maximum amount of data per-partition the server will return. Records are fetched in batches by the consumer. If the first record batch in the first non-empty partition of the fetch is larger than this limit, the batch will still be returned to ensure that the consumer can make progress. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config). See `fetch.max.bytes` for limiting the consumer request size.

Type:	<b>int</b>
Default:	1048576 (1 mebibyte)
Valid Values:	[0,...]
Importance:	high

- [\*\*session.timeout.ms\*\*](#)

The timeout used to detect client failures when using Kafka's group management facility. The client sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this client from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.

Type:	<b>int</b>
Default:	45000 (45 seconds)
Valid Values:	
Importance:	high

- [\*\*ssl.key.password\*\*](#)

The password of the private key in the key store file or the PEM key specified in 'ssl.keystore.key'.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.certificate.chain\*\*](#)

Certificate chain in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with a list of X.509 certificates

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.key\*\*](#)

Private key in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with PKCS#8 keys. If the key is encrypted, key password must be specified using 'ssl.key.password'

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.location\*\*](#)

The location of the key store file. This is optional for client and can be used for two-way authentication for client.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.password\*\*](#)

The store password for the key store file. This is optional for client and only needed if 'ssl.keystore.location' is configured. Key store password is not supported for PEM format.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.certificates\*\*](#)

Trusted certificates in the format specified by 'ssl.truststore.type'. Default SSL engine factory supports only PEM format with X.509 certificates.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.location\*\*](#)

The location of the trust store file.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.password\*\*](#)

The password for the trust store file. If a password is not set, trust store file configured will still be used, but integrity checking is disabled. Trust store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*allow.auto.create.topics\*\*](#)

Allow automatic topic creation on the broker when subscribing to or assigning a topic. A topic being subscribed to will be automatically created only if the broker allows for it using `auto.create.topics.enable` broker configuration. This configuration must be set to `false` when using brokers older than 0.11.0

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium

- [\*\*auto.offset.reset\*\*](#)

What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):

- earliest: automatically reset the offset to the earliest offset
- latest: automatically reset the offset to the latest offset
- none: throw exception to the consumer if no previous offset is found for the consumer's group
- anything else: throw exception to the consumer.

Type:	<b>string</b>
Default:	latest
Valid Values:	[latest, earliest, none]
Importance:	medium

- [\*\*client.dns.lookup\*\*](#)

Controls how the client uses DNS lookups. If set to `use_all_dns_ips`, connect to each returned IP address in sequence until a successful connection is established. After a disconnection, the next IP is used. Once all IPs have been used once, the client resolves the IP(s) from the hostname again (both the JVM and the OS cache DNS name lookups, however). If set to `resolve_canonical_bootstrap_servers_only`, resolve each bootstrap address into a list of canonical names. After the bootstrap phase, this behaves the same as `use_all_dns_ips`.

<b>Type:</b>	<b>string</b>
Default:	<code>use_all_dns_ips</code>
Valid Values:	[ <code>use_all_dns_ips</code> , <code>resolve_canonical_bootstrap_servers_only</code> ]
Importance:	medium

- [\*\*connections.max.idle.ms\*\*](#)

Close idle connections after the number of milliseconds specified by this config.

<b>Type:</b>	<b>long</b>
Default:	540000 (9 minutes)
Valid Values:	
Importance:	medium

- [\*\*default.api.timeout.ms\*\*](#)

Specifies the timeout (in milliseconds) for client APIs. This configuration is used as the default timeout for all client operations that do not specify a `timeout` parameter.

<b>Type:</b>	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	medium

- [\*\*enable.auto.commit\*\*](#)

If true the consumer's offset will be periodically committed in the background.

<b>Type:</b>	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium

- [\*\*exclude.internal.topics\*\*](#)

Whether internal topics matching a subscribed pattern should be excluded from the subscription. It is always possible to explicitly subscribe to an internal topic.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	medium

- [\*\*fetch.max.bytes\*\*](#)

The maximum amount of data the server should return for a fetch request. Records are fetched in batches by the consumer, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that the consumer can make progress. As such, this is not an absolute maximum. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config).

Note that the consumer performs multiple fetches in parallel.

Type:	<b>int</b>
Default:	52428800 (50 mebibytes)
Valid Values:	[0,...]
Importance:	medium

- [\*\*group.instance.id\*\*](#)

A unique identifier of the consumer instance provided by the end user. Only non-empty strings are permitted. If set, the consumer is treated as a static member, which means that only one instance with this ID is allowed in the consumer group at any time. This can be used in combination with a larger session timeout to avoid group rebalances caused by transient unavailability (e.g. process restarts). If not set, the consumer will join the group as a dynamic member, which is the traditional behavior.

Type:	<b>string</b>
Default:	null
Valid Values:	non-empty string
Importance:	medium

- [\*\*isolation.level\*\*](#)

Controls how to read messages written transactionally. If set to `read_committed`, `consumer.poll()` will only return transactional messages which have been committed. If set to `read_uncommitted` (the default), `consumer.poll()` will return all messages, even transactional messages which have been aborted. Non-transactional messages will be returned unconditionally in either mode.

Messages will always be returned in offset order. Hence, in `read_committed` mode, `consumer.poll()` will only return messages up to the last stable offset (LSO), which is the one less than the offset of the first open transaction. In particular any messages appearing after messages belonging to ongoing transactions will be withheld until the relevant transaction has been completed. As a result, `read_committed` consumers will not be able to read up to the high watermark when there are in flight transactions.

Further, when in `read_committed` the `seekToEnd` method will return the LSO

<b>Type:</b>	<b>string</b>
Default:	read_uncommitted
Valid Values:	[read_committed, read_uncommitted]
Importance:	medium

- [\*\*max.poll.interval.ms\*\*](#)

The maximum delay between invocations of poll() when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If poll() is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member. For consumers using a non-null `group.instance.id` which reach this timeout, partitions will not be immediately reassigned. Instead, the consumer will stop sending heartbeats and partitions will be reassigned after expiration of `session.timeout.ms`. This mirrors the behavior of a static consumer which has shutdown.

<b>Type:</b>	<b>int</b>
Default:	300000 (5 minutes)
Valid Values:	[1,...]
Importance:	medium

- [\*\*max.poll.records\*\*](#)

The maximum number of records returned in a single call to poll(). Note, that `max.poll.records` does not impact the underlying fetching behavior. The consumer will cache the records from each fetch request and returns them incrementally from each poll.

<b>Type:</b>	<b>int</b>
Default:	500
Valid Values:	[1,...]
Importance:	medium

- [\*\*partition.assignment.strategy\*\*](#)

A list of class names or class types, ordered by preference, of supported partition assignment strategies that the client will use to distribute partition ownership amongst consumer instances when group management is used. Available options are:

- `org.apache.kafka.clients.consumer.RangeAssignor`: Assigns partitions on a per-topic basis.
- `org.apache.kafka.clients.consumer.RoundRobinAssignor`: Assigns partitions to consumers in a round-robin fashion.
- `org.apache.kafka.clients.consumer.StickyAssignor`: Guarantees an assignment that is maximally balanced while preserving as many existing partition assignments as possible.
- `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`: Follows the same StickyAssignor logic, but allows for cooperative rebalancing.

The default assignor is [RangeAssignor, CooperativeStickyAssignor], which will use the RangeAssignor by default, but allows upgrading to the CooperativeStickyAssignor with just a single rolling bounce that removes the RangeAssignor from the list.

Implementing the `org.apache.kafka.clients.consumer.ConsumerPartitionAssignor` interface allows you to plug in a custom assignment strategy.

<b>Type:</b>	<code>list</code>
Default:	class org.apache.kafka.clients.consumer.RangeAssignor, class org.apache.kafka.clients.consumer.CooperativeStickyAssignor
Valid Values:	non-null string
Importance:	medium

- [\*\*receive.buffer.bytes\*\*](#)

The size of the TCP receive buffer (SO\_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.

<b>Type:</b>	<code>int</code>
Default:	65536 (64 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- [\*\*request.timeout.ms\*\*](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

<b>Type:</b>	<code>int</code>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	medium

- [\*\*sasl.client.callback.handler.class\*\*](#)

The fully qualified name of a SASL client callback handler class that implements the `AuthenticateCallbackHandler` interface.

<b>Type:</b>	<code>class</code>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.jaas.config\*\*](#)

JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described [here](#). The format for the value is: `loginModuleClass controlFlag (optionName=optionValue)*;`. For brokers, the config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-`

256.sasl.jaas.config=com.example.ScramLoginModule required;

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.kerberos.service.name\*\*](#)

The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.login.callback.handler.class\*\*](#)

The fully qualified name of a SASL login callback handler class that implements the AuthenticateCallbackHandler interface. For brokers, login callback handler config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.scram-sha-

256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.login.class\*\*](#)

The fully qualified name of a class that implements the Login interface. For brokers, login config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, listener.name.sasl\_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.mechanism\*\*](#)

SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.

Type:	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium

- **[sasl.oauthbearer.jwks.endpoint.url](#)**

The OAuth/OIDC provider URL from which the provider's [JWKS \(JSON Web Key Set\)](#) can be retrieved. The URL can be HTTP(S)-based or file-based. If the URL is HTTP(S)-based, the JWKS data will be retrieved from the OAuth/OIDC provider via the configured URL on broker startup. All then-current keys will be cached on the broker for incoming requests. If an authentication request is received for a JWT that includes a "kid" header claim value that isn't yet in the cache, the JWKS endpoint will be queried again on demand. However, the broker polls the URL every `sasl.oauthbearer.jwks.endpoint.refresh.ms` milliseconds to refresh the cache with any forthcoming keys before any JWT requests that include them are received. If the URL is file-based, the broker will load the JWKS file from a configured location on startup. In the event that the JWT includes a "kid" header value that isn't in the JWKS file, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **[sasl.oauthbearer.token.endpoint.url](#)**

The URL for the OAuth/OIDC identity provider. If the URL is HTTP(S)-based, it is the issuer's token endpoint URL to which requests will be made to login based on the configuration in `sasl.jaas.config`. If the URL is file-based, it specifies a file containing an access token (in JWT serialized form) issued by the OAuth/OIDC identity provider to use for authorization.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **[security.protocol](#)**

Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL.

Type:	<b>string</b>
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium

- [\*\*send.buffer.bytes\*\*](#)

The size of the TCP send buffer (SO\_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.

Type:	int
Default:	131072 (128 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- [\*\*socket.connection.setup.timeout.max.ms\*\*](#)

The maximum amount of time the client will wait for the socket connection to be established. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the timeout resulting in a random range between 20% below and 20% above the computed value.

Type:	long
Default:	30000 (30 seconds)
Valid Values:	
Importance:	medium

- [\*\*socket.connection.setup.timeout.ms\*\*](#)

The amount of time the client will wait for the socket connection to be established. If the connection is not built before the timeout elapses, clients will close the socket channel.

Type:	long
Default:	10000 (10 seconds)
Valid Values:	
Importance:	medium

- [\*\*ssl.enabled.protocols\*\*](#)

The list of protocols enabled for SSL connections. The default is 'TLSv1.2,TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. With the default value for Java 11, clients and servers will prefer TLSv1.3 if both support it and fallback to TLSv1.2 otherwise (assuming both support at least TLSv1.2). This default should be fine for most cases. Also see the config documentation for `ssl.protocol`.

Type:	list
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [\*\*ssl.keystore.type\*\*](#)

The file format of the key store file. This is optional for client. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [\*\*ssl.protocol\*\*](#)

The SSL protocol used to generate the SSLContext. The default is 'TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. This value should be fine for most use cases. Allowed values in recent JVMs are 'TLSv1.2' and 'TLSv1.3'. 'TLS', 'TLSv1.1', 'SSL', 'SSLv2' and 'SSLv3' may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. With the default value for this config and 'ssl.enabled.protocols', clients will downgrade to 'TLSv1.2' if the server does not support 'TLSv1.3'. If this config is set to 'TLSv1.2', clients will not use 'TLSv1.3' even if it is one of the values in `ssl.enabled.protocols` and the server only supports 'TLSv1.3'.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [\*\*ssl.provider\*\*](#)

The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*ssl.truststore.type\*\*](#)

The file format of the trust store file. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [auto.commit.interval.ms](#)

The frequency in milliseconds that the consumer offsets are auto-committed to Kafka if `enable.auto.commit` is set to `true`.

Type:	<b>int</b>
Default:	5000 (5 seconds)
Valid Values:	[0,...]
Importance:	low

- [auto.include.jmx.reporter](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [check.crcs](#)

Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [client.id](#)

An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [client.rack](#)

A rack identifier for this client. This can be any string value which indicates where this client is physically located. It corresponds with the broker config 'broker.rack'

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [fetch.max.wait.ms](#)

The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by `fetch.min.bytes`.

Type:	<b>int</b>
Default:	500
Valid Values:	[0,...]
Importance:	low

- [interceptor.classes](#)

A list of classes to use as interceptors. Implementing the `org.apache.kafka.clients.consumer.ConsumerInterceptor` interface allows you to intercept (and possibly mutate) records received by the consumer. By default, there are no interceptors.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string
Importance:	low

- [metadata.max.age.ms](#)

The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	low

- [metric.reporters](#)

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string
Importance:	low

- [\*\*metrics.num.samples\*\*](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low

- [\*\*metrics.recording.level\*\*](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	[INFO, DEBUG, TRACE]
Importance:	low

- [\*\*metrics.sample.window.ms\*\*](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [\*\*reconnect.backoff.max.ms\*\*](#)

The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.

Type:	<b>long</b>
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	low

- [reconnect.backoff.ms](#)

The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.

Type:	<b>long</b>
Default:	50
Valid Values:	[0,...]
Importance:	low

- [retry.backoff.ms](#)

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

Type:	<b>long</b>
Default:	100
Valid Values:	[0,...]
Importance:	low

- [sasl.kerberos.kinit.cmd](#)

Kerberos kinit command path.

Type:	<b>string</b>
Default:	/usr/bin/kinit
Valid Values:	
Importance:	low

- [sasl.kerberos.min.time.before.relogin](#)

Login thread sleep time between refresh attempts.

Type:	<b>long</b>
Default:	60000
Valid Values:	
Importance:	low

- [sasl.kerberos.ticket.renew.jitter](#)

Percentage of random jitter added to the renewal time.

Type:	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.ticket.renew.window.factor\*\*](#)

Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	low

- [\*\*sasl.login.connect.timeout.ms\*\*](#)

The (optional) value in milliseconds for the external authentication provider connection timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.login.read.timeout.ms\*\*](#)

The (optional) value in milliseconds for the external authentication provider read timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.login.refresh.buffer.seconds\*\*](#)

The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and sasl.login.refresh.min.period.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	300
Valid Values:	[0,...,3600]
Importance:	low

- [\*\*sasl.login.refresh.min.period.seconds\*\*](#)

The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and sasl.login.refresh.buffer.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	60
Valid Values:	[0,...,900]
Importance:	low

- [\*\*sasl.login.refresh.window.factor\*\*](#)

Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.8
Valid Values:	[0.5,...,1.0]
Importance:	low

- [\*\*sasl.login.refresh.window.jitter\*\*](#)

The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.05
Valid Values:	[0.0,...,0.25]
Importance:	low

- [\*\*sasl.login.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [sasl.login.retry.backoff.ms](#)

The (optional) value in milliseconds for the initial wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [sasl.oauthbearer.clock.skew.seconds](#)

The (optional) value in seconds to allow for differences between the time of the OAuth/OIDC identity provider and the broker.

Type:	<b>int</b>
Default:	30
Valid Values:	
Importance:	low

- [sasl.oauthbearer.expected.audience](#)

The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth "aud" claim and if this value is set, the broker will match the value from JWT's "aud" claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.oauthbearer.expected.issuer](#)

The (optional) setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth "iss" claim and if this value is set, the broker will match it exactly against what is in the JWT's "iss" claim. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.refresh.ms\*\*](#)

The (optional) value in milliseconds for the broker to wait between refreshing its JWKS (JSON Web Key Set) cache that contains the keys to verify the signature of the JWT.

<b>Type:</b>	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between attempts to retrieve the JWKS (JSON Web Key Set) from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between JWKS (JSON Web Key Set) retrieval attempts from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.scope.claim.name\*\*](#)

The OAuth claim for the scope is often named "scope", but this (optional) setting can provide a different name to use for the scope included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

<b>Type:</b>	<b>string</b>
Default:	scope
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.sub.claim.name\*\*](#)

The OAuth claim for the subject is often named "sub", but this (optional) setting can provide a different name to use for the subject included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	sub
Valid Values:	
Importance:	low

- [\*\*security.providers\*\*](#)

A list of configurable creator classes each returning a provider implementing security algorithms. These classes should implement the

`org.apache.kafka.common.security.auth.SecurityProviderCreator` interface.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.cipher.suites\*\*](#)

A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.endpoint.identification.algorithm\*\*](#)

The endpoint identification algorithm to validate server hostname using server certificate.

Type:	<b>string</b>
Default:	https
Valid Values:	
Importance:	low

- [\*\*ssl.engine.factory.class\*\*](#)

The class of type `org.apache.kafka.common.security.auth.SslEngineFactory` to provide SSLEngine objects. Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.keymanager.algorithm](#)**

The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.

Type:	<b>string</b>
Default:	SunX509
Valid Values:	
Importance:	low

- **[ssl.secure.random.implementation](#)**

The SecureRandom PRNG implementation to use for SSL cryptography operations.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.trustmanager.algorithm](#)**

The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

Type:	<b>string</b>
Default:	PKIX
Valid Values:	
Importance:	low

## [3.5 Kafka Connect Configs](#)

Below is the configuration of the Kafka Connect framework.

- **[config.storage.topic](#)**

The name of the Kafka topic where connector configurations are stored

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*group.id\*\*](#)

A unique string that identifies the Connect cluster group this worker belongs to.

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*key.converter\*\*](#)

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	<b>class</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*offset.storage.topic\*\*](#)

The name of the Kafka topic where source connector offsets are stored

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*status.storage.topic\*\*](#)

The name of the Kafka topic where connector and task status are stored

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*value.converter\*\*](#)

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	class
Default:	
Valid Values:	
Importance:	high

- [\*\*bootstrap.servers\*\*](#)

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

Type:	list
Default:	localhost:9092
Valid Values:	
Importance:	high

- [\*\*exactly.once.source.support\*\*](#)

Whether to enable exactly-once support for source connectors in the cluster by using transactions to write source records and their source offsets, and by proactively fencing out old task generations before bringing up new ones.

To enable exactly-once source support on a new cluster, set this property to 'enabled'. To enable support on an existing cluster, first set to 'preparing' on every worker in the cluster, then set to 'enabled'. A rolling upgrade may be used for both changes. For more information on this feature, see the [exactly-once source support documentation](#).

Type:	string
Default:	disabled
Valid Values:	(case insensitive) [DISABLED, ENABLED, PREPARING]
Importance:	high

- [\*\*heartbeat.interval.ms\*\*](#)

The expected time between heartbeats to the group coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the worker's session stays active and to facilitate rebalancing when new members join or leave the group. The value must be set lower than `session.timeout.ms`, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.

Type:	int
Default:	3000 (3 seconds)
Valid Values:	
Importance:	high

- [rebalance.timeout.ms](#)

The maximum allowed time for each worker to join the group once a rebalance has begun. This is basically a limit on the amount of time needed for all tasks to flush any pending data and commit offsets. If the timeout is exceeded, then the worker will be removed from the group, which will cause offset commit failures.

Type:	int
Default:	60000 (1 minute)
Valid Values:	
Importance:	high

- [session.timeout.ms](#)

The timeout used to detect worker failures. The worker sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove the worker from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by `group.min.session.timeout.ms` and `group.max.session.timeout.ms`.

Type:	int
Default:	10000 (10 seconds)
Valid Values:	
Importance:	high

- [ssl.key.password](#)

The password of the private key in the key store file or the PEM key specified in 'ssl.keystore.key'.

Type:	password
Default:	null
Valid Values:	
Importance:	high

- [ssl.keystore.certificate.chain](#)

Certificate chain in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with a list of X.509 certificates

Type:	password
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.key\*\*](#)

Private key in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with PKCS#8 keys. If the key is encrypted, key password must be specified using 'ssl.key.password'

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.location\*\*](#)

The location of the key store file. This is optional for client and can be used for two-way authentication for client.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.password\*\*](#)

The store password for the key store file. This is optional for client and only needed if 'ssl.keystore.location' is configured. Key store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.certificates\*\*](#)

Trusted certificates in the format specified by 'ssl.truststore.type'. Default SSL engine factory supports only PEM format with X.509 certificates.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.location\*\*](#)

The location of the trust store file.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.password\*\*](#)

The password for the trust store file. If a password is not set, trust store file configured will still be used, but integrity checking is disabled. Trust store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*client.dns.lookup\*\*](#)

Controls how the client uses DNS lookups. If set to `use_all_dns_ips`, connect to each returned IP address in sequence until a successful connection is established. After a disconnection, the next IP is used. Once all IPs have been used once, the client resolves the IP(s) from the hostname again (both the JVM and the OS cache DNS name lookups, however). If set to `resolve_canonical_bootstrap_servers_only`, resolve each bootstrap address into a list of canonical names. After the bootstrap phase, this behaves the same as `use_all_dns_ips`.

Type:	<b>string</b>
Default:	<code>use_all_dns_ips</code>
Valid Values:	[ <code>use_all_dns_ips</code> , <code>resolve_canonical_bootstrap_servers_only</code> ]
Importance:	medium

- [\*\*connections.max.idle.ms\*\*](#)

Close idle connections after the number of milliseconds specified by this config.

Type:	<b>long</b>
Default:	540000 (9 minutes)
Valid Values:	
Importance:	medium

- [\*\*connector.client.config.override.policy\*\*](#)

Class name or alias of implementation of `ConnectorClientConfigOverridePolicy`. Defines what client configurations can be overridden by the connector. The default implementation is `All`, meaning connector configurations can override all client properties. The other possible policies in the framework include `None` to disallow connectors from overriding client properties, and `Principal` to allow connectors to override only client principals.

Type:	<b>string</b>
Default:	All
Valid Values:	
Importance:	medium

- **[receive.buffer.bytes](#)**

The size of the TCP receive buffer (SO\_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	32768 (32 kibibytes)
Valid Values:	[0,...]
Importance:	medium

- **[request.timeout.ms](#)**

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

Type:	<b>int</b>
Default:	40000 (40 seconds)
Valid Values:	[0,...]
Importance:	medium

- **[sasl.client.callback.handler.class](#)**

The fully qualified name of a SASL client callback handler class that implements the AuthenticateCallbackHandler interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **[sasl.jaas.config](#)**

JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described [here](#). The format for the value is: `loginModuleClass controlFlag (optionName=optionValue)*;`. For brokers, the config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule required;`

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.kerberos.service.name\*\*](#)

The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.login.callback.handler.class\*\*](#)

The fully qualified name of a SASL login callback handler class that implements the AuthenticateCallbackHandler interface. For brokers, login callback handler config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example,  
 listener.name.sasl\_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.login.class\*\*](#)

The fully qualified name of a class that implements the Login interface. For brokers, login config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example,  
 listener.name.sasl\_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*sasl.mechanism\*\*](#)

SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.

Type:	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.jwks.endpoint.url**

The OAuth/OIDC provider URL from which the provider's [JWKS \(JSON Web Key Set\)](#) can be retrieved. The URL can be HTTP(S)-based or file-based. If the URL is HTTP(S)-based, the JWKS data will be retrieved from the OAuth/OIDC provider via the configured URL on broker startup. All then-current keys will be cached on the broker for incoming requests. If an authentication request is received for a JWT that includes a "kid" header claim value that isn't yet in the cache, the JWKS endpoint will be queried again on demand. However, the broker polls the URL every `sasl.oauthbearer.jwks.endpoint.refresh.ms` milliseconds to refresh the cache with any forthcoming keys before any JWT requests that include them are received. If the URL is file-based, the broker will load the JWKS file from a configured location on startup. In the event that the JWT includes a "kid" header value that isn't in the JWKS file, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.token.endpoint.url**

The URL for the OAuth/OIDC identity provider. If the URL is HTTP(S)-based, it is the issuer's token endpoint URL to which requests will be made to login based on the configuration in `sasl.jaas.config`. If the URL is file-based, it specifies a file containing an access token (in JWT serialized form) issued by the OAuth/OIDC identity provider to use for authorization.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **security.protocol**

Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL.

Type:	<b>string</b>
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium

- [send.buffer.bytes](#)

The size of the TCP send buffer (SO\_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	131072 (128 kibibytes)
Valid Values:	[0,...]
Importance:	medium

- [ssl.enabled.protocols](#)

The list of protocols enabled for SSL connections. The default is 'TLSv1.2,TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. With the default value for Java 11, clients and servers will prefer TLSv1.3 if both support it and fallback to TLSv1.2 otherwise (assuming both support at least TLSv1.2). This default should be fine for most cases. Also see the config documentation for `ssl.protocol`.

Type:	<b>list</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [ssl.keystore.type](#)

The file format of the key store file. This is optional for client. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [ssl.protocol](#)

The SSL protocol used to generate the SSLContext. The default is 'TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. This value should be fine for most use cases. Allowed values in recent JVMs are 'TLSv1.2' and 'TLSv1.3'. 'TLS', 'TLSv1.1', 'SSL', 'SSLv2' and 'SSLv3' may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. With the default value for this config and 'ssl.enabled.protocols', clients will downgrade to 'TLSv1.2' if the server does not support 'TLSv1.3'. If this config is set to 'TLSv1.2', clients will not use 'TLSv1.3' even if it is one of the values in `ssl.enabled.protocols` and the server only supports 'TLSv1.3'.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [ssl.provider](#)

The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [ssl.truststore.type](#)

The file format of the trust store file. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [worker.sync.timeout.ms](#)

When the worker is out of sync with other workers and needs to resynchronize configurations, wait up to this amount of time before giving up, leaving the group, and waiting a backoff period before rejoining.

Type:	<b>int</b>
Default:	3000 (3 seconds)
Valid Values:	
Importance:	medium

- [worker.unsync.backoff.ms](#)

When the worker is out of sync with other workers and fails to catch up within `worker.sync.timeout.ms`, leave the Connect cluster for this long before rejoining.

Type:	<b>int</b>
Default:	300000 (5 minutes)
Valid Values:	
Importance:	medium

- [access.control.allow.methods](#)

Sets the methods supported for cross origin requests by setting the Access-Control-Allow-Methods header. The default value of the Access-Control-Allow-Methods header allows cross origin requests for GET, POST and HEAD.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*access.control.allow.origin\*\*](#)

Value to set the Access-Control-Allow-Origin header to for REST API requests. To enable cross origin access, set this to the domain of the application that should be permitted to access the API, or '\*' to allow access from any domain. The default value only allows access from the domain of the REST API.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*admin.listeners\*\*](#)

List of comma-separated URIs the Admin REST API will listen on. The supported protocols are HTTP and HTTPS. An empty or blank string will disable this feature. The default behavior is to use the regular listener (specified by the 'listeners' property).

Type:	<b>list</b>
Default:	null
Valid Values:	List of comma-separated URLs, ex: <a href="http://localhost:8080">http://localhost:8080</a> , <a href="https://localhost:8443">https://localhost:8443</a> .
Importance:	low

- [\*\*auto.include.jmx.reporter\*\*](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [\*\*client.id\*\*](#)

An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*config.providers\*\*](#)

Comma-separated names of `ConfigProvider` classes, loaded and used in the order specified. Implementing the interface `ConfigProvider` allows you to replace variable references in connector configurations, such as for externalized secrets.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*config.storage.replication.factor\*\*](#)

Replication factor used when creating the configuration storage topic

Type:	<b>short</b>
Default:	3
Valid Values:	Positive number not larger than the number of brokers in the Kafka cluster, or -1 to use the broker's default
Importance:	low

- [\*\*connect.protocol\*\*](#)

Compatibility mode for Kafka Connect Protocol

Type:	<b>string</b>
Default:	sessioned
Valid Values:	[eager, compatible, sessioned]
Importance:	low

- [\*\*header.converter\*\*](#)

HeaderConverter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the header values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro. By default, the SimpleHeaderConverter is used to serialize header values to strings and deserialize them by inferring the schemas.

<b>Type:</b>	<b>class</b>
Default:	org.apache.kafka.connect.storage.SimpleHeaderConverter
Valid Values:	
Importance:	low

- [\*\*inter.worker.key.generation.algorithm\*\*](#)

The algorithm to use for generating internal request keys. The algorithm 'HmacSHA256' will be used as a default on JVMs that support it; on other JVMs, no default is used and a value for this property must be manually specified in the worker config.

<b>Type:</b>	<b>string</b>
Default:	HmacSHA256
Valid Values:	Any KeyGenerator algorithm supported by the worker JVM
Importance:	low

- [\*\*inter.worker.key.size\*\*](#)

The size of the key to use for signing internal requests, in bits. If null, the default key size for the key generation algorithm will be used.

	<b>Type:</b>	<b>int</b>
	Default:	null
	Valid Values:	
	Importance:	low

- [\*\*inter.worker.key.ttl.ms\*\*](#)

The TTL of generated session keys used for internal request validation (in milliseconds)

	<b>Type:</b>	<b>int</b>
	Default:	3600000 (1 hour)
	Valid Values:	[0,...,2147483647]
	Importance:	low

- [\*\*inter.worker.signature.algorithm\*\*](#)

The algorithm used to sign internal requestsThe algorithm 'inter.worker.signature.algorithm' will be used as a default on JVMs that support it; on other JVMs, no default is used and a value for this property must be manually specified in the worker config.

<b>Type:</b>	<b>string</b>
Default:	HmacSHA256
Valid Values:	Any MAC algorithm supported by the worker JVM
Importance:	low

- **inter.worker.verification.algorithms**

A list of permitted algorithms for verifying internal requests, which must include the algorithm used for the inter.worker.signature.algorithm property. The algorithm(s) '[HmacSHA256]' will be used as a default on JVMs that provide them; on other JVMs, no default is used and a value for this property must be manually specified in the worker config.

<b>Type:</b>	list
Default:	HmacSHA256
Valid Values:	A list of one or more MAC algorithms, each supported by the worker JVM
Importance:	low

- **listeners**

List of comma-separated URIs the REST API will listen on. The supported protocols are HTTP and HTTPS.

Specify hostname as 0.0.0.0 to bind to all interfaces.

Leave hostname empty to bind to default interface.

Examples of legal listener lists: <HTTP://myhost:8083>,<HTTPS://myhost:8084>

<b>Type:</b>	list
Default:	<a href="http://:8083">http://:8083</a>
Valid Values:	List of comma-separated URLs, ex: <a href="http://localhost:8080">http://localhost:8080</a> , <a href="https://localhost:8443">https://localhost:8443</a> .
Importance:	low

- **metadata.max.age.ms**

The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

<b>Type:</b>	long
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	low

- **metric.reporters**

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

<b>Type:</b>	list
Default:	""
Valid Values:	
Importance:	low

- [\*\*metrics.num.samples\*\*](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low

- [\*\*metrics.recording.level\*\*](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	[INFO, DEBUG]
Importance:	low

- [\*\*metrics.sample.window.ms\*\*](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [\*\*offset.flush.interval.ms\*\*](#)

Interval at which to try committing offsets for tasks.

Type:	<b>long</b>
Default:	60000 (1 minute)
Valid Values:	
Importance:	low

- [\*\*offset.flush.timeout.ms\*\*](#)

Maximum number of milliseconds to wait for records to flush and partition offset data to be committed to offset storage before cancelling the process and restoring the offset data to be committed in a future attempt. This property has no effect for source connectors running with exactly-once support.

Type:	<b>long</b>
Default:	5000 (5 seconds)
Valid Values:	
Importance:	low

- [offset.storage.partitions](#)

The number of partitions used when creating the offset storage topic

<b>Type:</b>	<b>int</b>
Default:	25
Valid Values:	Positive number, or -1 to use the broker's default
Importance:	low

- [offset.storage.replication.factor](#)

Replication factor used when creating the offset storage topic

<b>Type:</b>	<b>short</b>
Default:	3
Valid Values:	Positive number not larger than the number of brokers in the Kafka cluster, or -1 to use the broker's default
Importance:	low

- [plugin.path](#)

List of paths separated by commas (,) that contain plugins (connectors, converters, transformations). The list should consist of top level directories that include any combination of:  
a) directories immediately containing jars with plugins and their dependencies  
b) uber-jars with plugins and their dependencies  
c) directories immediately containing the package directory structure of classes of plugins and their dependencies

Note: symlinks will be followed to discover dependencies or plugins.

Examples: plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/connectors

Do not use config provider variables in this property, since the raw path is used by the worker's scanner before config providers are initialized and used to replace variables.

<b>Type:</b>	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [reconnect.backoff.max.ms](#)

The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.

<b>Type:</b>	<b>long</b>
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	low

- [reconnect.backoff.ms](#)

The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.

	<b>Type:</b>	<b>long</b>
	Default:	50
	Valid Values:	[0,...]
	Importance:	low

- [response.http.headers.config](#)

Rules for REST API HTTP response headers

	<b>Type:</b>	<b>string</b>
	Default:	""
	Valid Values:	Comma-separated header rules, where each header rule is of the form '[action] [header name]:[header value]' and optionally surrounded by double quotes if any part of a header rule contains a comma
	Importance:	low

- [rest.advertised.host.name](#)

If this is set, this is the hostname that will be given out to other workers to connect to.

	<b>Type:</b>	<b>string</b>
	Default:	null
	Valid Values:	
	Importance:	low

- [rest.advertised.listener](#)

Sets the advertised listener (HTTP or HTTPS) which will be given to other workers to use.

	<b>Type:</b>	<b>string</b>
	Default:	null
	Valid Values:	
	Importance:	low

- [rest.advertised.port](#)

If this is set, this is the port that will be given out to other workers to connect to.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*rest.extension.classes\*\*](#)

Comma-separated names of `ConnectRestExtension` classes, loaded and called in the order specified. Implementing the interface `ConnectRestExtension` allows you to inject into Connect's REST API user defined resources like filters. Typically used to add custom capability like logging, security, etc.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*retry.backoff.ms\*\*](#)

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

Type:	<b>long</b>
Default:	100
Valid Values:	[0,...]
Importance:	low

- [\*\*sasl.kerberos.kinit.cmd\*\*](#)

Kerberos kinit command path.

Type:	<b>string</b>
Default:	/usr/bin/kinit
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.min.time.before.relogin\*\*](#)

Login thread sleep time between refresh attempts.

Type:	<b>long</b>
Default:	60000
Valid Values:	
Importance:	low

- [sasl.kerberos.ticket.renew.jitter](#)

Percentage of random jitter added to the renewal time.

Type:	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	low

- [sasl.kerberos.ticket.renew.window.factor](#)

Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	low

- [sasl.login.connect.timeout.ms](#)

The (optional) value in milliseconds for the external authentication provider connection timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.login.read.timeout.ms](#)

The (optional) value in milliseconds for the external authentication provider read timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.login.refresh.buffer.seconds](#)

The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and sasl.login.refresh.min.period.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	300
Valid Values:	[0,...,3600]
Importance:	low

- [sasl.login.refresh.min.period.seconds](#)

The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and sasl.login.refresh.buffer.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	60
Valid Values:	[0,...,900]
Importance:	low

- [sasl.login.refresh.window.factor](#)

Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.8
Valid Values:	[0.5,...,1.0]
Importance:	low

- [sasl.login.refresh.window.jitter](#)

The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.05
Valid Values:	[0.0,...,0.25]
Importance:	low

- [sasl.login.retry.backoff.max.ms](#)

The (optional) value in milliseconds for the maximum wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [\*\*sasl.login.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.clock.skew.seconds\*\*](#)

The (optional) value in seconds to allow for differences between the time of the OAuth/OIDC identity provider and the broker.

<b>Type:</b>	<b>int</b>
Default:	30
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.expected.audience\*\*](#)

The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth "aud" claim and if this value is set, the broker will match the value from JWT's "aud" claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

<b>Type:</b>	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.expected.issuer\*\*](#)

The (optional) setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth "iss" claim and if this value is set, the broker will match it exactly against what is in the JWT's "iss" claim. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.refresh.ms](#)**

The (optional) value in milliseconds for the broker to wait between refreshing its JWKS (JSON Web Key Set) cache that contains the keys to verify the signature of the JWT.

Type:	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms](#)**

The (optional) value in milliseconds for the maximum wait between attempts to retrieve the JWKS (JSON Web Key Set) from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.jwks.endpoint.retry.backoff.ms](#)**

The (optional) value in milliseconds for the initial wait between JWKS (JSON Web Key Set) retrieval attempts from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- **[sasl.oauthbearer.scope.claim.name](#)**

The OAuth claim for the scope is often named "scope", but this (optional) setting can provide a different name to use for the scope included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	scope
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.sub.claim.name\*\*](#)

The OAuth claim for the subject is often named "sub", but this (optional) setting can provide a different name to use for the subject included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	sub
Valid Values:	
Importance:	low

- [\*\*scheduled.rebalance.max.delay.ms\*\*](#)

The maximum delay that is scheduled in order to wait for the return of one or more departed workers before rebalancing and reassigning their connectors and tasks to the group. During this period the connectors and tasks of the departed workers remain unassigned

Type:	<b>int</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...,2147483647]
Importance:	low

- [\*\*socket.connection.setup.timeout.max.ms\*\*](#)

The maximum amount of time the client will wait for the socket connection to be established. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the timeout resulting in a random range between 20% below and 20% above the computed value.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [\*\*socket.connection.setup.timeout.ms\*\*](#)

The amount of time the client will wait for the socket connection to be established. If the connection is not built before the timeout elapses, clients will close the socket channel.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	[0,...]
Importance:	low

- [ssl.cipher.suites](#)

A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.

<b>Type:</b>	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [ssl.client.auth](#)

Configures kafka broker to request client authentication. The following settings are common:

- `ssl.client.auth=required` If set to required client authentication is required.
- `ssl.client.auth=requested` This means client authentication is optional. unlike required, if this option is set client can choose not to provide authentication information about itself
- `ssl.client.auth=None` This means client authentication is not needed.

<b>Type:</b>	<b>string</b>
Default:	none
Valid Values:	[required, requested, none]
Importance:	low

- [ssl.endpoint.identification.algorithm](#)

The endpoint identification algorithm to validate server hostname using server certificate.

<b>Type:</b>	<b>string</b>
Default:	https
Valid Values:	
Importance:	low

- [ssl.engine.factory.class](#)

The class of type org.apache.kafka.common.security.auth.SslEngineFactory to provide SSLEngine objects. Default value is org.apache.kafka.common.security.ssl.DefaultSslEngineFactory

<b>Type:</b>	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.keymanager.algorithm\*\*](#)

The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.

<b>Type:</b>	<b>string</b>
Default:	SunX509
Valid Values:	
Importance:	low

- [\*\*ssl.secure.random.implementation\*\*](#)

The SecureRandom PRNG implementation to use for SSL cryptography operations.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.trustmanager.algorithm\*\*](#)

The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

<b>Type:</b>	<b>string</b>
Default:	PKIX
Valid Values:	
Importance:	low

- [\*\*status.storage.partitions\*\*](#)

The number of partitions used when creating the status storage topic

<b>Type:</b>	<b>int</b>
Default:	5
Valid Values:	Positive number, or -1 to use the broker's default
Importance:	low

- [\*\*status.storage.replication.factor\*\*](#)

Replication factor used when creating the status storage topic

Type:	<b>short</b>
Default:	3
Valid Values:	Positive number not larger than the number of brokers in the Kafka cluster, or -1 to use the broker's default
Importance:	low

- [\*\*task.shutdown.graceful.timeout.ms\*\*](#)

Amount of time to wait for tasks to shutdown gracefully. This is the total amount of time, not per task. All task have shutdown triggered, then they are waited on sequentially.

Type:	<b>long</b>
Default:	5000 (5 seconds)
Valid Values:	
Importance:	low

- [\*\*topic.creation.enable\*\*](#)

Whether to allow automatic creation of topics used by source connectors, when source connectors are configured with `topic.creation.` properties. Each task will use an admin client to create its topics and will not depend on the Kafka brokers to create topics automatically.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [\*\*topic.tracking.allow.reset\*\*](#)

If set to true, it allows user requests to reset the set of active topics per connector.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [\*\*topic.tracking.enable\*\*](#)

Enable tracking the set of active topics per connector during runtime.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

### [3.5.1 Source Connector Configs](#)

Below is the configuration of a source connector.

- [\*\*name\*\*](#)

Globally unique name to use for this connector.

Type:	<b>string</b>
Default:	
Valid Values:	non-empty string without ISO control characters
Importance:	high

- [\*\*connector.class\*\*](#)

Name or alias of the class for this connector. Must be a subclass of org.apache.kafka.connect.connector.Connector. If the connector is org.apache.kafka.connect.file.FileStreamSinkConnector, you can either specify this full name, or use "FileStreamSink" or "FileStreamSinkConnector" to make the configuration a bit shorter

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*tasks.max\*\*](#)

Maximum number of tasks to use for this connector.

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	high

- [\*\*key.converter\*\*](#)

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*value.converter\*\*](#)

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*header.converter\*\*](#)

HeaderConverter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the header values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro. By default, the SimpleHeaderConverter is used to serialize header values to strings and deserialize them by inferring the schemas.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*config.action.reload\*\*](#)

The action that Connect should take on the connector when changes in external configuration providers result in a change in the connector's configuration properties. A value of 'none' indicates that Connect will do nothing. A value of 'restart' indicates that Connect should restart/reload the connector with the updated configuration properties. The restart may actually be scheduled in the future if the external configuration provider indicates that a configuration value will expire in the future.

Type:	<b>string</b>
Default:	restart
Valid Values:	[none, restart]
Importance:	low

- [transforms](#)

Aliases for the transformations to be applied to records.

Type:	list
Default:	""
Valid Values:	non-null string, unique transformation aliases
Importance:	low

- [predicates](#)

Aliases for the predicates used by transformations.

Type:	list
Default:	""
Valid Values:	non-null string, unique predicate aliases
Importance:	low

- [errors.retry.timeout](#)

The maximum duration in milliseconds that a failed operation will be reattempted. The default is 0, which means no retries will be attempted. Use -1 for infinite retries.

Type:	long
Default:	0
Valid Values:	
Importance:	medium

- [errors.retry.delay.max.ms](#)

The maximum duration in milliseconds between consecutive retry attempts. Jitter will be added to the delay once this limit is reached to prevent thundering herd issues.

Type:	long
Default:	60000 (1 minute)
Valid Values:	
Importance:	medium

- [errors.tolerance](#)

Behavior for tolerating errors during connector operation. 'none' is the default value and signals that any error will result in an immediate connector task failure; 'all' changes the behavior to skip over problematic records.

Type:	<b>string</b>
Default:	none
Valid Values:	[none, all]
Importance:	medium

- [\*\*errors.log.enable\*\*](#)

If true, write each error and the details of the failed operation and problematic record to the Connect application log. This is 'false' by default, so that only errors that are not tolerated are reported.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

- [\*\*errors.log.include.messages\*\*](#)

Whether to include in the log the Connect record that resulted in a failure. For sink records, the topic, partition, offset, and timestamp will be logged. For source records, the key and value (and their schemas), all headers, and the timestamp, Kafka topic, Kafka partition, source partition, and source offset will be logged. This is 'false' by default, which will prevent record keys, values, and headers from being written to log files.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

- [\*\*topic.creation.groups\*\*](#)

Groups of configurations for topics created by source connectors

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string, unique topic creation groups
Importance:	low

- [\*\*exactly.once.support\*\*](#)

Permitted values are requested, required. If set to "required", forces a preflight check for the connector to ensure that it can provide exactly-once semantics with the given configuration. Some connectors may be capable of providing exactly-once semantics but not signal to Connect that they support this; in that case, documentation for the connector should be consulted carefully before creating it, and the value for this property should be set to "requested". Additionally, if the value is set to "required" but the worker that performs preflight validation does not have exactly-once support enabled for source connectors, requests to create or validate the connector will fail.

Type:	<b>string</b>
Default:	requested
Valid Values:	(case insensitive) [REQUIRED, REQUESTED]
Importance:	medium

- [\*\*transaction.boundary\*\*](#)

Permitted values are: poll, interval, connector. If set to 'poll', a new producer transaction will be started and committed for every batch of records that each task from this connector provides to Connect. If set to 'connector', relies on connector-defined transaction boundaries; note that not all connectors are capable of defining their own transaction boundaries, and in that case, attempts to instantiate a connector with this value will fail. Finally, if set to 'interval', commits transactions only after a user-defined time interval has passed.

Type:	<b>string</b>
Default:	poll
Valid Values:	(case insensitive) [INTERVAL, POLL, CONNECTOR]
Importance:	medium

- [\*\*transaction.boundary.interval.ms\*\*](#)

If 'transaction.boundary' is set to 'interval', determines the interval for producer transaction commits by connector tasks. If unset, defaults to the value of the worker-level 'offset.flush.interval.ms' property. It has no effect if a different transaction.boundary is specified.

Type:	<b>long</b>
Default:	null
Valid Values:	[0,...]
Importance:	low

- [\*\*offsets.storage.topic\*\*](#)

The name of a separate offsets topic to use for this connector. If empty or not specified, the worker's global offsets topic name will be used. If specified, the offsets topic will be created if it does not already exist on the Kafka cluster targeted by this connector (which may be different from the one used for the worker's global offsets topic if the bootstrap.servers property of the connector's producer has been overridden from the worker's). Only applicable in distributed mode; in standalone mode, setting this property will have no effect.

Type:	<b>string</b>
Default:	null
Valid Values:	non-empty string
Importance:	low

### 3.5.2 Sink Connector Configs

Below is the configuration of a sink connector.

- [name](#)

Globally unique name to use for this connector.

Type:	<b>string</b>
Default:	
Valid Values:	non-empty string without ISO control characters
Importance:	high

- [connector.class](#)

Name or alias of the class for this connector. Must be a subclass of `org.apache.kafka.connect.connector.Connector`. If the connector is `org.apache.kafka.connect.file FileStreamSinkConnector`, you can either specify this full name, or use "FileStreamSink" or "FileStreamSinkConnector" to make the configuration a bit shorter

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [tasks.max](#)

Maximum number of tasks to use for this connector.

Type:	<b>int</b>
Default:	1
Valid Values:	[1,...]
Importance:	high

- [topics](#)

List of topics to consume, separated by commas

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	high

- [topics.regex](#)

Regular expression giving topics to consume. Under the hood, the regex is compiled to a `java.util.regex.Pattern`. Only one of topics or topics.regex should be specified.

Type:	<b>string</b>
Default:	""
Valid Values:	valid regex
Importance:	high

- **[key.converter](#)**

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- **[value.converter](#)**

Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- **[header.converter](#)**

HeaderConverter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the header values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro. By default, the SimpleHeaderConverter is used to serialize header values to strings and deserialize them by inferring the schemas.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*config.action.reload\*\*](#)

The action that Connect should take on the connector when changes in external configuration providers result in a change in the connector's configuration properties. A value of 'none' indicates that Connect will do nothing. A value of 'restart' indicates that Connect should restart/reload the connector with the updated configuration properties. The restart may actually be scheduled in the future if the external configuration provider indicates that a configuration value will expire in the future.

Type:	<b>string</b>
Default:	restart
Valid Values:	[none, restart]
Importance:	low

- [\*\*transforms\*\*](#)

Aliases for the transformations to be applied to records.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string, unique transformation aliases
Importance:	low

- [\*\*predicates\*\*](#)

Aliases for the predicates used by transformations.

Type:	<b>list</b>
Default:	""
Valid Values:	non-null string, unique predicate aliases
Importance:	low

- [\*\*errors.retry.timeout\*\*](#)

The maximum duration in milliseconds that a failed operation will be reattempted. The default is 0, which means no retries will be attempted. Use -1 for infinite retries.

Type:	<b>long</b>
Default:	0
Valid Values:	
Importance:	medium

- [\*\*errors.retry.delay.max.ms\*\*](#)

The maximum duration in milliseconds between consecutive retry attempts. Jitter will be added to the delay once this limit is reached to prevent thundering herd issues.

Type:	<b>long</b>
Default:	60000 (1 minute)
Valid Values:	
Importance:	medium

- [\*\*errors.tolerance\*\*](#)

Behavior for tolerating errors during connector operation. 'none' is the default value and signals that any error will result in an immediate connector task failure; 'all' changes the behavior to skip over problematic records.

Type:	<b>string</b>
Default:	none
Valid Values:	[none, all]
Importance:	medium

- [\*\*errors.log.enable\*\*](#)

If true, write each error and the details of the failed operation and problematic record to the Connect application log. This is 'false' by default, so that only errors that are not tolerated are reported.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

- [\*\*errors.log.include.messages\*\*](#)

Whether to include in the log the Connect record that resulted in a failure. For sink records, the topic, partition, offset, and timestamp will be logged. For source records, the key and value (and their schemas), all headers, and the timestamp, Kafka topic, Kafka partition, source partition, and source offset will be logged. This is 'false' by default, which will prevent record keys, values, and headers from being written to log files.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

- [\*\*errors.deadletterqueue.topic.name\*\*](#)

The name of the topic to be used as the dead letter queue (DLQ) for messages that result in an error when processed by this sink connector, or its transformations or converters. The topic name is blank by default, which means that no messages are to be recorded in the DLQ.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium

- [\*\*errors.deadletterqueue.topic.replication.factor\*\*](#)

Replication factor used to create the dead letter queue topic when it doesn't already exist.

Type:	<b>short</b>
Default:	3
Valid Values:	
Importance:	medium

- [\*\*errors.deadletterqueue.context.headers.enable\*\*](#)

If true, add headers containing error context to the messages written to the dead letter queue. To avoid clashing with headers from the original record, all error context header keys, all error context header keys will start with `__connect.errors`.

Type:	<b>boolean</b>
Default:	false
Valid Values:	
Importance:	medium

## [\*\*3.6 Kafka Streams Configs\*\*](#)

Below is the configuration of the Kafka Streams client library.

- [\*\*application.id\*\*](#)

An identifier for the stream processing application. Must be unique within the Kafka cluster. It is used as 1) the default client-id prefix, 2) the group-id for membership management, 3) the changelog topic prefix.

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*bootstrap.servers\*\*](#)

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

Type:	<b>list</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*num.standby.replicas\*\*](#)

The number of standby replicas for each task.

Type:	<b>int</b>
Default:	0
Valid Values:	
Importance:	high

- [\*\*state.dir\*\*](#)

Directory location for state store. This path must be unique for each streams instance sharing the same underlying filesystem.

Type:	<b>string</b>
Default:	/var/folders/0t/68svdzmx1sld0mxjl8dgmmzm0000gq/T//kafka-streams
Valid Values:	
Importance:	high

- [\*\*acceptable.recovery.lag\*\*](#)

The maximum acceptable lag (number of offsets to catch up) for a client to be considered caught-up enough to receive an active task assignment. Upon assignment, it will still restore the rest of the changelog before processing. To avoid a pause in processing during rebalances, this config should correspond to a recovery time of well under a minute for a given workload. Must be at least 0.

Type:	<b>long</b>
Default:	10000
Valid Values:	[0,...]
Importance:	medium

- [\*\*cache.max.bytes.buffering\*\*](#)

Maximum number of memory bytes to be used for buffering across all threads

Type:	<b>long</b>
Default:	10485760
Valid Values:	[0,...]
Importance:	medium

- [client.id](#)

An ID prefix string used for the client IDs of internal consumer, producer and restore-consumer, with pattern `<client.id>-StreamThread-<threadSequenceNumber$gt;-<consumer|producer|restore-consumer>`.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium

- [default.deserialization.exception.handler](#)

Exception handling class that implements the `org.apache.kafka.streams.errors.DeserializationExceptionHandler` interface.

Type:	<b>class</b>
Default:	<code>org.apache.kafka.streams.errors.LogAndFailExceptionHandler</code>
Valid Values:	
Importance:	medium

- [default.key.serde](#)

Default serializer / deserializer class for key that implements the `org.apache.kafka.common.serialization.Serde` interface. Note when windowed serde class is used, one needs to set the inner serde class that implements the `org.apache.kafka.common.serialization.Serde` interface via 'default.windowed.key.serde.inner' or 'default.windowed.value.serde.inner' as well

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [default.list.key.serde.inner](#)

Default inner class of list serde for key that implements the `org.apache.kafka.common.serialization.Serde` interface. This configuration will be read if and only if `default.key.serde` configuration is set to `org.apache.kafka.common.serialization.Serdes.ListSerde`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*default.list.key.serde.type\*\*](#)

Default class for key that implements the `java.util.List` interface. This configuration will be read if and only if `default.key.serde` configuration is set to `org.apache.kafka.common.serialization.Serdes.ListSerde`. Note when list serde class is used, one needs to set the inner serde class that implements the `org.apache.kafka.common.serialization.Serde` interface via 'default.list.key.serde.inner'

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*default.list.value.serde.inner\*\*](#)

Default inner class of list serde for value that implements the `org.apache.kafka.common.serialization.Serde` interface. This configuration will be read if and only if `default.value.serde` configuration is set to `org.apache.kafka.common.serialization.Serdes.ListSerde`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*default.list.value.serde.type\*\*](#)

Default class for value that implements the `java.util.List` interface. This configuration will be read if and only if `default.value.serde` configuration is set to `org.apache.kafka.common.serialization.Serdes.ListSerde`. Note when list serde class is used, one needs to set the inner serde class that implements the `org.apache.kafka.common.serialization.Serde` interface via 'default.list.value.serde.inner'

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*default.production.exception.handler\*\*](#)

Exception handling class that implements the `org.apache.kafka.streams.errors.ProductionExceptionHandler` interface.

Type:	<b>class</b>
Default:	<code>org.apache.kafka.streams.errors.DefaultProductionExceptionHandler</code>
Valid Values:	
Importance:	medium

- [default.timestamp.extractor](#)

Default timestamp extractor class that implements the `org.apache.kafka.streams.processor.TimestampExtractor` interface.

Type:	<b>class</b>
Default:	<code>org.apache.kafka.streams.processor.FailOnInvalidTimestamp</code>
Valid Values:	
Importance:	medium

- [default.value.serde](#)

Default serializer / deserializer class for value that implements the `org.apache.kafka.common.serialization.Serde` interface. Note when windowed serde class is used, one needs to set the inner serde class that implements the `org.apache.kafka.common.serialization.Serde` interface via 'default.windowed.key.serde.inner' or 'default.windowed.value.serde.inner' as well

Type:	<b>class</b>
Default:	<code>null</code>
Valid Values:	
Importance:	medium

- [max.task.idle.ms](#)

This config controls whether joins and merges may produce out-of-order results. The config value is the maximum amount of time in milliseconds a stream task will stay idle when it is fully caught up on some (but not all) input partitions to wait for producers to send additional records and avoid potential out-of-order record processing across multiple input streams. The default (zero) does not wait for producers to send more records, but it does wait to fetch data that is already present on the brokers. This default means that for records that are already present on the brokers, Streams will process them in timestamp order. Set to -1 to disable idling entirely and process any locally available data, even though doing so may produce out-of-order processing.

Type:	<b>long</b>
Default:	0
Valid Values:	
Importance:	medium

- [max.warmup.replicas](#)

The maximum number of warmup replicas (extra standbys beyond the configured num.standbys) that can be assigned at once for the purpose of keeping the task available on one instance while it is warming up on another instance it has been reassigned to. Used to throttle how much extra broker traffic and cluster state can be used for high availability. Must be at least 1.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	medium

- [\*\*num.stream.threads\*\*](#)

The number of threads to execute stream processing.

Type:	<b>int</b>
Default:	1
Valid Values:	
Importance:	medium

- [\*\*processing.guarantee\*\*](#)

The processing guarantee that should be used. Possible values are `at_least_once` (default) and `exactly_once_v2` (requires brokers version 2.5 or higher). Deprecated options are `exactly_once` (requires brokers version 0.11.0 or higher) and `exactly_once_beta` (requires brokers version 2.5 or higher). Note that exactly-once processing requires a cluster of at least three brokers by default what is the recommended setting for production; for development you can change this, by adjusting broker setting `transaction.state.log.replication.factor` and `transaction.state.log.min_isr`.

Type:	<b>string</b>
Default:	<code>at_least_once</code>
Valid Values:	[ <code>at_least_once</code> , <code>exactly_once</code> , <code>exactly_once_beta</code> , <code>exactly_once_v2</code> ]
Importance:	medium

- [\*\*rack.aware.assignment.tags\*\*](#)

List of client tag keys used to distribute standby replicas across Kafka Streams instances. When configured, Kafka Streams will make a best-effort to distribute the standby tasks over each client tag dimension.

Type:	<b>list</b>
Default:	""
Valid Values:	List containing maximum of 5 elements
Importance:	medium

- [\*\*replication.factor\*\*](#)

The replication factor for change log topics and repartition topics created by the stream processing application. The default of `-1` (meaning: use broker default replication factor) requires broker version 2.4 or newer

Type:	<b>int</b>
Default:	-1
Valid Values:	
Importance:	medium

- [\*\*security.protocol\*\*](#)

Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL.

Type:	<b>string</b>
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium

- [\*\*statestore.cache.max.bytes\*\*](#)

Maximum number of memory bytes to be used for statestore cache across all threads

Type:	<b>long</b>
Default:	10485760 (10 mebibytes)
Valid Values:	[0,...]
Importance:	medium

- [\*\*task.timeout.ms\*\*](#)

The maximum amount of time in milliseconds a task might stall due to internal errors and retries until an error is raised. For a timeout of 0ms, a task would raise an error for the first internal error. For any timeout larger than 0ms, a task will retry at least once before an error is raised.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	medium

- [\*\*topology.optimization\*\*](#)

A configuration telling Kafka Streams if it should optimize the topology and what optimizations to apply. Acceptable values are: "+NO\_OPTIMIZATION+", "+OPTIMIZE+", or a comma separated list of specific optimizations: ("+REUSE\_KTABLE\_SOURCE\_TOPICS+", "+MERGE\_REPARTITION\_TOPICS+" + "SINGLE\_STORE\_SELF\_JOIN+")."NO\_OPTIMIZATION" by default.

Type:	<b>string</b>
Default:	none
Valid Values:	org.apache.kafka.streams.StreamsConfig\$\$Lambda\$3/521645586@49c2faae
Importance:	medium

- [application.server](#)

A host:port pair pointing to a user-defined endpoint that can be used for state store discovery and interactive queries on this KafkaStreams instance.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	low

- [auto.include.jmx.reporter](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [buffered.records.per.partition](#)

Maximum number of records to buffer per partition.

Type:	<b>int</b>
Default:	1000
Valid Values:	
Importance:	low

- [built.in.metrics.version](#)

Version of the built-in metrics to use.

Type:	<b>string</b>
Default:	latest
Valid Values:	[latest]
Importance:	low

- [commit.interval.ms](#)

The frequency in milliseconds with which to commit processing progress. For at-least-once processing, committing means to save the position (ie, offsets) of the processor. For exactly-once processing, it means to commit the transaction which includes to save the position and to make the committed data in the output topic visible to consumers with isolation level read\_committed. (Note, if `processing.guarantee` is set to `exactly_once_v2`, `exactly_once`, the default value is `100`, otherwise the default value is `30000`).

<b>Type:</b>	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [\*\*connections.max.idle.ms\*\*](#)

Close idle connections after the number of milliseconds specified by this config.

<b>Type:</b>	<b>long</b>
Default:	540000 (9 minutes)
Valid Values:	
Importance:	low

- [\*\*default.dsl.store\*\*](#)

The default state store type used by DSL operators.

<b>Type:</b>	<b>string</b>
Default:	rocksDB
Valid Values:	[rocksDB, in_memory]
Importance:	low

- [\*\*metadata.max.age.ms\*\*](#)

The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

<b>Type:</b>	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	low

- [\*\*metric.reporters\*\*](#)

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

<b>Type:</b>	<b>list</b>
Default:	""
Valid Values:	
Importance:	low

- [metrics.num.samples](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low

- [metrics.recording.level](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	[INFO, DEBUG, TRACE]
Importance:	low

- [metrics.sample.window.ms](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [poll.ms](#)

The amount of time in milliseconds to block waiting for input.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [probing.rebalance.interval.ms](#)

The maximum time in milliseconds to wait before triggering a rebalance to probe for warmup replicas that have finished warming up and are ready to become active. Probing rebalances will continue to be triggered until the assignment is balanced. Must be at least 1 minute.

Type:	<b>long</b>
Default:	600000 (10 minutes)
Valid Values:	[60000,...]
Importance:	low

- [receive.buffer.bytes](#)

The size of the TCP receive buffer (SO\_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	32768 (32 kibibytes)
Valid Values:	[-1,...]
Importance:	low

- [reconnect.backoff.max.ms](#)

The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.

Type:	<b>long</b>
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	low

- [reconnect.backoff.ms](#)

The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.

Type:	<b>long</b>
Default:	50
Valid Values:	[0,...]
Importance:	low

- [repartition.purge.interval.ms](#)

The frequency in milliseconds with which to delete fully consumed records from repartition topics. Purging will occur after at least this value since the last purge, but may be delayed until later. (Note, unlike `commit.interval.ms`, the default for this value remains unchanged when `processing.guarantee` is set to `exactly_once_v2` ).

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [request.timeout.ms](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

<b>Type:</b>	<b>int</b>
Default:	40000 (40 seconds)
Valid Values:	[0,...]
Importance:	low

- [retries](#)

Setting a value greater than zero will cause the client to resend any request that fails with a potentially transient error. It is recommended to set the value to either zero or `MAX_VALUE` and use corresponding timeout parameters to control how long a client should retry a request.

<b>Type:</b>	<b>int</b>
Default:	0
Valid Values:	[0,...,2147483647]
Importance:	low

- [retry.backoff.ms](#)

The amount of time to wait before attempting to retry a failed request to a given topic partition. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	[0,...]
Importance:	low

- [rocksdb.config.setter](#)

A Rocks DB config setter class or class name that implements the `org.apache.kafka.streams.state.RocksDBConfigSetter` interface

<b>Type:</b>	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- [send.buffer.bytes](#)

The size of the TCP send buffer (SO\_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	131072 (128 kibibytes)
Valid Values:	[-1,...]
Importance:	low

- [\*\*state.cleanup.delay.ms\*\*](#)

The amount of time in milliseconds to wait before deleting state when a partition has migrated. Only state directories that have not been modified for at least `state.cleanup.delay.ms` will be removed

Type:	<b>long</b>
Default:	600000 (10 minutes)
Valid Values:	
Importance:	low

- [\*\*upgrade.from\*\*](#)

Allows upgrading in a backward compatible way. This is needed when upgrading from [0.10.0, 1.1] to 2.0+, or when upgrading from [2.0, 2.3] to 2.4+. When upgrading from 3.3 to a newer version it is not required to specify this config. Default is `null`. Accepted values are "0.10.0", "0.10.1", "0.10.2", "0.11.0", "1.0", "1.1", "2.0", "2.1", "2.2", "2.3", "2.4", "2.5", "2.6", "2.7", "2.8", "3.0", "3.1", "3.2", "3.3" (for upgrading from the corresponding old version).

Type:	<b>string</b>
Default:	<code>null</code>
Valid Values:	[ <code>null</code> , 0.10.0, 0.10.1, 0.10.2, 0.11.0, 1.0, 1.1, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 3.0, 3.1, 3.2, 3.3]
Importance:	low

- [\*\*window.size.ms\*\*](#)

Sets window size for the deserializer in order to calculate window end times.

Type:	<b>long</b>
Default:	<code>null</code>
Valid Values:	
Importance:	low

- [\*\*windowed.inner.class.serde\*\*](#)

Default serializer / deserializer for the inner class of a windowed record. Must implement the `org.apache.kafka.common.serialization.Serde` interface. Note that setting this config in KafkaStreams application would result in an error as it is meant to be used only from Plain consumer client.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*windowstore.changelog.additional.retention.ms\*\*](#)

Added to a windows maintainMs to ensure data is not deleted from the log prematurely. Allows for clock drift. Default is 1 day

Type:	<b>long</b>
Default:	86400000 (1 day)
Valid Values:	
Importance:	low

## 3.7 Admin Configs

Below is the configuration of the Kafka Admin client library.

- [\*\*bootstrap.servers\*\*](#)

A list of host/port pairs to use for establishing the initial connection to the Kafka cluster. The client will make use of all servers irrespective of which servers are specified here for bootstrapping—this list only impacts the initial hosts used to discover the full set of servers. This list should be in the form `host1:port1,host2:port2,...`. Since these servers are just used for the initial connection to discover the full cluster membership (which may change dynamically), this list need not contain the full set of servers (you may want more than one, though, in case a server is down).

Type:	<b>list</b>
Default:	
Valid Values:	
Importance:	high

- [\*\*ssl.key.password\*\*](#)

The password of the private key in the key store file or the PEM key specified in 'ssl.keystore.key'.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.certificate.chain\*\*](#)

Certificate chain in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with a list of X.509 certificates

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.key\*\*](#)

Private key in the format specified by 'ssl.keystore.type'. Default SSL engine factory supports only PEM format with PKCS#8 keys. If the key is encrypted, key password must be specified using 'ssl.key.password'

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.location\*\*](#)

The location of the key store file. This is optional for client and can be used for two-way authentication for client.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.keystore.password\*\*](#)

The store password for the key store file. This is optional for client and only needed if 'ssl.keystore.location' is configured. Key store password is not supported for PEM format.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.certificates\*\*](#)

Trusted certificates in the format specified by 'ssl.truststore.type'. Default SSL engine factory supports only PEM format with X.509 certificates.

<b>Type:</b>	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.location\*\*](#)

The location of the trust store file.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*ssl.truststore.password\*\*](#)

The password for the trust store file. If a password is not set, trust store file configured will still be used, but integrity checking is disabled. Trust store password is not supported for PEM format.

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	high

- [\*\*client.dns.lookup\*\*](#)

Controls how the client uses DNS lookups. If set to `use_all_dns_ips`, connect to each returned IP address in sequence until a successful connection is established. After a disconnection, the next IP is used. Once all IPs have been used once, the client resolves the IP(s) from the hostname again (both the JVM and the OS cache DNS name lookups, however). If set to `resolve_canonical_bootstrap_servers_only`, resolve each bootstrap address into a list of canonical names. After the bootstrap phase, this behaves the same as `use_all_dns_ips`.

Type:	<b>string</b>
Default:	<code>use_all_dns_ips</code>
Valid Values:	[ <code>use_all_dns_ips</code> , <code>resolve_canonical_bootstrap_servers_only</code> ]
Importance:	medium

- [\*\*client.id\*\*](#)

An id string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium

- [connections.max.idle.ms](#)

Close idle connections after the number of milliseconds specified by this config.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	
Importance:	medium

- [default.api.timeout.ms](#)

Specifies the timeout (in milliseconds) for client APIs. This configuration is used as the default timeout for all client operations that do not specify a `timeout` parameter.

Type:	<b>int</b>
Default:	60000 (1 minute)
Valid Values:	[0,...]
Importance:	medium

- [receive.buffer.bytes](#)

The size of the TCP receive buffer (SO\_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	65536 (64 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- [request.timeout.ms](#)

The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.

Type:	<b>int</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	medium

- [sasl.client.callback.handler.class](#)

The fully qualified name of a SASL client callback handler class that implements the `AuthenticateCallbackHandler` interface.

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.jaas.config**

JAAS login context parameters for SASL connections in the format used by JAAS configuration files. JAAS configuration file format is described [here](#). The format for the value is: `loginModuleClass controlFlag (optionName=optionValue)*;`. For brokers, the config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=com.example.ScramLoginModule` required;

Type:	<b>password</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.kerberos.service.name**

The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.login.callback.handler.class**

The fully qualified name of a SASL login callback handler class that implements the `AuthenticateCallbackHandler` interface. For brokers, login callback handler config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.login.callback.handler.class=com.example.CustomScramLoginCallbackHandler`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.login.class**

The fully qualified name of a class that implements the `Login` interface. For brokers, login config must be prefixed with listener prefix and SASL mechanism name in lower-case. For example, `listener.name.sasl_ssl.scram-sha-256.sasl.login.class=com.example.CustomScramLogin`

Type:	<b>class</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.mechanism**

SASL mechanism used for client connections. This may be any mechanism for which a security provider is available. GSSAPI is the default mechanism.

Type:	<b>string</b>
Default:	GSSAPI
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.jwks.endpoint.url**

The OAuth/OIDC provider URL from which the provider's [JWKS \(JSON Web Key Set\)](#) can be retrieved. The URL can be HTTP(S)-based or file-based. If the URL is HTTP(S)-based, the JWKS data will be retrieved from the OAuth/OIDC provider via the configured URL on broker startup. All then-current keys will be cached on the broker for incoming requests. If an authentication request is received for a JWT that includes a "kid" header claim value that isn't yet in the cache, the JWKS endpoint will be queried again on demand. However, the broker polls the URL every `sasl.oauthbearer.jwks.endpoint.refresh.ms` milliseconds to refresh the cache with any forthcoming keys before any JWT requests that include them are received. If the URL is file-based, the broker will load the JWKS file from a configured location on startup. In the event that the JWT includes a "kid" header value that isn't in the JWKS file, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- **sasl.oauthbearer.token.endpoint.url**

The URL for the OAuth/OIDC identity provider. If the URL is HTTP(S)-based, it is the issuer's token endpoint URL to which requests will be made to login based on the configuration in `sasl.jaas.config`. If the URL is file-based, it specifies a file containing an access token (in JWT serialized form) issued by the OAuth/OIDC identity provider to use for authorization.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*security.protocol\*\*](#)

Protocol used to communicate with brokers. Valid values are: PLAINTEXT, SSL, SASL\_PLAINTEXT, SASL\_SSL.

Type:	<b>string</b>
Default:	PLAINTEXT
Valid Values:	[PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL]
Importance:	medium

- [\*\*send.buffer.bytes\*\*](#)

The size of the TCP send buffer (SO\_SNDBUF) to use when sending data. If the value is -1, the OS default will be used.

Type:	<b>int</b>
Default:	131072 (128 kibibytes)
Valid Values:	[-1,...]
Importance:	medium

- [\*\*socket.connection.setup.timeout.max.ms\*\*](#)

The maximum amount of time the client will wait for the socket connection to be established. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the timeout resulting in a random range between 20% below and 20% above the computed value.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	
Importance:	medium

- [\*\*socket.connection.setup.timeout.ms\*\*](#)

The amount of time the client will wait for the socket connection to be established. If the connection is not built before the timeout elapses, clients will close the socket channel.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	medium

- [\*\*ssl.enabled.protocols\*\*](#)

The list of protocols enabled for SSL connections. The default is 'TLSv1.2,TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. With the default value for Java 11, clients and servers will prefer TLSv1.3 if both support it and fallback to TLSv1.2 otherwise (assuming both support at least TLSv1.2). This default should be fine for most cases. Also see the config documentation for `ssl.protocol`.

Type:	<b>list</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [\*\*ssl.keystore.type\*\*](#)

The file format of the key store file. This is optional for client. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [\*\*ssl.protocol\*\*](#)

The SSL protocol used to generate the SSLContext. The default is 'TLSv1.3' when running with Java 11 or newer, 'TLSv1.2' otherwise. This value should be fine for most use cases. Allowed values in recent JVMs are 'TLSv1.2' and 'TLSv1.3'. 'TLS', 'TLSv1.1', 'SSL', 'SSLv2' and 'SSLv3' may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities. With the default value for this config and 'ssl.enabled.protocols', clients will downgrade to 'TLSv1.2' if the server does not support 'TLSv1.3'. If this config is set to 'TLSv1.2', clients will not use 'TLSv1.3' even if it is one of the values in `ssl.enabled.protocols` and the server only supports 'TLSv1.3'.

Type:	<b>string</b>
Default:	TLSv1.2
Valid Values:	
Importance:	medium

- [\*\*ssl.provider\*\*](#)

The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*ssl.truststore.type\*\*](#)

The file format of the trust store file. The values currently supported by the default `ssl.engine.factory.class` are [JKS, PKCS12, PEM].

Type:	<b>string</b>
Default:	JKS
Valid Values:	
Importance:	medium

- [\*\*auto.include.jmx.reporter\*\*](#)

Deprecated. Whether to automatically include JmxReporter even if it's not listed in `metric.reporters`. This configuration will be removed in Kafka 4.0, users should instead include `org.apache.kafka.common.metrics.JmxReporter` in `metric.reporters` in order to enable the JmxReporter.

Type:	<b>boolean</b>
Default:	true
Valid Values:	
Importance:	low

- [\*\*metadata.max.age.ms\*\*](#)

The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

Type:	<b>long</b>
Default:	300000 (5 minutes)
Valid Values:	[0,...]
Importance:	low

- [\*\*metric.reporters\*\*](#)

A list of classes to use as metrics reporters. Implementing the `org.apache.kafka.common.metrics.MetricsReporter` interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	low

- [\*\*metrics.num.samples\*\*](#)

The number of samples maintained to compute metrics.

Type:	<b>int</b>
Default:	2
Valid Values:	[1,...]
Importance:	low

- [metrics.recording.level](#)

The highest recording level for metrics.

Type:	<b>string</b>
Default:	INFO
Valid Values:	[INFO, DEBUG, TRACE]
Importance:	low

- [metrics.sample.window.ms](#)

The window of time a metrics sample is computed over.

Type:	<b>long</b>
Default:	30000 (30 seconds)
Valid Values:	[0,...]
Importance:	low

- [reconnect.backoff.max.ms](#)

The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.

Type:	<b>long</b>
Default:	1000 (1 second)
Valid Values:	[0,...]
Importance:	low

- [reconnect.backoff.ms](#)

The base amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all connection attempts by the client to a broker.

Type:	<b>long</b>
Default:	50
Valid Values:	[0,...]
Importance:	low

- [retries](#)

Setting a value greater than zero will cause the client to resend any request that fails with a potentially transient error. It is recommended to set the value to either zero or `MAX_VALUE` and use corresponding timeout parameters to control how long a client should retry a request.

<b>Type:</b>	<b>int</b>
Default:	2147483647
Valid Values:	[0,...,2147483647]
Importance:	low

- [\*\*retry.backoff.ms\*\*](#)

The amount of time to wait before attempting to retry a failed request. This avoids repeatedly sending requests in a tight loop under some failure scenarios.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	[0,...]
Importance:	low

- [\*\*sasl.kerberos.kinit.cmd\*\*](#)

Kerberos kinit command path.

<b>Type:</b>	<b>string</b>
Default:	/usr/bin/kinit
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.min.time.before.relogin\*\*](#)

Login thread sleep time between refresh attempts.

<b>Type:</b>	<b>long</b>
Default:	60000
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.ticket.renew.jitter\*\*](#)

Percentage of random jitter added to the renewal time.

<b>Type:</b>	<b>double</b>
Default:	0.05
Valid Values:	
Importance:	low

- [\*\*sasl.kerberos.ticket.renew.window.factor\*\*](#)

Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.

Type:	<b>double</b>
Default:	0.8
Valid Values:	
Importance:	low

- [\*\*sasl.login.connect.timeout.ms\*\*](#)

The (optional) value in milliseconds for the external authentication provider connection timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.login.read.timeout.ms\*\*](#)

The (optional) value in milliseconds for the external authentication provider read timeout. Currently applies only to OAUTHBEARER.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.login.refresh.buffer.seconds\*\*](#)

The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and sasl.login.refresh.min.period.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	300
Valid Values:	[0,...,3600]
Importance:	low

- [\*\*sasl.login.refresh.min.period.seconds\*\*](#)

The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and sasl.login.refresh.buffer.seconds are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

Type:	<b>short</b>
Default:	60
Valid Values:	[0,...,900]
Importance:	low

- [\*\*sasl.login.refresh.window.factor\*\*](#)

Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.8
Valid Values:	[0.5,...,1.0]
Importance:	low

- [\*\*sasl.login.refresh.window.jitter\*\*](#)

The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.

Type:	<b>double</b>
Default:	0.05
Valid Values:	[0.0,...,0.25]
Importance:	low

- [\*\*sasl.login.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [sasl.login.retry.backoff.ms](#)

The (optional) value in milliseconds for the initial wait between login attempts to the external authentication provider. Login uses an exponential backoff algorithm with an initial wait based on the sasl.login.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.login.retry.backoff.max.ms setting. Currently applies only to OAUTHBEARER.

Type:	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [sasl.oauthbearer.clock.skew.seconds](#)

The (optional) value in seconds to allow for differences between the time of the OAuth/OIDC identity provider and the broker.

Type:	<b>int</b>
Default:	30
Valid Values:	
Importance:	low

- [sasl.oauthbearer.expected.audience](#)

The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth "aud" claim and if this value is set, the broker will match the value from JWT's "aud" claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [sasl.oauthbearer.expected.issuer](#)

The (optional) setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth "iss" claim and if this value is set, the broker will match it exactly against what is in the JWT's "iss" claim. If there is no match, the broker will reject the JWT and authentication will fail.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.refresh.ms\*\*](#)

The (optional) value in milliseconds for the broker to wait between refreshing its JWKS (JSON Web Key Set) cache that contains the keys to verify the signature of the JWT.

<b>Type:</b>	<b>long</b>
Default:	3600000 (1 hour)
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms\*\*](#)

The (optional) value in milliseconds for the maximum wait between attempts to retrieve the JWKS (JSON Web Key Set) from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

<b>Type:</b>	<b>long</b>
Default:	10000 (10 seconds)
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.jwks.endpoint.retry.backoff.ms\*\*](#)

The (optional) value in milliseconds for the initial wait between JWKS (JSON Web Key Set) retrieval attempts from the external authentication provider. JWKS retrieval uses an exponential backoff algorithm with an initial wait based on the sasl.oauthbearer.jwks.endpoint.retry.backoff.ms setting and will double in wait length between attempts up to a maximum wait length specified by the sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms setting.

<b>Type:</b>	<b>long</b>
Default:	100
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.scope.claim.name\*\*](#)

The OAuth claim for the scope is often named "scope", but this (optional) setting can provide a different name to use for the scope included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

<b>Type:</b>	<b>string</b>
Default:	scope
Valid Values:	
Importance:	low

- [\*\*sasl.oauthbearer.sub.claim.name\*\*](#)

The OAuth claim for the subject is often named "sub", but this (optional) setting can provide a different name to use for the subject included in the JWT payload's claims if the OAuth/OIDC provider uses a different name for that claim.

Type:	<b>string</b>
Default:	sub
Valid Values:	
Importance:	low

- [\*\*security.providers\*\*](#)

A list of configurable creator classes each returning a provider implementing security algorithms. These classes should implement the

`org.apache.kafka.common.security.auth.SecurityProviderCreator` interface.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.cipher.suites\*\*](#)

A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default all the available cipher suites are supported.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [\*\*ssl.endpoint.identification.algorithm\*\*](#)

The endpoint identification algorithm to validate server hostname using server certificate.

Type:	<b>string</b>
Default:	https
Valid Values:	
Importance:	low

- [\*\*ssl.engine.factory.class\*\*](#)

The class of type `org.apache.kafka.common.security.auth.SslEngineFactory` to provide SSLEngine objects. Default value is `org.apache.kafka.common.security.ssl.DefaultSslEngineFactory`

<b>Type:</b>	<b>class</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.keymanager.algorithm](#)**

The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.

<b>Type:</b>	<b>string</b>
Default:	SunX509
Valid Values:	
Importance:	low

- **[ssl.secure.random.implementation](#)**

The SecureRandom PRNG implementation to use for SSL cryptography operations.

<b>Type:</b>	<b>string</b>
Default:	null
Valid Values:	
Importance:	low

- **[ssl.trustmanager.algorithm](#)**

The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.

<b>Type:</b>	<b>string</b>
Default:	PKIX
Valid Values:	
Importance:	low

## **[4. DESIGN](#)**

### **[4.1 Motivation](#)**

We designed Kafka to be able to act as a unified platform for handling all the real-time data feeds [a large company might have](#). To do this we had to think through a fairly broad set of use cases.

It would have to have high-throughput to support high volume event streams such as real-time log aggregation.

It would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.

It also meant the system would have to handle low-latency delivery to handle more traditional messaging use-cases.

We wanted to support partitioned, distributed, real-time processing of these feeds to create new, derived feeds. This motivated our partitioning and consumer model.

Finally in cases where the stream is fed into other data systems for serving, we knew the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

Supporting these uses led us to a design with a number of unique elements, more akin to a database log than a traditional messaging system. We will outline some elements of the design in the following sections.

## **4.2 Persistence**

### **Don't fear the filesystem!**

Kafka relies heavily on the filesystem for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact disks are both much slower and much faster than people expect depending on how they are used; and a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result the performance of linear writes on a [JBOD](#) configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system. A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. A further discussion of this issue can be found in this [ACM Queue article](#); they actually find that [sequential disk access can in some cases be faster than random memory access!](#)

To compensate for this performance divergence, modern operating systems have become increasingly aggressive in their use of main memory for disk caching. A modern OS will happily divert *all* free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. This feature cannot easily be turned off without using direct I/O, so even if a process maintains an in-process cache of the data, this data will likely be duplicated in OS pagecache, effectively storing everything twice.

Furthermore, we are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

1. The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
2. Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure—we at least double the available cache by having automatic access to all free memory, and likely double again by storing a compact byte structure rather than individual objects. Doing so will result in a cache of up to 28-30GB on a 32GB machine without GC penalties.

Furthermore, this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory (which for a 10GB cache may take 10 minutes) or else it will need to start with a completely cold cache (which likely means terrible initial performance). This also greatly simplifies the code as all logic for maintaining coherency between the cache and filesystem is now in the OS, which tends to do so more efficiently and more correctly than one-off in-process attempts. If your disk usage favors linear reads then read-ahead is effectively pre-populating this cache with useful data on each disk read.

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that. All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk. In effect this just means that it is transferred into the kernel's pagecache.

This style of pagecache-centric design is described in an [article](#) on the design of Varnish here (along with a healthy dose of arrogance).

## Constant Time Suffices

The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTTree or other general-purpose random access data structures to maintain metadata about messages. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. They do come with a fairly high cost, though: Btree operations are  $O(\log N)$ . Normally  $O(\log N)$  is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at 10 ms a pop, and each disk can do only one seek at a time so parallelism is limited. Hence even a handful of disk seeks leads to very high overhead. Since storage systems mix very fast cached operations with very slow physical disk operations, the observed performance of tree structures is often superlinear as data increases with fixed cache--i.e. doubling your data makes things much worse than twice as slow.

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions. This structure has the advantage that all operations are  $O(1)$  and reads do not block writes or each other. This has obvious performance advantages since the performance is completely decoupled from the data size—one server can now take full advantage of a number of cheap, low-rotational speed 1+TB SATA drives. Though they have poor seek performance, these drives have acceptable performance for large reads and writes and come at 1/3 the price and 3x the capacity.

Having access to virtually unlimited disk space without any performance penalty means that we can provide some features not usually found in a messaging system. For example, in Kafka, instead of attempting to delete messages as soon as they are consumed, we can retain messages for a relatively long period (say a week). This leads to a great deal of flexibility for consumers, as we will describe.

## 4.3 Efficiency

We have put significant effort into efficiency. One of our primary use cases is handling web activity data, which is very high volume: each page view may generate dozens of writes. Furthermore, we assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.

We have also found, from experience building and running a number of similar systems, that efficiency is a key to effective multi-tenant operations. If the downstream infrastructure service can easily become a bottleneck due to a small bump in usage by the application, such small changes will often create problems. By being very fast we help ensure that the application will tip-over under load before the infrastructure. This is particularly important when trying to run a centralized service that supports dozens or hundreds of applications on a centralized cluster as changes in usage patterns are a near-daily occurrence.

We discussed disk efficiency in the previous section. Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system: too many small I/O operations, and excessive byte copying.

The small I/O problem happens both between the client and the server and in the server's own persistent operations.

To avoid this, our protocol is built around a "message set" abstraction that naturally groups messages together. This allows network requests to group messages together and amortize the overhead of the network roundtrip rather than sending a single message at a time. The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.

This simple optimization produces orders of magnitude speed up. Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on, all of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.

The other inefficiency is in byte copying. At low message rates this is not an issue, but under load the impact is significant. To avoid this we employ a standardized binary message format that is shared by the producer, the broker, and the consumer (so data chunks can be transferred without modification between them).

The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer. Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks. Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the [sendfile system call](#).

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space
2. The application reads the data from kernel space into a user-space buffer
3. The application writes the data back into kernel space into a socket buffer
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network

This is clearly inefficient, there are four copies and two system calls. Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly. So in this optimized path, only the final copy to the NIC buffer is needed.

We expect a common use case to be multiple consumers on a topic. Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption instead of being stored in memory and copied out to user-space every time it is read. This allows messages to be consumed at a rate that approaches the limit of the network connection.

This combination of pagecache and sendfile means that on a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.

TLS/SSL libraries operate at the user space (in-kernel `SSL_sendfile` is currently not supported by Kafka). Due to this restriction, `sendfile` is not used when SSL is enabled. For enabling SSL configuration, refer to `security.protocol` and `security.inter.broker.protocol`

For more background on the sendfile and zero-copy support in Java, see this [article](#).

## End-to-end Batch Compression

In some cases the bottleneck is actually not CPU or disk but network bandwidth. This is particularly true for a data pipeline that needs to send messages between data centers over a wide-area network. Of course, the user can always compress its messages one at a time without any support needed from Kafka, but this can lead to very poor compression ratios as much of the redundancy is due to repetition between messages of the same type (e.g. field names in JSON or user agents in web logs or common string values). Efficient compression requires compressing multiple messages together rather than compressing each message individually.

Kafka supports this with an efficient batching format. A batch of messages can be clumped together compressed and sent to the server in this form. This batch of messages will be written in compressed form and will remain compressed in the log and will only be decompressed by the consumer.

Kafka supports GZIP, Snappy, LZ4 and ZStandard compression protocols. More details on compression can be found [here](#).

## 4.4 The Producer

### Load balancing

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier. To help the producer do this all Kafka nodes can answer a request for metadata about which servers are alive and where the leaders for the partitions of a topic are at any given time to allow the producer to appropriately direct its requests.

The client controls which partition it publishes messages to. This can be done at random, implementing a kind of random load balancing, or it can be done by some semantic partitioning function. We expose the interface for semantic partitioning by allowing the user to specify a key to partition by and using this to hash to a partition (there is also an option to override the partition function if need be). For example if the key chosen was a user id then all data for a given user would be sent to the same partition. This in turn will allow consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in consumers.

### Asynchronous send

Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request. The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say 64k or 10 ms). This allows the accumulation of more bytes to send, and few larger I/O operations on the servers. This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

Details on [configuration](#) and the [api](#) for the producer can be found elsewhere in the documentation.

## 4.5 The Consumer

The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume. The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position. The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

### Push vs. pull

An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as [Scribe](#) and [Apache Flume](#), follow a very different push-based path where data is pushed downstream. There are pros and cons to both approaches. However, a push-based system has difficulty dealing with diverse consumers as the broker controls the rate at which data is transferred. The goal is generally for the consumer to be able to consume at the maximum possible rate; unfortunately, in a push system this means the consumer tends to be overwhelmed when its rate of consumption falls below the rate of production (a denial of service attack, in essence). A pull-based system has the nicer property that the consumer simply falls behind and catches up when it can. This can be mitigated with some kind of backoff protocol by which the consumer can indicate it is overwhelmed, but getting the rate of transfer to fully utilize (but never over-utilize) the consumer is trickier than it seems. Previous attempts at building systems in this fashion led us to go with a more traditional pull model.

Another advantage of a pull-based system is that it lends itself to aggressive batching of data sent to the consumer. A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it. If tuned for low latency, this will result in sending a single message at a time only for the transfer to end up being buffered anyway, which is wasteful. A pull-based design fixes this as the

consumer always pulls all available messages after its current position in the log (or up to some configurable max size). So one gets optimal batching without introducing unnecessary latency.

The deficiency of a naive pull-based system is that if the broker has no data the consumer may end up polling in a tight loop, effectively busy-waiting for data to arrive. To avoid this we have parameters in our pull request that allow the consumer request to block in a "long poll" waiting until data arrives (and optionally waiting until a given number of bytes is available to ensure large transfer sizes).

You could imagine other possible designs which would be only pull, end-to-end. The producer would locally write to a local log, and brokers would pull from that with consumers pulling from them. A similar type of "store-and-forward" producer is often proposed. This is intriguing but we felt not very suitable for our target use cases which have thousands of producers. Our experience running persistent data systems at scale led us to feel that involving thousands of disks in the system across many applications would not actually make things more reliable and would be a nightmare to operate. And in practice we have found that we can run a pipeline with strong SLAs at large scale without a need for producer persistence.

## Consumer Position

Keeping track of *what* has been consumed is, surprisingly, one of the key performance points of a messaging system.

Most messaging systems keep metadata about what messages have been consumed on the broker. That is, as a message is handed out to a consumer, the broker either records that fact locally immediately or it may wait for acknowledgement from the consumer. This is a fairly intuitive choice, and indeed for a single machine server it is not clear where else this state could go. Since the data structures used for storage in many messaging systems scale poorly, this is also a pragmatic choice--since the broker knows what is consumed it can immediately delete it, keeping the data size small.

What is perhaps not obvious is that getting the broker and consumer to come into agreement about what has been consumed is not a trivial problem. If the broker records a message as **consumed** immediately every time it is handed out over the network, then if the consumer fails to process the message (say because it crashes or the request times out or whatever) that message will be lost. To solve this problem, many messaging systems add an acknowledgement feature which means that messages are only marked as **sent** not **consumed** when they are sent; the broker waits for a specific acknowledgement from the consumer to record the message as **consumed**. This strategy fixes the problem of losing messages, but creates new problems. First of all, if the consumer processes the message but fails before it can send an acknowledgement then the message will be consumed twice. The second problem is around performance, now the broker must keep multiple states about every single message (first to lock it so it is not given out a second time, and then to mark it as permanently consumed so that it can be removed). Tricky problems must be dealt with, like what to do with messages that are sent but never acknowledged.

Kafka handles this differently. Our topic is divided into a set of totally ordered partitions, each of which is consumed by exactly one consumer within each subscribing consumer group at any given time. This means that the position of a consumer in each partition is just a single integer, the offset of the next message to consume. This makes the state about what has been consumed very small, just one number for each partition. This state can be periodically checkpointed. This makes the equivalent of message acknowledgements very cheap.

There is a side benefit of this decision. A consumer can deliberately *rewind* back to an old offset and re-consume data. This violates the common contract of a queue, but turns out to be an essential feature for many consumers. For example, if the consumer code has a bug and is discovered after some messages are consumed, the consumer can re-consume those messages once the bug is fixed.

## Offline Data Load

Scalable persistence allows for the possibility of consumers that only periodically consume such as batch data loads that periodically bulk-load data into an offline system such as Hadoop or a relational data warehouse.

In the case of Hadoop we parallelize the data load by splitting the load over individual map tasks, one for each node/topic/partition combination, allowing full parallelism in the loading. Hadoop provides the task management, and tasks which fail can restart without danger of duplicate data—they simply restart from their original position.

## Static Membership

Static membership aims to improve the availability of stream applications, consumer groups and other applications built on top of the group rebalance protocol. The rebalance protocol relies on the group coordinator to allocate entity ids to group members. These generated ids are ephemeral and will change when members restart and rejoin. For consumer based apps, this "dynamic membership" can cause a large percentage of tasks re-assigned to different instances during administrative operations such as code deploys, configuration updates and periodic restarts. For large state applications, shuffled tasks need a long time to recover their local states before processing and cause applications to be partially or entirely unavailable. Motivated by this observation, Kafka's group management protocol allows group members to provide persistent entity ids. Group membership remains unchanged based on those ids, thus no rebalance will be triggered.

If you want to use static membership,

- Upgrade both broker cluster and client apps to 2.3 or beyond, and also make sure the upgraded brokers are using `inter.broker.protocol.version` of 2.3 or beyond as well.
- Set the config `ConsumerConfig#GROUP_INSTANCE_ID_CONFIG` to a unique value for each consumer instance under one group.
- For Kafka Streams applications, it is sufficient to set a unique `ConsumerConfig#GROUP_INSTANCE_ID_CONFIG` per KafkaStreams instance, independent of the number of used threads for an instance.

If your broker is on an older version than 2.3, but you choose to set

`ConsumerConfig#GROUP_INSTANCE_ID_CONFIG` on the client side, the application will detect the broker version and then throws an `UnsupportedException`. If you accidentally configure duplicate ids for different instances, a fencing mechanism on broker side will inform your duplicate client to shutdown immediately by triggering a `org.apache.kafka.common.errors.FencedInstanceIdException`. For more details, see [KIP-345](#)

## 4.6 Message Delivery Semantics

Now that we understand a little about how producers and consumers work, let's discuss the semantic guarantees Kafka provides between producer and consumer. Clearly there are multiple possible message delivery guarantees that could be provided:

- *At most once*—Messages may be lost but are never redelivered.
- *At least once*—Messages are never lost but may be redelivered.
- *Exactly once*—this is what people actually want, each message is delivered once and only once.

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

Many systems claim to provide "exactly once" delivery semantics, but it is important to read the fine print, most of these claims are misleading (i.e. they don't translate to the case where consumers or producers can fail, cases where there are multiple consumer processes, or cases where data written to disk can be lost).

Kafka's semantics are straight-forward. When publishing a message we have a notion of the message being "committed" to the log. Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive". The definition of committed message, alive partition as well as a description of which types of failures we attempt to handle will be described in more detail in the next section. For now let's assume a perfect, lossless broker and try to understand the guarantees to the producer and consumer. If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed. This is similar to the semantics of inserting into a database table with an autogenerated key.

Prior to 0.11.0.0, if a producer failed to receive a response indicating that a message was committed, it had little choice but to resend the message. This provides at-least-once delivery semantics since the message may be written to the log again during resending if the original request had in fact succeeded. Since 0.11.0.0, the Kafka producer also supports an idempotent delivery option which guarantees that resending will not result in duplicate entries in the log. To achieve this, the broker assigns each producer an ID and deduplicates messages using a sequence number that is sent by the producer along with every message. Also beginning with 0.11.0.0, the producer supports the ability to send messages to multiple topic partitions using transaction-like semantics: i.e. either all messages are successfully written or none of them are. The main use case for this is exactly-once processing between Kafka topics (described below).

Not all use cases require such strong guarantees. For uses which are latency sensitive we allow the producer to specify the durability level it desires. If the producer specifies that it wants to wait on the message being committed this can take on the order of 10 ms. However the producer can also specify that it wants to perform the send completely asynchronously or that it wants to wait only until the leader (but not necessarily the followers) have the message.

Now let's describe the semantics from the point-of-view of the consumer. All replicas have the exact same log with the same offsets. The consumer controls its position in this log. If the consumer never crashed it could just store this position in memory, but if the consumer fails and we want this topic partition to be taken over by another process the new process will need to choose an appropriate position from which to start processing. Let's say the consumer reads some messages -- it has several options for processing the messages and updating its position.

1. It can read the messages, then save its position in the log, and finally process the messages. In this case there is a possibility that the consumer process crashes after saving its position but before saving the output of its message processing. In this case the process that took over processing would start at the saved position even though a few messages prior to that position had not been processed. This corresponds to "at-most-once" semantics as in the case of a consumer failure messages may not be processed.
2. It can read the messages, process the messages, and finally save its position. In this case there is a possibility that the consumer process crashes after processing messages but before saving its position. In this case when the new process takes over the first few messages it receives will already have been processed. This corresponds to the "at-least-once" semantics in the case of consumer failure. In many cases messages have a primary key and so the updates are idempotent (receiving the same message twice just overwrites a record with another copy of itself).

So what about exactly once semantics (i.e. the thing you actually want)? When consuming from a Kafka topic and producing to another topic (as in a [Kafka Streams](#) application), we can leverage the new transactional producer capabilities in 0.11.0.0 that were mentioned above. The consumer's position is stored as a message in a topic, so we can write the offset to Kafka in the same transaction as the output topics receiving the processed data. If the transaction is aborted, the consumer's position will revert to its old value and the produced data on the output topics will not be visible to other consumers, depending on their "isolation level." In the default "read\_uncommitted" isolation level, all messages are visible to consumers even if they were part of an aborted transaction, but in "read\_committed," the consumer will only return messages from transactions which were committed (and any messages which were not part of a transaction).

When writing to an external system, the limitation is in the need to coordinate the consumer's position with what is actually stored as output. The classic way of achieving this would be to introduce a two-phase commit between the storage of the consumer position and the storage of the consumers output. But this can be handled more simply and generally by letting the consumer store its offset in the same place as its output. This is better because many of the output systems a consumer might want to write to will not support a two-phase commit. As an example of this, consider a [Kafka Connect](#) connector which populates data in HDFS along with the offsets of the data it reads so that it is guaranteed that either data and offsets are both updated or neither is. We follow similar patterns for many other data systems which require these stronger semantics and for which the messages do not have a primary key to allow for deduplication.

So effectively Kafka supports exactly-once delivery in [Kafka Streams](#), and the transactional producer/consumer can be used generally to provide exactly-once delivery when transferring and processing data between Kafka topics. Exactly-once delivery for other destination systems generally requires cooperation with such systems, but Kafka provides the offset which makes implementing this feasible (see also [Kafka Connect](#)). Otherwise, Kafka guarantees at-least-once delivery by default, and allows the user to implement at-most-once delivery by disabling retries on the producer and committing offsets in the consumer prior to processing a batch of messages.

## 4.7 Replication

Kafka replicates the log for each topic's partitions across a configurable number of servers (you can set this replication factor on a topic-by-topic basis). This allows automatic failover to these replicas when a server in the cluster fails so messages remain available in the presence of failures.

Other messaging systems provide some replication-related features, but, in our (totally biased) opinion, this appears to be a tacked-on thing, not heavily used, and with large downsides: replicas are inactive, throughput is heavily impacted, it requires fiddly manual configuration, etc. Kafka is meant to be used with replication by default—in fact we implement un-replicated topics as replicated topics where the replication factor is one.

The unit of replication is the topic partition. Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers. The total number of replicas including the leader constitute the replication factor. All writes go to the leader of the partition, and reads can go to the leader or the followers of the partition. Typically, there are many more partitions than brokers and the leaders are evenly distributed among brokers. The logs on the followers are identical to the leader's log—all have the same offsets and messages in the same order (though, of course, at any given time the leader may have a few as-yet unreplicated messages at the end of its log).

Followers consume messages from the leader just as a normal Kafka consumer would and apply them to their own log. Having the followers pull from the leader has the nice property of allowing the follower to naturally batch together log entries they are applying to their log.

As with most distributed systems, automatically handling failures requires a precise definition of what it means for a node to be "alive." In Kafka, a special node known as the "controller" is responsible for managing the registration of brokers in the cluster. Broker liveness has two conditions:

1. Brokers must maintain an active session with the controller in order to receive regular metadata updates.
2. Brokers acting as followers must replicate the writes from the leader and not fall "too far" behind.

What is meant by an "active session" depends on the cluster configuration. For KRaft clusters, an active session is maintained by sending periodic heartbeats to the controller. If the controller fails to receive a heartbeat before the timeout configured by `broker.session.timeout.ms` expires, then the node is considered offline.

For clusters using Zookeeper, liveness is determined indirectly through the existence of an ephemeral node which is created by the broker on initialization of its Zookeeper session. If the broker loses its session after failing to send heartbeats to Zookeeper before expiration of `zookeeper.session.timeout.ms`, then the node gets deleted. The controller would then notice the node deletion through a Zookeeper watch and mark the broker offline.

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed". The leader keeps track of the set of "in sync" replicas, which is known as the ISR. If either of these conditions fail to be satisfied, then the broker will be removed from the ISR. For example, if a follower dies, then the controller will notice the failure through the loss of its session, and will remove the broker from the ISR. On the other hand, if the follower lags too far behind the leader but still has an active session, then the leader can also remove it from the ISR. The determination of lagging replicas is controlled through the `replica.lag.time.max.ms` configuration. Replicas that cannot catch up to the end of the log on the leader within the max time set by this configuration are removed from the ISR.

In distributed systems terminology we only attempt to handle a "fail/recover" model of failures where nodes suddenly cease working and then later recover (perhaps without knowing that they have died). Kafka does not handle so-called "Byzantine" failures in which nodes produce arbitrary or malicious responses (perhaps due to bugs or foul play).

We can now more precisely define that a message is considered committed when all replicas in the ISR for that partition have applied it to their log. Only committed messages are ever given out to the consumer. This means that the consumer need not worry about potentially seeing a message that could be lost if the leader fails. Producers, on the other hand, have the option of either waiting for the message to be committed or not, depending on their preference for tradeoff between latency and durability. This preference is controlled by the `acks` setting that the producer uses. Note that topics have a setting for the "minimum number" of in-sync replicas that is checked when the producer requests acknowledgment that a message has been written to the full set of in-sync replicas. If a less stringent acknowledgement is requested by the producer, then the message can be committed, and consumed, even if the number of in-sync replicas is lower than the minimum (e.g. it can be as low as just the leader).

The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.

Kafka will remain available in the presence of node failures after a short fail-over period, but may not remain available in the presence of network partitions.

## [Replicated Logs: Quorums, ISRs, and State Machines \(Oh my!\)](#)

At its heart a Kafka partition is a replicated log. The replicated log is one of the most basic primitives in distributed data systems, and there are many approaches for implementing one. A replicated log can be used by other systems as a primitive for implementing other distributed systems in the [state-machine style](#).

A replicated log models the process of coming into consensus on the order of a series of values (generally numbering the log entries 0, 1, 2, ...). There are many ways to implement this, but the simplest and fastest is with a leader who chooses the ordering of values provided to it. As long as the leader remains alive, all followers need to only copy the values and ordering the leader chooses.

Of course if leaders didn't fail we wouldn't need followers! When the leader does die we need to choose a new leader from among the followers. But followers themselves may fall behind or crash so we must ensure we choose an up-to-date follower. The fundamental guarantee a log replication algorithm must provide is that if we tell the client a message is committed, and the leader fails, the new leader we elect must also have that message. This yields a tradeoff: if the leader waits for more followers to acknowledge a message before declaring it committed then there will be more potentially electable leaders.

If you choose the number of acknowledgements required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap, then this is called a Quorum.

A common approach to this tradeoff is to use a majority vote for both the commit decision and the leader election. This is not what Kafka does, but let's explore it anyway to understand the tradeoffs. Let's say we have  $2f+1$  replicas. If  $f+1$  replicas must receive a message prior to a commit being declared by the leader, and if we elect a new leader by electing the follower with the most complete log from at least  $f+1$  replicas, then, with no more than  $f$  failures, the leader is guaranteed to have all committed messages. This is because among any  $f+1$  replicas, there must be at least one replica that contains all committed messages. That replica's log will be the most complete and therefore will be selected as the new leader. There are many remaining details that each algorithm must handle (such as precisely defined what makes a log more complete, ensuring log consistency during leader failure or changing the set of servers in the replica set) but we will ignore these for now.

This majority vote approach has a very nice property: the latency is dependent on only the fastest servers. That is, if the replication factor is three, the latency is determined by the faster follower not the slower one.

There are a rich variety of algorithms in this family including ZooKeeper's [Zab](#), [Raft](#), and [Viewstamped Replication](#). The most similar academic publication we are aware of to Kafka's actual implementation is [PacificA](#) from Microsoft.

The downside of majority vote is that it doesn't take many failures to leave you with no electable leaders. To tolerate one failure requires three copies of the data, and to tolerate two failures requires five copies of the data. In our experience having only enough redundancy to tolerate a single failure is not enough for a practical system, but doing every write five times, with 5x the disk space requirements and 1/5th the throughput, is not very practical for large volume data problems. This is likely why quorum algorithms more commonly appear for shared cluster configuration such as ZooKeeper but are less common for primary data storage. For example in HDFS the namenode's high-availability feature is built on a [majority-vote-based journal](#), but this more expensive approach is not used for the data itself.

Kafka takes a slightly different approach to choosing its quorum set. Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader. Only members of this set are eligible for election as leader. A write to a Kafka partition is not considered committed until *all* in-sync replicas have received the write. This ISR set is persisted in the cluster metadata whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important. With this ISR model and  $f+1$  replicas, a Kafka topic can tolerate  $f$  failures without losing committed messages.

For most use cases we hope to handle, we think this tradeoff is a reasonable one. In practice, to tolerate  $f$  failures, both the majority vote and the ISR approach will wait for the same number of replicas to acknowledge before committing a message (e.g. to survive one failure a majority quorum needs three replicas and one acknowledgement and the ISR approach requires two replicas and one acknowledgement). The ability to commit without the slowest servers is an advantage of the majority vote approach. However, we think it is ameliorated by allowing the client to choose whether they block on the message commit or not, and the additional throughput and disk space due to the lower required replication factor is worth it.

Another important design distinction is that Kafka does not require that crashed nodes recover with all their data intact. It is not uncommon for replication algorithms in this space to depend on the existence of "stable storage" that cannot be lost in any failure-recovery scenario without potential consistency violations. There are two primary problems with this assumption. First, disk errors are the most common problem we observe in real operation of persistent data systems and they often do not leave data intact. Secondly, even if this were not a problem, we do not want to require the use of fsync on every write for our consistency guarantees as this can reduce performance by two to three orders of magnitude. Our protocol for allowing a replica to rejoin the ISR ensures that before rejoining, it must fully re-sync again even if it lost unflushed data in its crash.

## Unclean leader election: What if they all die?

Note that Kafka's guarantee with respect to data loss is predicated on at least one replica remaining in sync. If all the nodes replicating a partition die, this guarantee no longer holds.

However a practical system needs to do something reasonable when all the replicas die. If you are unlucky enough to have this occur, it is important to consider what will happen. There are two behaviors that could be implemented:

1. Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).
2. Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

This is a simple tradeoff between availability and consistency. If we wait for replicas in the ISR, then we will remain unavailable as long as those replicas are down. If such replicas were destroyed or their data was lost, then we are permanently down. If, on the other hand, a non-in-sync replica comes back to life and we allow it to become leader, then its log becomes the source of truth even though it is not guaranteed to have every committed message. By default from version 0.11.0.0, Kafka chooses the first strategy and favor waiting for a consistent replica. This behavior can be changed using configuration property `unclean.leader.election.enable`, to support use cases where uptime is preferable to consistency.

This dilemma is not specific to Kafka. It exists in any quorum-based scheme. For example in a majority voting scheme, if a majority of servers suffer a permanent failure, then you must either choose to lose 100% of your data or violate consistency by taking what remains on an existing server as your new source of truth.

## Availability and Durability Guarantees

When writing to Kafka, producers can choose whether they wait for the message to be acknowledged by 0, 1 or all (-1) replicas. Note that "acknowledgement by all replicas" does not guarantee that the full set of assigned replicas have received the message. By default, when `acks=all`, acknowledgement happens as soon as all the current in-sync replicas have received the message. For example, if a topic is configured with only two replicas and one fails (i.e., only one in sync replica remains), then writes that specify `acks=all` will succeed. However, these writes could be lost if the remaining replica also fails. Although this ensures maximum availability of the partition, this behavior may be undesirable to some users who prefer durability over availability. Therefore, we provide two topic-level configurations that can be used to prefer message durability over availability:

1. Disable unclean leader election - if all replicas become unavailable, then the partition will remain unavailable until the most recent leader becomes available again. This effectively prefers unavailability over the risk of message loss. See the previous section on Unclean Leader Election for clarification.
2. Specify a minimum ISR size - the partition will only accept writes if the size of the ISR is above a certain minimum, in order to prevent the loss of messages that were written to just a single replica, which subsequently becomes unavailable. This setting only takes effect if the producer uses `acks=all` and guarantees that the message will be acknowledged by at least this many in-sync replicas. This setting offers a trade-off between consistency and availability. A higher setting for minimum ISR size guarantees better consistency since the message is guaranteed to be written to more replicas which reduces the probability that it will be lost. However, it reduces availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum threshold.

## Replica Management

The above discussion on replicated logs really covers only a single log, i.e. one topic partition. However a Kafka cluster will manage hundreds or thousands of these partitions. We attempt to balance partitions within a cluster in a round-robin fashion to avoid clustering all partitions for high-volume topics on a small number of nodes. Likewise we try to balance leadership so that each node is the leader for a proportional share of its partitions.

It is also important to optimize the leadership election process as that is the critical window of unavailability. A naive implementation of leader election would end up running an election per partition for all partitions a node hosted when that node failed. As discussed above in the section on [replication](#), Kafka clusters have a special role known as the "controller" which is responsible for managing the registration of brokers. If the controller detects the failure of a broker, it is responsible for electing one of the remaining members of the ISR to serve as the new leader. The result is that we are able to batch together many of the required leadership change notifications which makes the election process far cheaper and faster for a large number of partitions. If the controller itself fails, then another controller will be elected.

## [4.8 Log Compaction](#)

Log compaction ensures that Kafka will always retain at least the last known value for each message key within the log of data for a single topic partition. It addresses use cases and scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance. Let's dive into these use cases in more detail and then describe how compaction works.

So far we have described only the simpler approach to data retention where old log data is discarded after a fixed period of time or when the log reaches some predetermined size. This works well for temporal event data such as logging where each record stands alone. However an important class of data streams are the log of changes to keyed, mutable data (for example, the changes to a database table).

Let's discuss a concrete example of such a stream. Say we have a topic containing user email addresses; every time a user updates their email address we send a message to this topic using their user id as the primary key. Now say we send the following messages over some time period for a user with id 123, each message corresponding to a change in email address (messages for other ids are omitted):

```
123 => bill@microsoft.com  
.  
.  
.  
123 => bill@gatesfoundation.org  
.  
.  
.  
123 => bill@gmail.com
```

Log compaction gives us a more granular retention mechanism so that we are guaranteed to retain at least the last update for each primary key (e.g. `bill@gmail.com`). By doing this we guarantee that the log contains a full snapshot of the final value for every key not just keys that changed recently. This means downstream consumers can restore their own state off this topic without us having to retain a complete log of all changes.

Let's start by looking at a few use cases where this is useful, then we'll see how it can be used.

1. *Database change subscription.* It is often necessary to have a data set in multiple data systems, and often one of these systems is a database of some kind (either a RDBMS or perhaps a new-fangled key-value store). For example you might have a database, a cache, a search cluster, and a Hadoop cluster. Each change to the database will need to be reflected in the cache, the search cluster, and eventually in Hadoop. In the case that one is only handling the real-time updates you only need recent log. But if you want to be able to reload the cache or restore a failed search node you may need a complete data set.
2. *Event sourcing.* This is a style of application design which co-locates query processing with application design and uses a log of changes as the primary store for the application.
3. *Journaling for high-availability.* A process that does local computation can be made fault-tolerant by logging out changes that it makes to its local state so another process can reload these changes and

carry on if it should fail. A concrete example of this is handling counts, aggregations, and other "group by"-like processing in a stream query system. Samza, a real-time stream-processing framework, [uses this feature](#) for exactly this purpose.

In each of these cases one needs primarily to handle the real-time feed of changes, but occasionally, when a machine crashes or data needs to be re-loaded or re-processed, one needs to do a full load. Log compaction allows feeding both of these use cases off the same backing topic. This style of usage of a log is described in more detail in [this blog post](#).

The general idea is quite simple. If we had infinite log retention, and we logged each change in the above cases, then we would have captured the state of the system at each time from when it first began. Using this complete log, we could restore to any point in time by replaying the first N records in the log. This hypothetical complete log is not very practical for systems that update a single record many times as the log will grow without bound even for a stable dataset. The simple log retention mechanism which throws away old updates will bound space but the log is no longer a way to restore the current state—now restoring from the beginning of the log no longer recreates the current state as old updates may not be captured at all.

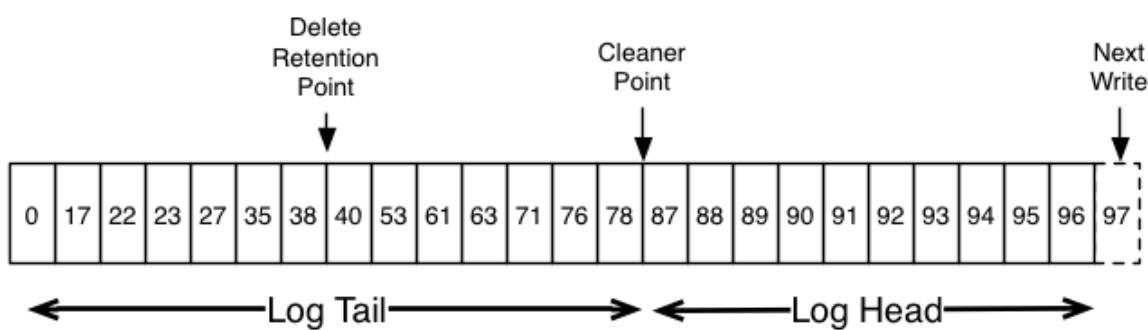
Log compaction is a mechanism to give finer-grained per-record retention, rather than the coarser-grained time-based retention. The idea is to selectively remove records where we have a more recent update with the same primary key. This way the log is guaranteed to have at least the last state for each key.

This retention policy can be set per-topic, so a single cluster can have some topics where retention is enforced by size or time and other topics where retention is enforced by compaction.

This functionality is inspired by one of LinkedIn's oldest and most successful pieces of infrastructure—a database changelog caching service called [Databus](#). Unlike most log-structured storage systems Kafka is built for subscription and organizes data for fast linear reads and writes. Unlike Databus, Kafka acts as a source-of-truth store so it is useful even in situations where the upstream data source would not otherwise be replayable.

## [Log Compaction Basics](#)

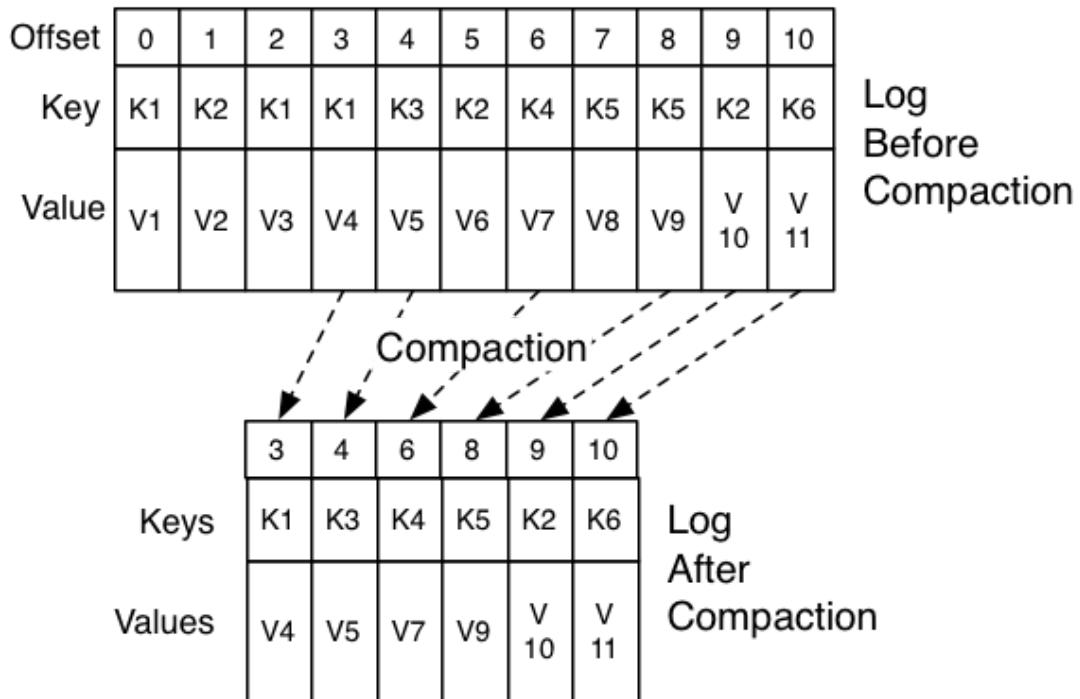
Here is a high-level picture that shows the logical structure of a Kafka log with the offset for each message.



The head of the log is identical to a traditional Kafka log. It has dense, sequential offsets and retains all messages. Log compaction adds an option for handling the tail of the log. The picture above shows a log with a compacted tail. Note that the messages in the tail of the log retain the original offset assigned when they were first written—that never changes. Note also that all offsets remain valid positions in the log, even if the message with that offset has been compacted away; in this case this position is indistinguishable from the next highest offset that does appear in the log. For example, in the picture above the offsets 36, 37, and 38 are all equivalent positions and a read beginning at any of these offsets would return a message set beginning with 38.

Compaction also allows for deletes. A message with a key and a null payload will be treated as a delete from the log. Such a record is sometimes referred to as a *tombstone*. This delete marker will cause any prior message with that key to be removed (as would any new message with that key), but delete markers are special in that they will themselves be cleaned out of the log after a period of time to free up space. The point in time at which deletes are no longer retained is marked as the "delete retention point" in the above diagram.

The compaction is done in the background by periodically recopying log segments. Cleaning does not block reads and can be throttled to use no more than a configurable amount of I/O throughput to avoid impacting producers and consumers. The actual process of compacting a log segment looks something like this:



## What guarantees does log compaction provide?

Log compaction guarantees the following:

1. Any consumer that stays caught-up to within the head of the log will see every message that is written; these messages will have sequential offsets. The topic's `min.compaction.lag.ms` can be used to guarantee the minimum length of time must pass after a message is written before it could be compacted. I.e. it provides a lower bound on how long each message will remain in the (uncompacted) head. The topic's `max.compaction.lag.ms` can be used to guarantee the maximum delay between the time a message is written and the time the message becomes eligible for compaction.
2. Ordering of messages is always maintained. Compaction will never re-order messages, just remove some.
3. The offset for a message never changes. It is the permanent identifier for a position in the log.
4. Any consumer progressing from the start of the log will see at least the final state of all records in the order they were written. Additionally, all delete markers for deleted records will be seen, provided the consumer reaches the head of the log in a time period less than the topic's `delete.retention.ms` setting (the default is 24 hours). In other words: since the removal of delete markers happens concurrently with reads, it is possible for a consumer to miss delete markers if it lags by more than `delete.retention.ms`.

## Log Compaction Details

Log compaction is handled by the log cleaner, a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Each compactor thread works as follows:

1. It chooses the log that has the highest ratio of log head to log tail
2. It creates a succinct summary of the last offset for each key in the head of the log
3. It recopies the log from beginning to end removing keys which have a later occurrence in the log.  
New, clean segments are swapped into the log immediately so the additional disk space required is just one additional log segment (not a fully copy of the log).
4. The summary of the log head is essentially just a space-compact hash table. It uses exactly 24 bytes per entry. As a result with 8GB of cleaner buffer one cleaner iteration can clean around 366GB of log head (assuming 1k messages).

## Configuring The Log Cleaner

The log cleaner is enabled by default. This will start the pool of cleaner threads. To enable log cleaning on a particular topic, add the log-specific property

```
log.cleanup.policy=compact
```

The `log.cleanup.policy` property is a broker configuration setting defined in the broker's `server.properties` file; it affects all of the topics in the cluster that do not have a configuration override in place as documented [here](#). The log cleaner can be configured to retain a minimum amount of the uncompacted "head" of the log. This is enabled by setting the compaction time lag.

```
log.cleaner.min.compaction.lag.ms
```

This can be used to prevent messages newer than a minimum message age from being subject to compaction. If not set, all log segments are eligible for compaction except for the last segment, i.e. the one currently being written to. The active segment will not be compacted even if all of its messages are older than the minimum compaction time lag. The log cleaner can be configured to ensure a maximum delay after which the uncompacted "head" of the log becomes eligible for log compaction.

```
log.cleaner.max.compaction.lag.ms
```

This can be used to prevent log with low produce rate from remaining ineligible for compaction for an unbounded duration. If not set, logs that do not exceed `min.cleanable.dirty.ratio` are not compacted. Note that this compaction deadline is not a hard guarantee since it is still subjected to the availability of log cleaner threads and the actual compaction time. You will want to monitor the `uncleanable-partitions-count`, `max-clean-time-secs` and `max-compaction-delay-secs` metrics.

Further cleaner configurations are described [here](#).

## 4.9 Quotas

Kafka cluster has the ability to enforce quotas on requests to control the broker resources used by clients. Two types of client quotas can be enforced by Kafka brokers for each group of clients sharing a quota:

1. Network bandwidth quotas define byte-rate thresholds (since 0.9)
2. Request rate quotas define CPU utilization thresholds as a percentage of network and I/O threads (since 0.11)

## Why are quotas necessary?

It is possible for producers and consumers to produce/consume very high volumes of data or generate requests at a very high rate and thus monopolize broker resources, cause network saturation and generally DOS other clients and the brokers themselves. Having quotas protects against these issues and is all the more important in large multi-tenant clusters where a small set of badly behaved clients can degrade user experience for the well behaved ones. In fact, when running Kafka as a service this even makes it possible to enforce API limits according to an agreed upon contract.

## Client groups

The identity of Kafka clients is the user principal which represents an authenticated user in a secure cluster. In a cluster that supports unauthenticated clients, user principal is a grouping of unauthenticated users chosen by the broker using a configurable `PrincipalBuilder`. Client-id is a logical grouping of clients with a meaningful name chosen by the client application. The tuple (user, client-id) defines a secure logical group of clients that share both user principal and client-id.

Quotas can be applied to (user, client-id), user or client-id groups. For a given connection, the most specific quota matching the connection is applied. All connections of a quota group share the quota configured for the group. For example, if (user="test-user", client-id="test-client") has a produce quota of 10MB/sec, this is shared across all producer instances of user "test-user" with the client-id "test-client".

## Quota Configuration

Quota configuration may be defined for (user, client-id), user and client-id groups. It is possible to override the default quota at any of the quota levels that needs a higher (or even lower) quota. The mechanism is similar to the per-topic log config overrides. User and (user, client-id) quota overrides are written to ZooKeeper under `/config/users` and client-id quota overrides are written under `/config/clients`. These overrides are read by all brokers and are effective immediately. This lets us change quotas without having to do a rolling restart of the entire cluster. See [here](#) for details. Default quotas for each group may also be updated dynamically using the same mechanism.

The order of precedence for quota configuration is:

1. `/config/users//clients/`
2. `/config/users//clients/`
3. `/config/users/`
4. `/config/users//clients/`
5. `/config/users//clients/`
6. `/config/users/`
7. `/config/clients/`
8. `/config/clients/`

## Network Bandwidth Quotas

Network bandwidth quotas are defined as the byte rate threshold for each group of clients sharing a quota. By default, each unique client group receives a fixed quota in bytes/sec as configured by the cluster. This quota is defined on a per-broker basis. Each group of clients can publish/fetch a maximum of X bytes/sec per broker before clients are throttled.

## Request Rate Quotas

Request rate quotas are defined as the percentage of time a client can utilize on request handler I/O threads and network threads of each broker within a quota window. A quota of `n%` represents `n%` of one thread, so the quota is out of a total capacity of `((num.io.threads + num.network.threads) * 100)%`. Each group of clients may use a total percentage of upto `n%` across all I/O and network threads in a quota window before being throttled. Since the number of threads allocated for I/O and network threads are typically based on the number of cores available on the broker host, request rate quotas represent the total percentage of CPU that may be used by each group of clients sharing the quota.

## Enforcement

By default, each unique client group receives a fixed quota as configured by the cluster. This quota is defined on a per-broker basis. Each client can utilize this quota per broker before it gets throttled. We decided that defining these quotas per broker is much better than having a fixed cluster wide bandwidth per client because that would require a mechanism to share client quota usage among all the brokers. This can be harder to get right than the quota implementation itself!

How does a broker react when it detects a quota violation? In our solution, the broker first computes the amount of delay needed to bring the violating client under its quota and returns a response with the delay immediately. In case of a fetch request, the response will not contain any data. Then, the broker mutes the channel to the client, not to process requests from the client anymore, until the delay is over. Upon receiving a response with a non-zero delay duration, the Kafka client will also refrain from sending further requests to the broker during the delay. Therefore, requests from a throttled client are effectively blocked from both sides. Even with older client implementations that do not respect the delay response from the broker, the back pressure applied by the broker via muting its socket channel can still handle the throttling of badly behaving clients. Those clients who sent further requests to the throttled channel will receive responses only after the delay is over.

Byte-rate and thread utilization are measured over multiple small windows (e.g. 30 windows of 1 second each) in order to detect and correct quota violations quickly. Typically, having large measurement windows (for e.g. 10 windows of 30 seconds each) leads to large bursts of traffic followed by long delays which is not great in terms of user experience.

## 5. IMPLEMENTATION

---

### 5.1 Network Layer

The network layer is a fairly straight-forward NIO server, and will not be described in great detail. The sendfile implementation is done by giving the `MessageSet` interface a `writeTo` method. This allows the file-backed message set to use the more efficient `transferTo` implementation instead of an in-process buffered write. The threading model is a single acceptor thread and  $N$  processor threads which handle a fixed number of connections each. This design has been pretty thoroughly tested [elsewhere](#) and found to be simple to implement and fast. The protocol is kept quite simple to allow for future implementation of clients in other languages.

### 5.2 Messages

Messages consist of a variable-length header, a variable-length opaque key byte array and a variable-length opaque value byte array. The format of the header is described in the following section. Leaving the key and value opaque is the right decision: there is a great deal of progress being made on serialization libraries right now, and any particular choice is unlikely to be right for all uses. Needless to say a particular application using Kafka would likely mandate a particular serialization type as part of its usage. The `RecordBatch` interface is simply an iterator over messages with specialized methods for bulk reading and writing to an NIO `channel`.

### 5.3 Message Format

Messages (aka Records) are always written in batches. The technical term for a batch of messages is a record batch, and a record batch contains one or more records. In the degenerate case, we could have a record batch containing a single record. Record batches and records have their own headers. The format of each is described below.

### 5.3.1 Record Batch

The following is the on-disk format of a RecordBatch.

```
baseOffset: int64
batchLength: int32
partitionLeaderEpoch: int32
magic: int8 (current magic value is 2)
crc: int32
attributes: int16
    bit 0~2:
        0: no compression
        1: gzip
        2: snappy
        3: lz4
        4: zstd
    bit 3: timestampType
    bit 4: isTransactional (0 means not transactional)
    bit 5: isControlBatch (0 means not a control batch)
    bit 6: hasDeleteHorizonMs (0 means baseTimestamp is not set as the delete horizon
for compaction)
    bit 7~15: unused
lastOffsetDelta: int32
baseTimestamp: int64
maxTimestamp: int64
producerId: int64
producerEpoch: int16
baseSequence: int32
records: [Record]
```

Note that when compression is enabled, the compressed record data is serialized directly following the count of the number of records.

The CRC covers the data from the attributes to the end of the batch (i.e. all the bytes that follow the CRC). It is located after the magic byte, which means that clients must parse the magic byte before deciding how to interpret the bytes between the batch length and the magic byte. The partition leader epoch field is not included in the CRC computation to avoid the need to recompute the CRC when this field is assigned for every batch that is received by the broker. The CRC-32C (Castagnoli) polynomial is used for the computation.

On compaction: unlike the older message formats, magic v2 and above preserves the first and last offset/sequence numbers from the original batch when the log is cleaned. This is required in order to be able to restore the producer's state when the log is reloaded. If we did not retain the last sequence number, for example, then after a partition leader failure, the producer might see an OutOfSequence error. The base sequence number must be preserved for duplicate checking (the broker checks incoming Produce requests for duplicates by verifying that the first and last sequence numbers of the incoming batch match the last from that producer). As a result, it is possible to have empty batches in the log when all the records in the batch are cleaned but batch is still retained in order to preserve a producer's last sequence number. One oddity here is that the baseTimestamp field is not preserved during compaction, so it will change if the first record in the batch is compacted away.

Compaction may also modify the baseTimestamp if the record batch contains records with a null payload or aborted transaction markers. The baseTimestamp will be set to the timestamp of when those records should be deleted with the delete horizon attribute bit also set.

### [5.3.1.1 Control Batches](#)

A control batch contains a single record called the control record. Control records should not be passed on to applications. Instead, they are used by consumers to filter out aborted transactional messages.

The key of a control record conforms to the following schema:

```
version: int16 (current version is 0)
type: int16 (0 indicates an abort marker, 1 indicates a commit)
```

The schema for the value of a control record is dependent on the type. The value is opaque to clients.

### [5.3.2 Record](#)

Record level headers were introduced in Kafka 0.11.0. The on-disk format of a record with Headers is delineated below.

```
length: varint
attributes: int8
    bit 0~7: unused
timestampDelta: varlong
offsetDelta: varint
keyLength: varint
key: byte[]
valueLen: varint
value: byte[]
Headers => [Header]
```

#### [5.3.2.1 Record Header](#)

```
headerKeyLength: varint
headerKey: String
headerValueLength: varint
value: byte[]
```

We use the same varint encoding as Protobuf. More information on the latter can be found [here](#). The count of headers in a record is also encoded as a varint.

### [5.3.3 Old Message Format](#)

Prior to Kafka 0.11, messages were transferred and stored in *message sets*. In a message set, each message has its own metadata. Note that although message sets are represented as an array, they are not preceded by an int32 array size like other array elements in the protocol.

#### **Message Set:**

```
MessageSet (Version: 0) => [offset message_size message]
offset => INT64
message_size => INT32
message => crc magic_byte attributes key value
    crc => INT32
    magic_byte => INT8
    attributes => INT8
        bit 0~2:
            0: no compression
            1: gzip
            2: snappy
        bit 3~7: unused
    key => BYTES
```

```

value => BYTES
MessageSet (Version: 1) => [offset message_size message]
offset => INT64
message_size => INT32
message => crc magic_byte attributes timestamp key value
    crc => INT32
    magic_byte => INT8
    attributes => INT8
        bit 0~2:
            0: no compression
            1: gzip
            2: snappy
            3: lz4
        bit 3: timestampType
            0: create time
            1: log append time
        bit 4~7: unused
    timestamp => INT64
    key => BYTES
    value => BYTES

```

In versions prior to Kafka 0.10, the only supported message format version (which is indicated in the magic value) was 0. Message format version 1 was introduced with timestamp support in version 0.10.

- Similarly to version 2 above, the lowest bits of attributes represent the compression type.
- In version 1, the producer should always set the timestamp type bit to 0. If the topic is configured to use log append time, (through either broker level config `log.message.timestamp.type = LogAppendTime` or topic level config `message.timestamp.type = LogAppendTime`), the broker will overwrite the timestamp type and the timestamp in the message set.
- The highest bits of attributes must be set to 0.

In message format versions 0 and 1 Kafka supports recursive messages to enable compression. In this case the message's attributes must be set to indicate one of the compression types and the value field will contain a message set compressed with that type. We often refer to the nested messages as "inner messages" and the wrapping message as the "outer message." Note that the key should be null for the outer message and its offset will be the offset of the last inner message.

When receiving recursive version 0 messages, the broker decompresses them and each inner message is assigned an offset individually. In version 1, to avoid server side re-compression, only the wrapper message will be assigned an offset. The inner messages will have relative offsets. The absolute offset can be computed using the offset from the outer message, which corresponds to the offset assigned to the last inner message.

The crc field contains the CRC32 (and not CRC-32C) of the subsequent message bytes (i.e. from magic byte to the value).

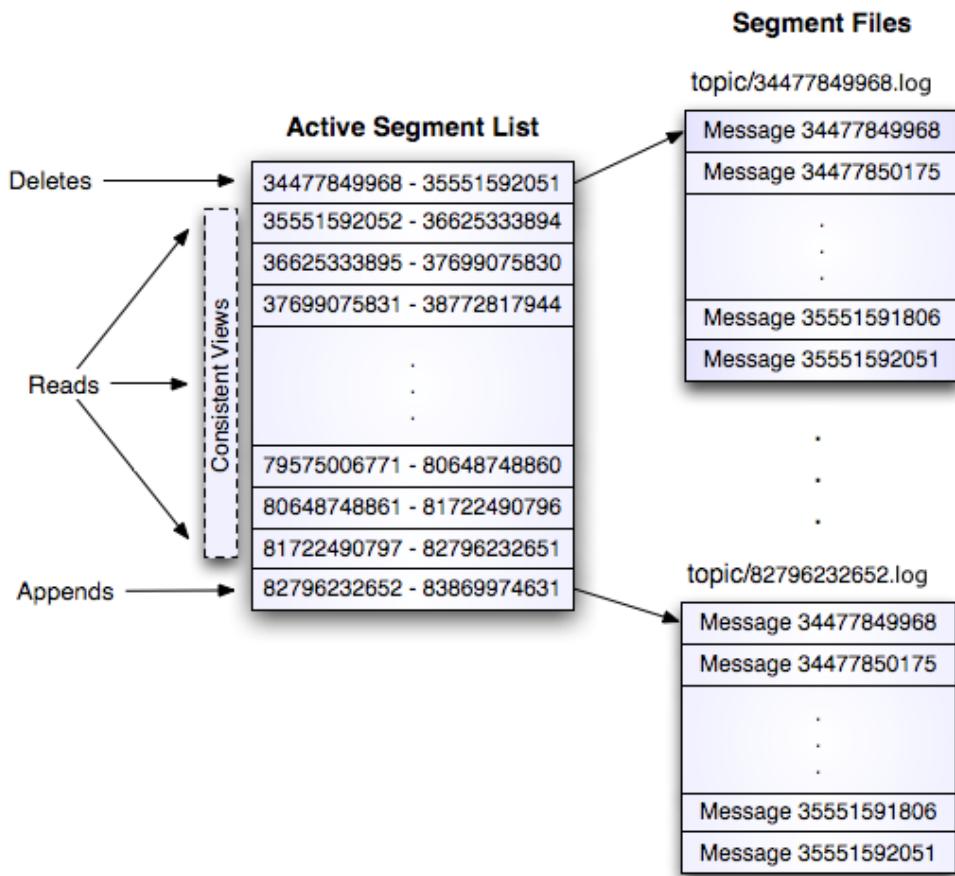
## [5.4 Log](#)

A log for a topic named "my-topic" with two partitions consists of two directories (namely `my-topic-0` and `my-topic-1`) populated with data files containing the messages for that topic. The format of the log files is a sequence of "log entries"; each log entry is a 4 byte integer  $N$  storing the message length which is followed by the  $N$  message bytes. Each message is uniquely identified by a 64-bit integer  $offset$  giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition. The on-disk format of each message is given below. Each log file is named with the offset of the first message it contains. So the first file created will be `00000000000000000000000000000000.log`, and each additional file will have an integer name roughly  $S$  bytes from the previous file where  $S$  is the max log file size given in the configuration.

The exact binary format for records is versioned and maintained as a standard interface so record batches can be transferred between producer, broker, and client without recopying or conversion when desirable. The previous section included details about the on-disk format of records.

The use of the message offset as the message id is unusual. Our original idea was to use a GUID generated by the producer, and maintain a mapping from GUID to offset on each broker. But since a consumer must maintain an ID for each server, the global uniqueness of the GUID provides no value. Furthermore, the complexity of maintaining the mapping from a random id to an offset requires a heavy weight index structure which must be synchronized with disk, essentially requiring a full persistent random-access data structure. Thus to simplify the lookup structure we decided to use a simple per-partition atomic counter which could be coupled with the partition id and node id to uniquely identify a message; this makes the lookup structure simpler, though multiple seeks per consumer request are still likely. However once we settled on a counter, the jump to directly using the offset seemed natural—both after all are monotonically increasing integers unique to a partition. Since the offset is hidden from the consumer API this decision is ultimately an implementation detail and we went with the more efficient approach.

## Kafka Log Implementation



## Writes

The log allows serial appends which always go to the last file. This file is rolled over to a fresh file when it reaches a configurable size (say 1GB). The log takes two configuration parameters:  $M$ , which gives the number of messages to write before forcing the OS to flush the file to disk, and  $S$ , which gives a number of seconds after which a flush is forced. This gives a durability guarantee of losing at most  $M$  messages or  $S$  seconds of data in the event of a system crash.

## Reads

Reads are done by giving the 64-bit logical offset of a message and an  $S$ -byte max chunk size. This will return an iterator over the messages contained in the  $S$ -byte buffer.  $S$  is intended to be larger than any single message, but in the event of an abnormally large message, the read can be retried multiple times, each time doubling the buffer size, until the message is read successfully. A maximum message and buffer size can be specified to make the server reject messages larger than some size, and to give a bound to the client on the maximum it needs to ever read to get a complete message. It is likely that the read buffer ends with a partial message, this is easily detected by the size delimiting.

The actual process of reading from an offset requires first locating the log segment file in which the data is stored, calculating the file-specific offset from the global offset value, and then reading from that file offset. The search is done as a simple binary search variation against an in-memory range maintained for each file.

The log provides the capability of getting the most recently written message to allow clients to start subscribing as of "right now". This is also useful in the case the consumer fails to consume its data within its SLA-specified number of days. In this case when the client attempts to consume a non-existent offset it is given an `OutOfRangeException` and can either reset itself or fail as appropriate to the use case.

The following is the format of the results sent to the consumer.

```
MessageSetSend (fetch result)

total length      : 4 bytes
error code        : 2 bytes
message 1         : x bytes
...
message n         : x bytes
MultiMessageSetSend (multiFetch result)

total length      : 4 bytes
error code        : 2 bytes
messageSetSend 1
...
messageSetSend n
```

## Deletes

Data is deleted one log segment at a time. The log manager applies two metrics to identify segments which are eligible for deletion: time and size. For time-based policies, the record timestamps are considered, with the largest timestamp in a segment file (order of records is not relevant) defining the retention time for the entire segment. Size-based retention is disabled by default. When enabled the log manager keeps deleting the oldest segment file until the overall size of the partition is within the configured limit again. If both policies are enabled at the same time, a segment that is eligible for deletion due to either policy will be deleted. To avoid locking reads while still allowing deletes that modify the segment list we use a copy-on-write style segment list implementation that provides consistent views to allow a binary search to proceed on an immutable static snapshot view of the log segments while deletes are progressing.

## Guarantees

The log provides a configuration parameter  $M$  which controls the maximum number of messages that are written before forcing a flush to disk. On startup a log recovery process is run that iterates over all messages in the newest log segment and verifies that each message entry is valid. A message entry is valid if the sum of its size and offset are less than the length of the file AND the CRC32 of the message payload matches the CRC stored with the message. In the event corruption is detected the log is truncated to the last valid offset.

Note that two kinds of corruption must be handled: truncation in which an unwritten block is lost due to a crash, and corruption in which a nonsense block is ADDED to the file. The reason for this is that in general the OS makes no guarantee of the write order between the file inode and the actual block data so in addition to losing written data the file can gain nonsense data if the inode is updated with a new size but a crash occurs before the block containing that data is written. The CRC detects this corner case, and prevents it from corrupting the log (though the unwritten messages are, of course, lost).

## 5.5 Distribution

### Consumer Offset Tracking

Kafka consumer tracks the maximum offset it has consumed in each partition and has the capability to commit offsets so that it can resume from those offsets in the event of a restart. Kafka provides the option to store all the offsets for a given consumer group in a designated broker (for that group) called the group coordinator. i.e., any consumer instance in that consumer group should send its offset commits and fetches to that group coordinator (broker). Consumer groups are assigned to coordinators based on their group names. A consumer can look up its coordinator by issuing a `FindCoordinatorRequest` to any Kafka broker and reading the `FindCoordinatorResponse` which will contain the coordinator details. The consumer can then proceed to commit or fetch offsets from the coordinator broker. In case the coordinator moves, the consumer will need to rediscover the coordinator. Offset commits can be done automatically or manually by consumer instance.

When the group coordinator receives an `OffsetCommitRequest`, it appends the request to a special [compacted](#) Kafka topic named `_consumer_offsets`. The broker sends a successful offset commit response to the consumer only after all the replicas of the offsets topic receive the offsets. In case the offsets fail to replicate within a configurable timeout, the offset commit will fail and the consumer may retry the commit after backing off. The brokers periodically compact the offsets topic since it only needs to maintain the most recent offset commit per partition. The coordinator also caches the offsets in an in-memory table in order to serve offset fetches quickly.

When the coordinator receives an offset fetch request, it simply returns the last committed offset vector from the offsets cache. In case coordinator was just started or if it just became the coordinator for a new set of consumer groups (by becoming a leader for a partition of the offsets topic), it may need to load the offsets topic partition into the cache. In this case, the offset fetch will fail with an `CoordinatorLoadInProgressException` and the consumer may retry the `OffsetFetchRequest` after backing off.

### ZooKeeper Directories

The following gives the ZooKeeper structures and algorithms used for co-ordination between consumers and brokers.

### Notation

When an element in a path is denoted `[xyz]`, that means that the value of xyz is not fixed and there is in fact a ZooKeeper znode for each possible value of xyz. For example `/topics/[topic]` would be a directory named /topics containing a sub-directory for each topic name. Numerical ranges are also given such as `[0...5]` to indicate the subdirectories 0, 1, 2, 3, 4. An arrow `->` is used to indicate the contents of a znode. For example `/hello -> world` would indicate a znode /hello containing the value "world".

### Broker Node Registry

```
/brokers/ids/[0...N] --> {"jmx_port":..., "timestamp":..., "endpoints":  
[...], "host":..., "version":..., "port":...} (ephemeral node)
```

This is a list of all present broker nodes, each of which provides a unique logical broker id which identifies it to consumers (which must be given as part of its configuration). On startup, a broker node registers itself by creating a znode with the logical broker id under `/brokers/ids`. The purpose of the logical broker id is to allow a broker to be moved to a different physical machine without affecting consumers. An attempt to register a broker id that is already in use (say because two servers are configured with the same broker id) results in an error.

Since the broker registers itself in ZooKeeper using ephemeral znodes, this registration is dynamic and will disappear if the broker is shutdown or dies (thus notifying consumers it is no longer available).

## [Broker Topic Registry](#)

```
/brokers/topics/[topic]/partitions/[0...N]/state -->
{"controller_epoch":..., "leader":..., "version":..., "leader_epoch":..., "isr": [...]}
(ephemeral node)
```

Each broker registers itself under the topics it maintains and stores the number of partitions for that topic.

## [Cluster Id](#)

The cluster id is a unique and immutable identifier assigned to a Kafka cluster. The cluster id can have a maximum of 22 characters and the allowed characters are defined by the regular expression [a-zA-Z0-9\_-]+, which corresponds to the characters used by the URL-safe Base64 variant with no padding. Conceptually, it is auto-generated when a cluster is started for the first time.

Implementation-wise, it is generated when a broker with version 0.10.1 or later is successfully started for the first time. The broker tries to get the cluster id from the `/cluster/id` znode during startup. If the znode does not exist, the broker generates a new cluster id and creates the znode with this cluster id.

## [Broker node registration](#)

The broker nodes are basically independent, so they only publish information about what they have. When a broker joins, it registers itself under the broker node registry directory and writes information about its host name and port. The broker also register the list of existing topics and their logical partitions in the broker topic registry. New topics are registered dynamically when they are created on the broker.

# [6. OPERATIONS](#)

---

Here is some information on actually running Kafka as a production system based on usage and experience at LinkedIn. Please send us any additional tips you know of.

## [6.1 Basic Kafka Operations](#)

This section will review the most common operations you will perform on your Kafka cluster. All of the tools reviewed in this section are available under the `bin/` directory of the Kafka distribution and each tool will print details on all possible commandline options if it is run with no arguments.

### [Adding and removing topics](#)

You have the option of either adding topics manually or having them be created automatically when data is first published to a non-existent topic. If topics are auto-created then you may want to tune the default [topic configurations](#) used for auto-created topics.

Topics are added and modified using the topic tool:

```
> bin/kafka-topics.sh --bootstrap-server broker_host:port --create --topic my_topic_name \
--partitions 20 --replication-factor 3 --config x=y
```

The replication factor controls how many servers will replicate each message that is written. If you have a replication factor of 3 then up to 2 servers can fail before you will lose access to your data. We recommend you use a replication factor of 2 or 3 so that you can transparently bounce machines without interrupting data consumption.

The partition count controls how many logs the topic will be sharded into. There are several impacts of the partition count. First each partition must fit entirely on a single server. So if you have 20 partitions the full data set (and read and write load) will be handled by no more than 20 servers (not counting replicas). Finally the partition count impacts the maximum parallelism of your consumers. This is discussed in greater detail in the [concepts section](#).

Each sharded partition log is placed into its own folder under the Kafka log directory. The name of such folders consists of the topic name, appended by a dash (-) and the partition id. Since a typical folder name can not be over 255 characters long, there will be a limitation on the length of topic names. We assume the number of partitions will not ever be above 100,000. Therefore, topic names cannot be longer than 249 characters. This leaves just enough room in the folder name for a dash and a potentially 5 digit long partition id.

The configurations added on the command line override the default settings the server has for things like the length of time data should be retained. The complete set of per-topic configurations is documented [here](#).

## Modifying topics

You can change the configuration or partitioning of a topic using the same topic tool.

To add partitions you can do

```
> bin/kafka-topics.sh --bootstrap-server broker_host:port --alter --topic my_topic_name \
--partitions 40
```

Be aware that one use case for partitions is to semantically partition data, and adding partitions doesn't change the partitioning of existing data so this may disturb consumers if they rely on that partition. That is if data is partitioned by `hash(key) % number_of_partitions` then this partitioning will potentially be shuffled by adding partitions but Kafka will not attempt to automatically redistribute data in any way.

To add configs:

```
> bin/kafka-configs.sh --bootstrap-server broker_host:port --entity-type topics --entity-name my_topic_name --alter --add-config x=y
```

To remove a config:

```
> bin/kafka-configs.sh --bootstrap-server broker_host:port --entity-type topics --entity-name my_topic_name --alter --delete-config x
```

And finally deleting a topic:

```
> bin/kafka-topics.sh --bootstrap-server broker_host:port --delete --topic my_topic_name
```

Kafka does not currently support reducing the number of partitions for a topic.

Instructions for changing the replication factor of a topic can be found [here](#).

## Graceful shutdown

The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine. This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes. For the latter cases Kafka supports a more graceful mechanism for stopping a server than just killing it. When a server is stopped gracefully it has two optimizations it will take advantage of:

1. It will sync all its logs to disk to avoid needing to do any log recovery when it restarts (i.e. validating the checksum for all messages in the tail of the log). Log recovery takes time so this speeds up intentional restarts.
2. It will migrate any partitions the server is the leader for to other replicas prior to shutting down. This will make the leadership transfer faster and minimize the time each partition is unavailable to a few milliseconds.

Syncing the logs will happen automatically whenever the server is stopped other than by a hard kill, but the controlled leadership migration requires using a special setting:

```
controlled.shutdown.enable=true
```

Note that controlled shutdown will only succeed if *all* the partitions hosted on the broker have replicas (i.e. the replication factor is greater than 1 *and* at least one of these replicas is alive). This is generally what you want since shutting down the last replica would make that topic partition unavailable.

## Balancing leadership

Whenever a broker stops or crashes, leadership for that broker's partitions transfers to other replicas. When the broker is restarted it will only be a follower for all its partitions, meaning it will not be used for client reads and writes.

To avoid this imbalance, Kafka has a notion of preferred replicas. If the list of replicas for a partition is 1,5,9 then node 1 is preferred as the leader to either node 5 or 9 because it is earlier in the replica list. By default the Kafka cluster will try to restore leadership to the preferred replicas. This behaviour is configured with:

```
auto.leader.rebalance.enable=true
```

You can also set this to false, but you will then need to manually restore leadership to the restored replicas by running the command:

```
> bin/kafka-leader-election.sh --bootstrap-server broker_host:port --election-type preferred --all-topic-partitions
```

## Balancing Replicas Across Racks

The rack awareness feature spreads replicas of the same partition across different racks. This extends the guarantees Kafka provides for broker-failure to cover rack-failure, limiting the risk of data loss should all the brokers on a rack fail at once. The feature can also be applied to other broker groupings such as availability zones in EC2.

You can specify that a broker belongs to a particular rack by adding a property to the broker config:

```
broker.rack=my-rack-id
```

When a topic is [created](#), [modified](#) or replicas are [redistributed](#), the rack constraint will be honoured, ensuring replicas span as many racks as they can (a partition will span min(#racks, replication-factor) different racks).

The algorithm used to assign replicas to brokers ensures that the number of leaders per broker will be constant, regardless of how brokers are distributed across racks. This ensures balanced throughput.

However if racks are assigned different numbers of brokers, the assignment of replicas will not be even. Racks with fewer brokers will get more replicas, meaning they will use more storage and put more resources into replication. Hence it is sensible to configure an equal number of brokers per rack.

## [Mirroring data between clusters & Geo-replication](#)

Kafka administrators can define data flows that cross the boundaries of individual Kafka clusters, data centers, or geographical regions. Please refer to the section on [Geo-Replication](#) for further information.

## [Checking consumer position](#)

Sometimes it's useful to see the position of your consumers. We have a tool that will show the position of all consumers in a consumer group as well as how far behind the end of the log they are. To run this tool on a consumer group named *my-group* consuming a topic named *my-topic* would look like this:

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group

      TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
CONSUMER-ID                               HOST
CLIENT-ID
  my-topic          0            2              4            2
  consumer-1-029af89c-873c-4751-a720-cefd41a669d6  /127.0.0.1
  consumer-1
    my-topic        1            2              3            1
    consumer-1-029af89c-873c-4751-a720-cefd41a669d6  /127.0.0.1
    consumer-1
      my-topic       2            2              3            1
      consumer-2-42c1abd4-e3b2-425d-a8bb-e1ea49b29bb2  /127.0.0.1
      consumer-2
```

## [Managing Consumer Groups](#)

With the `ConsumerGroupCommand` tool, we can list, describe, or delete the consumer groups. The consumer group can be deleted manually, or automatically when the last committed offset for that group expires. Manual deletion works only if the group does not have any active members. For example, to list all consumer groups across all topics:

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

test-consumer-group
```

To view offsets, as mentioned earlier, we "describe" the consumer group like this:

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group
```

TOPIC ID	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID
			HOST	CLIENT-ID	
topic3	0	241019	395308	154289	consumer2
consumer2-e76ea8c3-5d30-4299-9005-47eb41f3d3c4		/127.0.0.1			
topic2	1	520678	803288	282610	consumer2
consumer2-e76ea8c3-5d30-4299-9005-47eb41f3d3c4		/127.0.0.1			
topic3	1	241018	398817	157799	consumer2
consumer2-e76ea8c3-5d30-4299-9005-47eb41f3d3c4		/127.0.0.1			
topic1	0	854144	855809	1665	consumer1
consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1		/127.0.0.1			
topic2	0	460537	803290	342753	consumer1
consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1		/127.0.0.1			
topic3	2	243655	398812	155157	consumer4
consumer4-117fe4d3-c6c1-4178-8ee9-eb4a3954bee0		/127.0.0.1			

There are a number of additional "describe" options that can be used to provide more detailed information about a consumer group:

- --members: This option provides the list of all active members in the consumer group.

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group --members
```

CONSUMER-ID	HOST	CLIENT-ID
<b>#PARTITIONS</b>		
consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1	/127.0.0.1	consumer1
2		
consumer4-117fe4d3-c6c1-4178-8ee9-eb4a3954bee0	/127.0.0.1	consumer4
1		
consumer2-e76ea8c3-5d30-4299-9005-47eb41f3d3c4	/127.0.0.1	consumer2
3		
consumer3-ecea43e4-1f01-479f-8349-f9130b75d8ee	/127.0.0.1	consumer3
0		

- --members --verbose: On top of the information reported by the "--members" options above, this option also provides the partitions assigned to each member.

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group --members --verbose
```

CONSUMER-ID	HOST	CLIENT-ID
<b>#PARTITIONS</b>		
consumer1-3fc8d6f1-581a-4472-bdf3-3515b4aee8c1	/127.0.0.1	consumer1
2	topic1(0), topic2(0)	
consumer4-117fe4d3-c6c1-4178-8ee9-eb4a3954bee0	/127.0.0.1	consumer4
1	topic3(2)	
consumer2-e76ea8c3-5d30-4299-9005-47eb41f3d3c4	/127.0.0.1	consumer2
3	topic2(1), topic3(0,1)	
consumer3-ecea43e4-1f01-479f-8349-f9130b75d8ee	/127.0.0.1	consumer3
0	-	

- --offsets: This is the default describe option and provides the same output as the "--describe" option.
- --state: This option provides useful group-level information.

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe  
--group my-group --state
```

COORDINATOR (ID) <b>#MEMBERS</b>	ASSIGNMENT-STRATEGY	STATE	
localhost:9092 (0)	range	Stable	4

To manually delete one or multiple consumer groups, the "--delete" option can be used:

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete --group  
my-group --group my-other-group  
  
Deletion of requested consumer groups ('my-group', 'my-other-group') was successful.
```

To reset offsets of a consumer group, "--reset-offsets" option can be used. This option supports one consumer group at the time. It requires defining following scopes: --all-topics or --topic. One scope must be selected, unless you use '--from-file' scenario. Also, first make sure that the consumer instances are inactive. See [KIP-122](#) for more details.

It has 3 execution options:

- (default) to display which offsets to reset.
- --execute : to execute --reset-offsets process.
- --export : to export the results to a CSV format.

--reset-offsets also has following scenarios to choose from (at least one scenario must be selected):

- --to-datetime <String: datetime> : Reset offsets to offsets from datetime. Format: 'YYYY-MM-DDTHH:mm:SS.sss'
- --to-earliest : Reset offsets to earliest offset.
- --to-latest : Reset offsets to latest offset.
- --shift-by <Long: number-of-offsets> : Reset offsets shifting current offset by 'n', where 'n' can be positive or negative.
- --from-file : Reset offsets to values defined in CSV file.
- --to-current : Resets offsets to current offset.
- --by-duration <String: duration> : Reset offsets to offset by duration from current timestamp.  
Format: 'PnDTnHnMnS'
- --to-offset : Reset offsets to a specific offset.

Please note, that out of range offsets will be adjusted to available offset end. For example, if offset end is at 10 and offset shift request is of 15, then, offset at 10 will actually be selected.

For example, to reset offsets of a consumer group to the latest offset:

```
> bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-offsets --  
group consumergroup1 --topic topic1 --to-latest
```

TOPIC	PARTITION	NEW-OFFSET
topic1	0	0

If you are using the old high-level consumer and storing the group metadata in ZooKeeper (i.e. `offsets.storage=zookeeper`), pass `--zookeeper` instead of `--bootstrap-server`:

```
> bin/kafka-consumer-groups.sh --zookeeper localhost:2181 --list
```

## Expanding your cluster

Adding servers to a Kafka cluster is easy, just assign them a unique broker id and start up Kafka on your new servers. However these new servers will not automatically be assigned any data partitions, so unless partitions are moved to them they won't be doing any work until new topics are created. So usually when you add machines to your cluster you will want to migrate some existing data to these machines.

The process of migrating data is manually initiated but fully automated. Under the covers what happens is that Kafka will add the new server as a follower of the partition it is migrating and allow it to fully replicate the existing data in that partition. When the new server has fully replicated the contents of this partition and joined the in-sync replica one of the existing replicas will delete their partition's data.

The partition reassignment tool can be used to move partitions across brokers. An ideal partition distribution would ensure even data load and partition sizes across all brokers. The partition reassignment tool does not have the capability to automatically study the data distribution in a Kafka cluster and move partitions around to attain an even load distribution. As such, the admin has to figure out which topics or partitions should be moved around.

The partition reassignment tool can run in 3 mutually exclusive modes:

- **--generate**: In this mode, given a list of topics and a list of brokers, the tool generates a candidate reassignment to move all partitions of the specified topics to the new brokers. This option merely provides a convenient way to generate a partition reassignment plan given a list of topics and target brokers.
- **--execute**: In this mode, the tool kicks off the reassignment of partitions based on the user provided reassignment plan. (using the **--reassignment-json-file** option). This can either be a custom reassignment plan hand crafted by the admin or provided by using the **--generate** option
- **--verify**: In this mode, the tool verifies the status of the reassignment for all partitions listed during the last **--execute**. The status can be either of successfully completed, failed or in progress

## Automatically migrating data to new machines

The partition reassignment tool can be used to move some topics off of the current set of brokers to the newly added brokers. This is typically useful while expanding an existing cluster since it is easier to move entire topics to the new set of brokers, than moving one partition at a time. When used to do this, the user should provide a list of topics that should be moved to the new set of brokers and a target list of new brokers. The tool then evenly distributes all partitions for the given list of topics across the new set of brokers. During this move, the replication factor of the topic is kept constant. Effectively the replicas for all partitions for the input list of topics are moved from the old set of brokers to the newly added brokers.

For instance, the following example will move all partitions for topics `foo1`,`foo2` to the new set of brokers 5,6. At the end of this move, all partitions for topics `foo1` and `foo2` will *only* exist on brokers 5,6.

Since the tool accepts the input list of topics as a json file, you first need to identify the topics you want to move and create the json file as follows:

```
> cat topics-to-move.json
{"topics": [{"topic": "foo1"}, {"topic": "foo2"}],
"version":1
}
```

Once the json file is ready, use the partition reassignment tool to generate a candidate assignment:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --topics-to-
move-json-file topics-to-move.json --broker-list "5,6" --generate
current partition replica assignment
```

```
{"version":1,
"partitions":[{"topic":"foo1","partition":0,"replicas":[2,1]},
 {"topic":"foo1","partition":1,"replicas":[1,3]},
 {"topic":"foo1","partition":2,"replicas":[3,4]},
 {"topic":"foo2","partition":0,"replicas":[4,2]},
 {"topic":"foo2","partition":1,"replicas":[2,1]},
 {"topic":"foo2","partition":2,"replicas":[1,3]}]
}
```

Proposed partition reassignment configuration

```
{"version":1,
"partitions":[{"topic":"foo1","partition":0,"replicas":[6,5]},
 {"topic":"foo1","partition":1,"replicas":[5,6]},
 {"topic":"foo1","partition":2,"replicas":[6,5]},
 {"topic":"foo2","partition":0,"replicas":[5,6]},
 {"topic":"foo2","partition":1,"replicas":[6,5]},
 {"topic":"foo2","partition":2,"replicas":[5,6]}]
}
```

The tool generates a candidate assignment that will move all partitions from topics foo1,foo2 to brokers 5,6. Note, however, that at this point, the partition movement has not started, it merely tells you the current assignment and the proposed new assignment. The current assignment should be saved in case you want to rollback to it. The new assignment should be saved in a json file (e.g. expand-cluster-reassignment.json) to be input to the tool with the --execute option as follows:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file expand-cluster-reassignment.json --execute
Current partition replica assignment

{"version":1,
"partitions":[{"topic":"foo1","partition":0,"replicas":[2,1]},
 {"topic":"foo1","partition":1,"replicas":[1,3]},
 {"topic":"foo1","partition":2,"replicas":[3,4]},
 {"topic":"foo2","partition":0,"replicas":[4,2]},
 {"topic":"foo2","partition":1,"replicas":[2,1]},
 {"topic":"foo2","partition":2,"replicas":[1,3]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started partition reassigments for foo1-0,foo1-1,foo1-2,foo2-0,foo2-
1,foo2-2
```

Finally, the --verify option can be used with the tool to check the status of the partition reassignment. Note that the same expand-cluster-reassignment.json (used with the --execute option) should be used with the --verify option:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file expand-cluster-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] is completed
Reassignment of partition [foo1,1] is still in progress
Reassignment of partition [foo1,2] is still in progress
Reassignment of partition [foo2,0] is completed
Reassignment of partition [foo2,1] is completed
Reassignment of partition [foo2,2] is completed
```

## Custom partition assignment and migration

The partition reassignment tool can also be used to selectively move replicas of a partition to a specific set of brokers. When used in this manner, it is assumed that the user knows the reassignment plan and does not require the tool to generate a candidate reassignment, effectively skipping the --generate step and moving straight to the --execute step

For instance, the following example moves partition 0 of topic foo1 to brokers 5,6 and partition 1 of topic foo2 to brokers 2,3:

The first step is to hand craft the custom reassignment plan in a json file:

```
> cat custom-reassignment.json
{"version":1,"partitions":[{"topic":"foo1","partition":0,"replicas":[5,6]},
 {"topic":"foo2","partition":1,"replicas":[2,3]}]}
```

Then, use the json file with the --execute option to start the reassignment process:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file custom-reassignment.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":0,"replicas":[1,2]},
 {"topic":"foo2","partition":1,"replicas":[3,4]}]
}

Save this to use as the --reassignment-json-file option during rollback
Successfully started partition reassignments for foo1-0,foo2-1
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same custom-reassignment.json (used with the --execute option) should be used with the --verify option:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file custom-reassignment.json --verify
Status of partition reassignment:
Reassignment of partition [foo1,0] is completed
Reassignment of partition [foo2,1] is completed
```

## Decommissioning brokers

The partition reassignment tool does not have the ability to automatically generate a reassignment plan for decommissioning brokers yet. As such, the admin has to come up with a reassignment plan to move the replica for all partitions hosted on the broker to be decommissioned, to the rest of the brokers. This can be relatively tedious as the reassignment needs to ensure that all the replicas are not moved from the decommissioned broker to only one other broker. To make this process effortless, we plan to add tooling support for decommissioning brokers in the future.

## Increasing replication factor

Increasing the replication factor of an existing partition is easy. Just specify the extra replicas in the custom reassignment json file and use it with the --execute option to increase the replication factor of the specified partitions.

For instance, the following example increases the replication factor of partition 0 of topic foo from 1 to 3. Before increasing the replication factor, the partition's only replica existed on broker 5. As part of increasing the replication factor, we will add more replicas on brokers 6 and 7.

The first step is to hand craft the custom reassignment plan in a json file:

```
> cat increase-replication-factor.json
{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5,6,7]}]}
```

Then, use the json file with the --execute option to start the reassignment process:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file increase-replication-factor.json --execute
Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo","partition":0,"replicas":[5]}]}

Save this to use as the --reassignment-json-file option during rollback
Successfully started partition reassignment for foo-0
```

The --verify option can be used with the tool to check the status of the partition reassignment. Note that the same increase-replication-factor.json (used with the --execute option) should be used with the --verify option:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --reassignment-
json-file increase-replication-factor.json --verify
Status of partition reassignment:
Reassignment of partition [foo,0] is completed
```

You can also verify the increase in replication factor with the kafka-topics tool:

```
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic foo --describe
Topic:foo PartitionCount:1    ReplicationFactor:3 Configs:
  Topic: foo    Partition: 0    Leader: 5    Replicas: 5,6,7 Isr: 5,6,7
```

## [Limiting Bandwidth Usage during Data Migration](#)

Kafka lets you apply a throttle to replication traffic, setting an upper bound on the bandwidth used to move replicas from machine to machine. This is useful when rebalancing a cluster, bootstrapping a new broker or adding or removing brokers, as it limits the impact these data-intensive operations will have on users.

There are two interfaces that can be used to engage a throttle. The simplest, and safest, is to apply a throttle when invoking the kafka-reassign-partitions.sh, but kafka-configs.sh can also be used to view and alter the throttle values directly.

So for example, if you were to execute a rebalance, with the below command, it would move partitions at no more than 50MB/s.

```
$ bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --execute --
reassignment-json-file bigger-cluster.json --throttle 50000000
```

When you execute this script you will see the throttle engage:

```
The inter-broker throttle limit was set to 50000000 B/s
Successfully started partition reassignment for foo1-0
```

Should you wish to alter the throttle, during a rebalance, say to increase the throughput so it completes quicker, you can do this by re-running the execute command with the --additional option passing the same reassignment-json-file:

```
$ bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --additional --
execute --reassignment-json-file bigger-cluster.json --throttle 700000000
The inter-broker throttle limit was set to 700000000 B/s
```

Once the rebalance completes the administrator can check the status of the rebalance using the --verify option. If the rebalance has completed, the throttle will be removed via the --verify command. It is important that administrators remove the throttle in a timely manner once rebalancing completes by running the command with the --verify option. Failure to do so could cause regular replication traffic to be throttled.

When the --verify option is executed, and the reassignment has completed, the script will confirm that the throttle was removed:

```
> bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 --verify --
reassignment-json-file bigger-cluster.json
Status of partition reassignment:
Reassignment of partition [my-topic,1] is completed
Reassignment of partition [my-topic,0] is completed

Clearing broker-level throttles on brokers 1,2,3
Clearing topic-level throttles on topic my-topic
```

The administrator can also validate the assigned configs using the kafka-configs.sh. There are two pairs of throttle configuration used to manage the throttling process. First pair refers to the throttle value itself. This is configured, at a broker level, using the dynamic properties:

```
leader.replication.throttled.rate
follower.replication.throttled.rate
```

Then there is the configuration pair of enumerated sets of throttled replicas:

```
leader.replication.throttled.replicas
follower.replication.throttled.replicas
```

Which are configured per topic.

All four config values are automatically assigned by kafka-reassign-partitions.sh (discussed below).

To view the throttle limit configuration:

```
> bin/kafka-configs.sh --describe --bootstrap-server localhost:9092 --entity-type
brokers
Configs for brokers '2' are
leader.replication.throttled.rate=700000000,follower.replication.throttled.rate=700000
000
Configs for brokers '1' are
leader.replication.throttled.rate=700000000,follower.replication.throttled.rate=700000
000
```

This shows the throttle applied to both leader and follower side of the replication protocol. By default both sides are assigned the same throttled throughput value.

To view the list of throttled replicas:

```
> bin/kafka-configs.sh --describe --bootstrap-server localhost:9092 --entity-type topics
Configs for topic 'my-topic' are leader.replication.throttled.replicas=1:102,0:101,
follower.replication.throttled.replicas=1:101,0:102
```

Here we see the leader throttle is applied to partition 1 on broker 102 and partition 0 on broker 101. Likewise the follower throttle is applied to partition 1 on broker 101 and partition 0 on broker 102.

By default kafka-reassign-partitions.sh will apply the leader throttle to all replicas that exist before the rebalance, any one of which might be leader. It will apply the follower throttle to all move destinations. So if there is a partition with replicas on brokers 101,102, being reassigned to 102,103, a leader throttle, for that partition, would be applied to 101,102 and a follower throttle would be applied to 103 only.

If required, you can also use the --alter switch on kafka-configs.sh to alter the throttle configurations manually.

### **Safe usage of throttled replication**

Some care should be taken when using throttled replication. In particular:

#### *(1) Throttle Removal:*

The throttle should be removed in a timely manner once reassignment completes (by running kafka-reassign-partitions.sh --verify).

#### *(2) Ensuring Progress:*

If the throttle is set too low, in comparison to the incoming write rate, it is possible for replication to not make progress. This occurs when:

```
max(BytesInPerSec) > throttle
```

Where BytesInPerSec is the metric that monitors the write throughput of producers into each broker.

The administrator can monitor whether replication is making progress, during the rebalance, using the metric:

```
kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=([-.\w]+),topic=(-.\w+),partition=(0-9)+
```

The lag should constantly decrease during replication. If the metric does not decrease the administrator should increase the throttle throughput as described above.

### **Setting quotas**

Quotas overrides and defaults may be configured at (user, client-id), user or client-id levels as described [here](#). By default, clients receive an unlimited quota. It is possible to set custom quotas for each (user, client-id), user or client-id group.

Configure custom quota for (user=user1, client-id=clientA):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type
users --entity-name user1 --entity-type clients --entity-name clientA
Updated config for entity: user-principal 'user1', client-id 'clientA'.
```

Configure custom quota for user=user1:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type  
users --entity-name user1  
Updated config for entity: user-principal 'user1'.
```

Configure custom quota for client-id=clientA:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type  
clients --entity-name clientA  
Updated config for entity: client-id 'clientA'.
```

It is possible to set default quotas for each (user, client-id), user or client-id group by specifying `--entity-default` option instead of `--entity-name`.

Configure default client-id quota for user=userA:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type  
users --entity-name user1 --entity-type clients --entity-default  
Updated config for entity: user-principal 'user1', default client-id.
```

Configure default quota for user:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type  
users --entity-default  
Updated config for entity: default user-principal.
```

Configure default quota for client-id:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200' --entity-type  
clients --entity-default  
Updated config for entity: default client-id.
```

Here's how to describe the quota for a given (user, client-id):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type  
users --entity-name user1 --entity-type clients --entity-name clientA  
Configs for user-principal 'user1', client-id 'clientA' are  
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

Describe quota for a given user:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type  
users --entity-name user1  
Configs for user-principal 'user1' are  
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

Describe quota for a given client-id:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type clients --entity-name clientA
Configs for client-id 'clientA' are
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

If entity name is not specified, all entities of the specified type are described. For example, describe all users:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type users
Configs for user-principal 'user1' are
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
Configs for default user-principal are
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

Similarly for (user, client):

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type users --entity-type clients
Configs for user-principal 'user1', default client-id are
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
Configs for user-principal 'user1', client-id 'clientA' are
producer_byte_rate=1024,consumer_byte_rate=2048,request_percentage=200
```

## 6.2 Datacenters

Some deployments will need to manage a data pipeline that spans multiple datacenters. Our recommended approach to this is to deploy a local Kafka cluster in each datacenter, with application instances in each datacenter interacting only with their local cluster and mirroring data between clusters (see the documentation on [Geo-Replication](#) for how to do this).

This deployment pattern allows datacenters to act as independent entities and allows us to manage and tune inter-datacenter replication centrally. This allows each facility to stand alone and operate even if the inter-datacenter links are unavailable: when this occurs the mirroring falls behind until the link is restored at which time it catches up.

For applications that need a global view of all data you can use mirroring to provide clusters which have aggregate data mirrored from the local clusters in *all* datacenters. These aggregate clusters are used for reads by applications that require the full data set.

This is not the only possible deployment pattern. It is possible to read from or write to a remote Kafka cluster over the WAN, though obviously this will add whatever latency is required to get the cluster.

Kafka naturally batches data in both the producer and consumer so it can achieve high-throughput even over a high-latency connection. To allow this though it may be necessary to increase the TCP socket buffer sizes for the producer, consumer, and broker using the `socket.send.buffer.bytes` and `socket.receive.buffer.bytes` configurations. The appropriate way to set this is documented [here](#).

It is generally *not* advisable to run a *single* Kafka cluster that spans multiple datacenters over a high-latency link. This will incur very high replication latency both for Kafka writes and ZooKeeper writes, and neither Kafka nor ZooKeeper will remain available in all locations if the network between locations is unavailable.

## 6.3 Geo-Replication (Cross-Cluster Data Mirroring)

### Geo-Replication Overview

Kafka administrators can define data flows that cross the boundaries of individual Kafka clusters, data centers, or geo-regions. Such event streaming setups are often needed for organizational, technical, or legal requirements. Common scenarios include:

- Geo-replication
- Disaster recovery
- Feeding edge clusters into a central, aggregate cluster
- Physical isolation of clusters (such as production vs. testing)
- Cloud migration or hybrid cloud deployments
- Legal and compliance requirements

Administrators can set up such inter-cluster data flows with Kafka's MirrorMaker (version 2), a tool to replicate data between different Kafka environments in a streaming manner. MirrorMaker is built on top of the Kafka Connect framework and supports features such as:

- Replicates topics (data plus configurations)
- Replicates consumer groups including offsets to migrate applications between clusters
- Replicates ACLs
- Preserves partitioning
- Automatically detects new topics and partitions
- Provides a wide range of metrics, such as end-to-end replication latency across multiple data centers/clusters
- Fault-tolerant and horizontally scalable operations

*Note: Geo-replication with MirrorMaker replicates data across Kafka clusters. This inter-cluster replication is different from Kafka's [intra-cluster replication](#), which replicates data within the same Kafka cluster.*

### What Are Replication Flows

With MirrorMaker, Kafka administrators can replicate topics, topic configurations, consumer groups and their offsets, and ACLs from one or more source Kafka clusters to one or more target Kafka clusters, i.e., across cluster environments. In a nutshell, MirrorMaker uses Connectors to consume from source clusters and produce to target clusters.

These directional flows from source to target clusters are called replication flows. They are defined with the format `{source_cluster}->{target_cluster}` in the MirrorMaker configuration file as described later. Administrators can create complex replication topologies based on these flows.

Here are some example patterns:

- Active/Active high availability deployments: `A->B, B->A`
- Active/Passive or Active/Standby high availability deployments: `A->B`
- Aggregation (e.g., from many clusters to one): `A->K, B->K, C->K`
- Fan-out (e.g., from one to many clusters): `K->A, K->B, K->C`
- Forwarding: `A->B, B->C, C->D`

By default, a flow replicates all topics and consumer groups. However, each replication flow can be configured independently. For instance, you can define that only specific topics or consumer groups are replicated from the source cluster to the target cluster.

Here is a first example on how to configure data replication from a `primary` cluster to a `secondary` cluster (an active/passive setup):

```

# Basic settings
clusters = primary, secondary
primary.bootstrap.servers = broker3-primary:9092
secondary.bootstrap.servers = broker5-secondary:9092

# Define replication flows
primary->secondary.enabled = true
primary->secondary.topics = foobar-topic, quux-.*

```

## [Configuring Geo-Replication](#)

The following sections describe how to configure and run a dedicated MirrorMaker cluster. If you want to run MirrorMaker within an existing Kafka Connect cluster or other supported deployment setups, please refer to [KIP-382: MirrorMaker 2.0](#) and be aware that the names of configuration settings may vary between deployment modes.

Beyond what's covered in the following sections, further examples and information on configuration settings are available at:

- [MirrorMakerConfig](#), [MirrorConnectorConfig](#)
- [DefaultTopicFilter](#) for topics, [DefaultGroupFilter](#) for consumer groups
- Example configuration settings in [connect-mirror-maker.properties](#), [KIP-382: MirrorMaker 2.0](#)

### [Configuration File Syntax](#)

The MirrorMaker configuration file is typically named `connect-mirror-maker.properties`. You can configure a variety of components in this file:

- MirrorMaker settings: global settings including cluster definitions (aliases), plus custom settings per replication flow
- Kafka Connect and connector settings
- Kafka producer, consumer, and admin client settings

Example: Define MirrorMaker settings (explained in more detail later).

```

# Global settings
clusters = us-west, us-east    # defines cluster aliases
us-west.bootstrap.servers = broker3-west:9092
us-east.bootstrap.servers = broker5-east:9092

topics = .*      # all topics to be replicated by default

# Specific replication flow settings (here: flow from us-west to us-east)
us-west->us-east.enabled = true
us-west->us-east.topics = foo.* , bar.* # override the default above

```

MirrorMaker is based on the Kafka Connect framework. Any Kafka Connect, source connector, and sink connector settings as described in the [documentation chapter on Kafka Connect](#) can be used directly in the MirrorMaker configuration, without having to change or prefix the name of the configuration setting.

Example: Define custom Kafka Connect settings to be used by MirrorMaker.

```

# Setting Kafka Connect defaults for MirrorMaker
tasks.max = 5

```

Most of the default Kafka Connect settings work well for MirrorMaker out-of-the-box, with the exception of `tasks.max`. In order to evenly distribute the workload across more than one MirrorMaker process, it is recommended to set `tasks.max` to at least 2 (preferably higher) depending on the available hardware resources and the total number of topic-partitions to be replicated.

You can further customize MirrorMaker's Kafka Connect settings *per source or target cluster* (more precisely, you can specify Kafka Connect worker-level configuration settings "per connector"). Use the format of `{cluster}.{config_name}` in the MirrorMaker configuration file.

Example: Define custom connector settings for the `us-west` cluster.

```
# us-west custom settings
us-west.offset.storage.topic = my-mirrormaker-offsets
```

MirrorMaker internally uses the Kafka producer, consumer, and admin clients. Custom settings for these clients are often needed. To override the defaults, use the following format in the MirrorMaker configuration file:

- `{source}.consumer.{consumer_config_name}`
- `{target}.producer.{producer_config_name}`
- `{source_or_target}.admin.{admin_config_name}`

Example: Define custom producer, consumer, admin client settings.

```
# us-west cluster (from which to consume)
us-west.consumer.isolation.level = read_committed
us-west.admin.bootstrap.servers = broker57-primary:9092

# us-east cluster (to which to produce)
us-east.producer.compression.type = gzip
us-east.producer.buffer.memory = 32768
us-east.admin.bootstrap.servers = broker8-secondary:9092
```

## [Creating and Enabling Replication Flows](#)

To define a replication flow, you must first define the respective source and target Kafka clusters in the MirrorMaker configuration file.

- `clusters` (required): comma-separated list of Kafka cluster "aliases"
- `{clusterAlias}.bootstrap.servers` (required): connection information for the specific cluster; comma-separated list of "bootstrap" Kafka brokers

Example: Define two cluster aliases `primary` and `secondary`, including their connection information.

```
clusters = primary, secondary
primary.bootstrap.servers = broker10-primary:9092,broker-11-primary:9092
secondary.bootstrap.servers = broker5-secondary:9092,broker6-secondary:9092
```

Secondly, you must explicitly enable individual replication flows with `{source}->{target}.enabled = true` as needed. Remember that flows are directional: if you need two-way (bidirectional) replication, you must enable flows in both directions.

```
# Enable replication from primary to secondary
primary->secondary.enabled = true
```

By default, a replication flow will replicate all but a few special topics and consumer groups from the source cluster to the target cluster, and automatically detect any newly created topics and groups. The names of replicated topics in the target cluster will be prefixed with the name of the source cluster (see section further below). For example, the topic `foo` in the source cluster `us-west` would be replicated to a topic named `us-west.foo` in the target cluster `us-east`.

The subsequent sections explain how to customize this basic setup according to your needs.

## [Configuring Replication Flows](#)

The configuration of a replication flow is a combination of top-level default settings (e.g., `topics`), on top of which flow-specific settings, if any, are applied (e.g., `us-west->us-east.topics`). To change the top-level defaults, add the respective top-level setting to the MirrorMaker configuration file. To override the defaults for a specific replication flow only, use the syntax format `{source}->{target}.{config.name}`.

The most important settings are:

- `topics`: list of topics or a regular expression that defines which topics in the source cluster to replicate (default: `topics = .*`)
- `topics.exclude`: list of topics or a regular expression to subsequently exclude topics that were matched by the `topics` setting (default: `topics.exclude = .*[-\.]internal, .*\.replica, __.*`)
- `groups`: list of topics or regular expression that defines which consumer groups in the source cluster to replicate (default: `groups = .*`)
- `groups.exclude`: list of topics or a regular expression to subsequently exclude consumer groups that were matched by the `groups` setting (default: `groups.exclude = console-consumer-.*, connect-.*, __.*`)
- `{source}->{target}.enable`: set to `true` to enable the replication flow (default: `false`)

Example:

```
# Custom top-level defaults that apply to all replication flows
topics = .*
groups = consumer-group1, consumer-group2

# Don't forget to enable a flow!
us-west->us-east.enabled = true

# Custom settings for specific replication flows
us-west->us-east.topics = foo./*
us-west->us-east.groups = bar./*
us-west->us-east.emit.heartbeats = false
```

Additional configuration settings are supported, some of which are listed below. In most cases, you can leave these settings at their default values. See [MirrorMakerConfig](#) and [MirrorConnectorConfig](#) for further details.

- `refresh.topics.enabled`: whether to check for new topics in the source cluster periodically (default: true)
- `refresh.topics.interval.seconds`: frequency of checking for new topics in the source cluster; lower values than the default may lead to performance degradation (default: 600, every ten minutes)
- `refresh.groups.enabled`: whether to check for new consumer groups in the source cluster periodically (default: true)
- `refresh.groups.interval.seconds`: frequency of checking for new consumer groups in the source cluster; lower values than the default may lead to performance degradation (default: 600, every ten minutes)
- `sync.topic.configs.enabled`: whether to replicate topic configurations from the source cluster (default: true)
- `sync.topic.acls.enabled`: whether to sync ACLs from the source cluster (default: true)
- `emit.heartbeats.enabled`: whether to emit heartbeats periodically (default: true)
- `emit.heartbeats.interval.seconds`: frequency at which heartbeats are emitted (default: 1, every one seconds)
- `heartbeats.topic.replication.factor`: replication factor of MirrorMaker's internal heartbeat topics (default: 3)

- `emit.checkpoints.enabled`: whether to emit MirrorMaker's consumer offsets periodically (default: true)
- `emit.checkpoints.interval.seconds`: frequency at which checkpoints are emitted (default: 60, every minute)
- `checkpoints.topic.replication.factor`: replication factor of MirrorMaker's internal checkpoints topics (default: 3)
- `sync.group.offsets.enabled`: whether to periodically write the translated offsets of replicated consumer groups (in the source cluster) to `__consumer_offsets` topic in target cluster, as long as no active consumers in that group are connected to the target cluster (default: false)
- `sync.group.offsets.interval.seconds`: frequency at which consumer group offsets are synced (default: 60, every minute)
- `offset-syncs.topic.replication.factor`: replication factor of MirrorMaker's internal offset-sync topics (default: 3)

## [Securing Replication Flows](#)

MirrorMaker supports the same [security settings as Kafka Connect](#), so please refer to the linked section for further information.

Example: Encrypt communication between MirrorMaker and the `us-east` cluster.

```
us-east.security.protocol=SSL
us-east.ssl.truststore.location=/path/to/truststore.jks
us-east.ssl.truststore.password=my-secret-password
us-east.ssl.keystore.location=/path/to/keystore.jks
us-east.ssl.keystore.password=my-secret-password
us-east.ssl.key.password=my-secret-password
```

## [Custom Naming of Replicated Topics in Target Clusters](#)

Replicated topics in a target cluster—sometimes called *remote* topics—are renamed according to a replication policy. MirrorMaker uses this policy to ensure that events (aka records, messages) from different clusters are not written to the same topic-partition. By default as per [DefaultReplicationPolicy](#), the names of replicated topics in the target clusters have the format `{source}.{source_topic_name}`:

us-west	us-east
=====	=====
	bar-topic
foo-topic	--> us-west.foo-topic

You can customize the separator (default: `.`) with the `replication.policy.separator` setting:

```
# Defining a custom separator
us-west->us-east.replication.policy.separator = _
```

If you need further control over how replicated topics are named, you can implement a custom `ReplicationPolicy` and override `replication.policy.class` (default is `DefaultReplicationPolicy`) in the MirrorMaker configuration.

## [Preventing Configuration Conflicts](#)

MirrorMaker processes share configuration via their target Kafka clusters. This behavior may cause conflicts when configurations differ among MirrorMaker processes that operate against the same target cluster.

For example, the following two MirrorMaker processes would be racy:

```
# Configuration of process 1
A->B.enabled = true
A->B.topics = foo

# Configuration of process 2
A->B.enabled = true
A->B.topics = bar
```

In this case, the two processes will share configuration via cluster `B`, which causes a conflict. Depending on which of the two processes is the elected "leader", the result will be that either the topic `foo` or the topic `bar` is replicated, but not both.

It is therefore important to keep the MirrorMaker configuration consistent across replication flows to the same target cluster. This can be achieved, for example, through automation tooling or by using a single, shared MirrorMaker configuration file for your entire organization.

### [\*\*Best Practice: Consume from Remote, Produce to Local\*\*](#)

To minimize latency ("producer lag"), it is recommended to locate MirrorMaker processes as close as possible to their target clusters, i.e., the clusters that it produces data to. That's because Kafka producers typically struggle more with unreliable or high-latency network connections than Kafka consumers.

```
First DC           Second DC
=====           =====
primary ----- MirrorMaker --> secondary
      (remote)           (local)
```

To run such a "consume from remote, produce to local" setup, run the MirrorMaker processes close to and preferably in the same location as the target clusters, and explicitly set these "local" clusters in the `--clusters` command line parameter (blank-separated list of cluster aliases):

```
# Run in secondary's data center, reading from the remote `primary` cluster
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties --clusters secondary
```

The `--clusters secondary` tells the MirrorMaker process that the given cluster(s) are nearby, and prevents it from replicating data or sending configuration to clusters at other, remote locations.

### [\*\*Example: Active/Passive High Availability Deployment\*\*](#)

The following example shows the basic settings to replicate topics from a primary to a secondary Kafka environment, but not from the secondary back to the primary. Please be aware that most production setups will need further configuration, such as security settings.

```
# Unidirectional flow (one-way) from primary to secondary cluster
primary.bootstrap.servers = broker1-primary:9092
secondary.bootstrap.servers = broker2-secondary:9092

primary->secondary.enabled = true
secondary->primary.enabled = false

primary->secondary.topics = foo.* # only replicate some topics
```

### [Example: Active/Active High Availability Deployment](#)

The following example shows the basic settings to replicate topics between two clusters in both ways. Please be aware that most production setups will need further configuration, such as security settings.

```
# Bidirectional flow (two-way) between us-west and us-east clusters
clusters = us-west, us-east
us-west.bootstrap.servers = broker1-west:9092,broker2-west:9092
us-east.bootstrap.servers = broker3-east:9092,broker4-east:9092

us-west->us-east.enabled = true
us-east->us-west.enabled = true
```

*Note on preventing replication "loops" (where topics will be originally replicated from A to B, then the replicated topics will be replicated yet again from B to A, and so forth):* As long as you define the above flows in the same MirrorMaker configuration file, you do not need to explicitly add `topics.exclude` settings to prevent replication loops between the two clusters.

### [Example: Multi-Cluster Geo-Replication](#)

Let's put all the information from the previous sections together in a larger example. Imagine there are three data centers (west, east, north), with two Kafka clusters in each data center (e.g., `west-1`, `west-2`). The example in this section shows how to configure MirrorMaker (1) for Active/Active replication within each data center, as well as (2) for Cross Data Center Replication (XDCR).

First, define the source and target clusters along with their replication flows in the configuration:

```
# Basic settings
clusters: west-1, west-2, east-1, east-2, north-1, north-2
west-1.bootstrap.servers = ...
west-2.bootstrap.servers = ...
east-1.bootstrap.servers = ...
east-2.bootstrap.servers = ...
north-1.bootstrap.servers = ...
north-2.bootstrap.servers = ...

# Replication flows for Active/Active in West DC
west-1->west-2.enabled = true
west-2->west-1.enabled = true

# Replication flows for Active/Active in East DC
east-1->east-2.enabled = true
east-2->east-1.enabled = true

# Replication flows for Active/Active in North DC
north-1->north-2.enabled = true
north-2->north-1.enabled = true

# Replication flows for XDCR via west-1, east-1, north-1
west-1->east-1.enabled = true
west-1->north-1.enabled = true
east-1->west-1.enabled = true
east-1->north-1.enabled = true
north-1->west-1.enabled = true
north-1->east-1.enabled = true
```

Then, in each data center, launch one or more MirrorMaker as follows:

```
# In West DC:  
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties --clusters west-1  
west-2  
  
# In East DC:  
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties --clusters east-1  
east-2  
  
# In North DC:  
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties --clusters north-1  
north-2
```

With this configuration, records produced to any cluster will be replicated within the data center, as well as across to other data centers. By providing the `--clusters` parameter, we ensure that each MirrorMaker process produces data to nearby clusters only.

*Note:* The `--clusters` parameter is, technically, not required here. MirrorMaker will work fine without it. However, throughput may suffer from "producer lag" between data centers, and you may incur unnecessary data transfer costs.

## [Starting Geo-Replication](#)

You can run as few or as many MirrorMaker processes (think: nodes, servers) as needed. Because MirrorMaker is based on Kafka Connect, MirrorMaker processes that are configured to replicate the same Kafka clusters run in a distributed setup: They will find each other, share configuration (see section below), load balance their work, and so on. If, for example, you want to increase the throughput of replication flows, one option is to run additional MirrorMaker processes in parallel.

To start a MirrorMaker process, run the command:

```
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties
```

After startup, it may take a few minutes until a MirrorMaker process first begins to replicate data.

Optionally, as described previously, you can set the parameter `--clusters` to ensure that the MirrorMaker process produces data to nearby clusters only.

```
# Note: The cluster alias us-west must be defined in the configuration file  
$ ./bin/connect-mirror-maker.sh connect-mirror-maker.properties \  
--clusters us-west
```

*Note when testing replication of consumer groups:* By default, MirrorMaker does not replicate consumer groups created by the `kafka-console-consumer.sh` tool, which you might use to test your MirrorMaker setup on the command line. If you do want to replicate these consumer groups as well, set the `groups.exclude` configuration accordingly (default: `groups.exclude = console-consumer-.*, connect-.*, __.*`). Remember to update the configuration again once you completed your testing.

## [Stopping Geo-Replication](#)

You can stop a running MirrorMaker process by sending a SIGTERM signal with the command:

```
$ kill <MirrorMaker pid>
```

## [Applying Configuration Changes](#)

To make configuration changes take effect, the MirrorMaker process(es) must be restarted.

## [Monitoring Geo-Replication](#)

It is recommended to monitor MirrorMaker processes to ensure all defined replication flows are up and running correctly. MirrorMaker is built on the Connect framework and inherits all of Connect's metrics, such `source-record-poll-rate`. In addition, MirrorMaker produces its own metrics under the `kafka.connect.mirror` metric group. Metrics are tagged with the following properties:

- `source`: alias of source cluster (e.g., `primary`)
- `target`: alias of target cluster (e.g., `secondary`)
- `topic`: replicated topic on target cluster
- `partition`: partition being replicated

Metrics are tracked for each replicated topic. The source cluster can be inferred from the topic name. For example, replicating `topic1` from `primary->secondary` will yield metrics like:

- `target=secondary`
- `topic=primary.topic1`
- `partition=1`

The following metrics are emitted:

```
# MBean: kafka.connect.mirror:type=MirrorSourceConnector,target=([-w]+),topic=([-w]+),partition=([0-9]+)

record-count          # number of records replicated source -> target
record-age-ms         # age of records when they are replicated
record-age-ms-min
record-age-ms-max
record-age-ms-avg
replication-latency-ms # time it takes records to propagate source->target
replication-latency-ms-min
replication-latency-ms-max
replication-latency-ms-avg
byte-rate             # average number of bytes/sec in replicated records

# MBean: kafka.connect.mirror:type=MirrorCheckpointConnector,source=([-w]+),target=([-w]+)

checkpoint-latency-ms # time it takes to replicate consumer offsets
checkpoint-latency-ms-min
checkpoint-latency-ms-max
checkpoint-latency-ms-avg
```

These metrics do not differentiate between created-at and log-append timestamps.

## [6.4 Multi-Tenancy](#)

### [Multi-Tenancy Overview](#)

As a highly scalable event streaming platform, Kafka is used by many users as their central nervous system, connecting in real-time a wide range of different systems and applications from various teams and lines of businesses. Such multi-tenant cluster environments command proper control and management to ensure the peaceful coexistence of these different needs. This section highlights features and best practices to set up such shared environments, which should help you operate clusters that meet SLAs/OLAs and that minimize potential collateral damage caused by "noisy neighbors".

Multi-tenancy is a many-sided subject, including but not limited to:

- Creating user spaces for tenants (sometimes called namespaces)
- Configuring topics with data retention policies and more
- Securing topics and clusters with encryption, authentication, and authorization
- Isolating tenants with quotas and rate limits
- Monitoring and metering
- Inter-cluster data sharing (cf. geo-replication)

## [\*\*Creating User Spaces \(Namespaces\) For Tenants With Topic Naming\*\*](#)

Kafka administrators operating a multi-tenant cluster typically need to define user spaces for each tenant. For the purpose of this section, "user spaces" are a collection of topics, which are grouped together under the management of a single entity or user.

In Kafka, the main unit of data is the topic. Users can create and name each topic. They can also delete them, but it is not possible to rename a topic directly. Instead, to rename a topic, the user must create a new topic, move the messages from the original topic to the new, and then delete the original. With this in mind, it is recommended to define logical spaces, based on an hierarchical topic naming structure. This setup can then be combined with security features, such as prefixed ACLs, to isolate different spaces and tenants, while also minimizing the administrative overhead for securing the data in the cluster.

These logical user spaces can be grouped in different ways, and the concrete choice depends on how your organization prefers to use your Kafka clusters. The most common groupings are as follows.

*By team or organizational unit:* Here, the team is the main aggregator. In an organization where teams are the main user of the Kafka infrastructure, this might be the best grouping.

Example topic naming structure:

- `<organization>.<team>.<dataset>.<event-name>`  
(e.g., "acme.infosec.telemetry.logins")

*By project or product:* Here, a team manages more than one project. Their credentials will be different for each project, so all the controls and settings will always be project related.

Example topic naming structure:

- `<project>.<product>.<event-name>`  
(e.g., "mobility.payments.suspicious")

Certain information should normally not be put in a topic name, such as information that is likely to change over time (e.g., the name of the intended consumer) or that is a technical detail or metadata that is available elsewhere (e.g., the topic's partition count and other configuration settings).

To enforce a topic naming structure, several options are available:

- Use [prefix ACLs](#) (cf. [KIP-290](#)) to enforce a common prefix for topic names. For example, team A may only be permitted to create topics whose names start with `payments.teamA..`.
- Define a custom `CreateTopicPolicy` (cf. [KIP-108](#) and the setting `create.topic.policy.class.name`) to enforce strict naming patterns. These policies provide the most flexibility and can cover complex patterns and rules to match an organization's needs.
- Disable topic creation for normal users by denying it with an ACL, and then rely on an external process to create topics on behalf of users (e.g., scripting or your favorite automation toolkit).
- It may also be useful to disable the Kafka feature to auto-create topics on demand by setting `auto.create.topics.enable=false` in the broker configuration. Note that you should not rely solely on this option.

## [Configuring Topics: Data Retention And More](#)

Kafka's configuration is very flexible due to its fine granularity, and it supports a plethora of [per-topic configuration settings](#) to help administrators set up multi-tenant clusters. For example, administrators often need to define data retention policies to control how much and/or for how long data will be stored in a topic, with settings such as [retention.bytes](#) (size) and [retention.ms](#) (time). This limits storage consumption within the cluster, and helps complying with legal requirements such as GDPR.

## [Securing Clusters and Topics: Authentication, Authorization, Encryption](#)

Because the documentation has a dedicated chapter on [security](#) that applies to any Kafka deployment, this section focuses on additional considerations for multi-tenant environments.

Security settings for Kafka fall into three main categories, which are similar to how administrators would secure other client-server data systems, like relational databases and traditional messaging systems.

1. **Encryption** of data transferred between Kafka brokers and Kafka clients, between brokers, between brokers and ZooKeeper nodes, and between brokers and other, optional tools.
2. **Authentication** of connections from Kafka clients and applications to Kafka brokers, as well as connections from Kafka brokers to ZooKeeper nodes.
3. **Authorization** of client operations such as creating, deleting, and altering the configuration of topics; writing events to or reading events from a topic; creating and deleting ACLs. Administrators can also define custom policies to put in place additional restrictions, such as a [CreateTopicPolicy](#) and [AlterConfigPolicy](#) (see [KIP-108](#) and the settings [create.topic.policy.class.name](#), [alter.config.policy.class.name](#)).

When securing a multi-tenant Kafka environment, the most common administrative task is the third category (authorization), i.e., managing the user/client permissions that grant or deny access to certain topics and thus to the data stored by users within a cluster. This task is performed predominantly through the [setting of access control lists \(ACLs\)](#). Here, administrators of multi-tenant environments in particular benefit from putting a hierarchical topic naming structure in place as described in a previous section, because they can conveniently control access to topics through prefixed ACLs ([--resource-pattern-type Prefixed](#)). This significantly minimizes the administrative overhead of securing topics in multi-tenant environments: administrators can make their own trade-offs between higher developer convenience (more lenient permissions, using fewer and broader ACLs) vs. tighter security (more stringent permissions, using more and narrower ACLs).

In the following example, user Alice—a new member of ACME corporation's InfoSec team—is granted write permissions to all topics whose names start with "acme.infosec.", such as "acme.infosec.telemetry.logins" and "acme.infosec.syslogs.events".

```
# Grant permissions to user Alice
$ bin/kafka-acls.sh \
  --bootstrap-server broker1:9092 \
  --add --allow-principal User:Alice \
  --producer \
  --resource-pattern-type prefixed --topic acme.infosec.
```

You can similarly use this approach to isolate different customers on the same shared cluster.

## [Isolating Tenants: Quotas, Rate Limiting, Throttling](#)

Multi-tenant clusters should generally be configured with [quotas](#), which protect against users (tenants) eating up too many cluster resources, such as when they attempt to write or read very high volumes of data, or create requests to brokers at an excessively high rate. This may cause network saturation, monopolize broker resources, and impact other clients—all of which you want to avoid in a shared environment.

**Client quotas:** Kafka supports different types of (per-user principal) client quotas. Because a client's quotas apply irrespective of which topics the client is writing to or reading from, they are a convenient and effective tool to allocate resources in a multi-tenant cluster. [Request rate quotas](#), for example, help to limit a user's impact on broker CPU usage by limiting the time a broker spends on the [request handling path](#) for that user, after which throttling kicks in. In many situations, isolating users with request rate quotas has a bigger impact in multi-tenant clusters than setting incoming/outgoing network bandwidth quotas, because excessive broker CPU usage for processing requests reduces the effective bandwidth the broker can serve. Furthermore, administrators can also define quotas on topic operations —such as create, delete, and alter—to prevent Kafka clusters from being overwhelmed by highly concurrent topic operations (see [KIP-599](#) and the quota type `controller_mutations_rate`).

**Server quotas:** Kafka also supports different types of broker-side quotas. For example, administrators can set a limit on the rate with which the [broker accepts new connections](#), set the [maximum number of connections per broker](#), or set the maximum number of connections allowed [from a specific IP address](#).

For more information, please refer to the [quota overview](#) and [how to set quotas](#).

## [Monitoring and Metering](#)

[Monitoring](#) is a broader subject that is covered [elsewhere](#) in the documentation. Administrators of any Kafka environment, but especially multi-tenant ones, should set up monitoring according to these instructions. Kafka supports a wide range of metrics, such as the rate of failed authentication attempts, request latency, consumer lag, total number of consumer groups, metrics on the quotas described in the previous section, and many more.

For example, monitoring can be configured to track the size of topic-partitions (with the JMX metric `kafka.log.Log.Size.<TOPIC-NAME>`), and thus the total size of data stored in a topic. You can then define alerts when tenants on shared clusters are getting close to using too much storage space.

## [Multi-Tenancy and Geo-Replication](#)

Kafka lets you share data across different clusters, which may be located in different geographical regions, data centers, and so on. Apart from use cases such as disaster recovery, this functionality is useful when a multi-tenant setup requires inter-cluster data sharing. See the section [Geo-Replication \(Cross-Cluster Data Mirroring\)](#) for more information.

## [Further considerations](#)

**Data contracts:** You may need to define data contracts between the producers and the consumers of data in a cluster, using event schemas. This ensures that events written to Kafka can always be read properly again, and prevents malformed or corrupt events being written. The best way to achieve this is to deploy a so-called schema registry alongside the cluster. (Kafka does not include a schema registry, but there are third-party implementations available.) A schema registry manages the event schemas and maps the schemas to topics, so that producers know which topics are accepting which types (schemas) of events, and consumers know how to read and parse events in a topic. Some registry implementations provide further functionality, such as schema evolution, storing a history of all schemas, and schema compatibility settings.

## [6.5 Kafka Configuration](#)

### [Important Client Configurations](#)

The most important producer configurations are:

- `acks`
- `compression`
- `batch size`

The most important consumer configuration is the `fetch size`.

All configurations are documented in the [configuration](#) section.

## A Production Server Config

Here is an example production server configuration:

```
# ZooKeeper
zookeeper.connect=[list of zookeeper servers]

# Log configuration
num.partitions=8
default.replication.factor=3
log.dir=[List of directories. Kafka should have its own dedicated disk(s) or
SSD(s).]

# Other configurations
broker.id=[An integer. Start with 0 and increment by 1 for each new broker.]
listeners=[list of listeners]
auto.create.topics.enable=false
min.insync.replicas=2
queued.max.requests=[number of concurrent requests]
```

Our client configuration varies a fair amount between different use cases.

## 6.6 Java Version

Java 8, Java 11, and Java 17 are supported. Note that Java 8 support has been deprecated since Apache Kafka 3.0 and will be removed in Apache Kafka 4.0. Java 11 and later versions perform significantly better if TLS is enabled, so they are highly recommended (they also include a number of other performance improvements: G1GC, CRC32C, Compact Strings, Thread-Local Handshakes and more). From a security perspective, we recommend the latest released patch version as older freely available versions have disclosed security vulnerabilities. Typical arguments for running Kafka with OpenJDK-based Java implementations (including Oracle JDK) are:

```
-Xmx6g -Xms6g -XX:Metaspacesize=96m -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -
XX:+ExplicitGCInvokesConcurrent
```

For reference, here are the stats for one of LinkedIn's busiest clusters (at peak) that uses said Java arguments:

- 60 brokers
- 50k partitions (replication factor 2)
- 800k messages/sec in
- 300 MB/sec inbound, 1 GB/sec+ outbound

All of the brokers in that cluster have a 90% GC pause time of about 21ms with less than 1 young GC per second.

## 6.7 Hardware and OS

We are using dual quad-core Intel Xeon machines with 24GB of memory.

You need sufficient memory to buffer active readers and writers. You can do a back-of-the-envelope estimate of memory needs by assuming you want to be able to buffer for 30 seconds and compute your memory need as  $\text{write\_throughput} * 30$ .

The disk throughput is important. We have 8x7200 rpm SATA drives. In general disk throughput is the performance bottleneck, and more disks is better. Depending on how you configure flush behavior you may or may not benefit from more expensive disks (if you force flush often then higher RPM SAS drives may be better).

## OS

Kafka should run well on any unix system and has been tested on Linux and Solaris.

We have seen a few issues running on Windows and Windows is not currently a well supported platform though we would be happy to change that.

It is unlikely to require much OS-level tuning, but there are three potentially important OS-level configurations:

- File descriptor limits: Kafka uses file descriptors for log segments and open connections. If a broker hosts many partitions, consider that the broker needs at least  $(\text{number\_of\_partitions})^*$   $(\text{partition\_size}/\text{segment\_size})$  to track all log segments in addition to the number of connections the broker makes. We recommend at least 100000 allowed file descriptors for the broker processes as a starting point. Note: The `mmap()` function adds an extra reference to the file associated with the file descriptor `fildes` which is not removed by a subsequent `close()` on that file descriptor. This reference is removed when there are no more mappings to the file.
- Max socket buffer size: can be increased to enable high-performance data transfer between data centers as [described here](#).
- Maximum number of memory map areas a process may have (aka `vm.max_map_count`). [See the Linux kernel documentation](#). You should keep an eye at this OS-level property when considering the maximum number of partitions a broker may have. By default, on a number of Linux systems, the value of `vm.max_map_count` is somewhere around 65535. Each log segment, allocated per partition, requires a pair of index/timeindex files, and each of these files consumes 1 map area. In other words, each log segment uses 2 map areas. Thus, each partition requires minimum 2 map areas, as long as it hosts a single log segment. That is to say, creating 50000 partitions on a broker will result allocation of 100000 map areas and likely cause broker crash with `OutOfMemoryError` (`Map failed`) on a system with default `vm.max_map_count`. Keep in mind that the number of log segments per partition varies depending on the segment size, load intensity, retention policy and, generally, tends to be more than one.

## Disks and Filesystem

We recommend using multiple drives to get good throughput and not sharing the same drives used for Kafka data with application logs or other OS filesystem activity to ensure good latency. You can either RAID these drives together into a single volume or format and mount each drive as its own directory. Since Kafka has replication the redundancy provided by RAID can also be provided at the application level. This choice has several tradeoffs.

If you configure multiple data directories partitions will be assigned round-robin to data directories. Each partition will be entirely in one of the data directories. If data is not well balanced among partitions this can lead to load imbalance between disks.

RAID can potentially do better at balancing load between disks (although it doesn't always seem to) because it balances load at a lower level. The primary downside of RAID is that it is usually a big performance hit for write throughput and reduces the available disk space.

Another potential benefit of RAID is the ability to tolerate disk failures. However our experience has been that rebuilding the RAID array is so I/O intensive that it effectively disables the server, so this does not provide much real availability improvement.

## Application vs. OS Flush Management

Kafka always immediately writes all data to the filesystem and supports the ability to configure the flush policy that controls when data is forced out of the OS cache and onto disk using the flush. This flush policy can be controlled to force data to disk after a period of time or after a certain number of messages has been written. There are several choices in this configuration.

Kafka must eventually call fsync to know that data was flushed. When recovering from a crash for any log segment not known to be fsync'd Kafka will check the integrity of each message by checking its CRC and also rebuild the accompanying offset index file as part of the recovery process executed on startup.

Note that durability in Kafka does not require syncing data to disk, as a failed node will always recover from its replicas.

We recommend using the default flush settings which disable application fsync entirely. This means relying on the background flush done by the OS and Kafka's own background flush. This provides the best of all worlds for most uses: no knobs to tune, great throughput and latency, and full recovery guarantees. We generally feel that the guarantees provided by replication are stronger than sync to local disk, however the paranoid still may prefer having both and application level fsync policies are still supported.

The drawback of using application level flush settings is that it is less efficient in its disk usage pattern (it gives the OS less leeway to re-order writes) and it can introduce latency as fsync in most Linux filesystems blocks writes to the file whereas the background flushing does much more granular page-level locking.

In general you don't need to do any low-level tuning of the filesystem, but in the next few sections we will go over some of this in case it is useful.

## Understanding Linux OS Flush Behavior

In Linux, data written to the filesystem is maintained in [pagecache](#) until it must be written out to disk (due to an application-level fsync or the OS's own flush policy). The flushing of data is done by a set of background threads called pdflush (or in post 2.6.32 kernels "flusher threads").

Pdflush has a configurable policy that controls how much dirty data can be maintained in cache and for how long before it must be written back to disk. This policy is described [here](#). When Pdflush cannot keep up with the rate of data being written it will eventually cause the writing process to block incurring latency in the writes to slow down the accumulation of data.

You can see the current state of OS memory usage by doing

```
> cat /proc/meminfo
```

The meaning of these values are described in the link above.

Using pagecache has several advantages over an in-process cache for storing data that will be written out to disk:

- The I/O scheduler will batch together consecutive small writes into bigger physical writes which improves throughput.
- The I/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves throughput.
- It automatically uses all the free memory on the machine

## Filesystem Selection

Kafka uses regular files on disk, and as such it has no hard dependency on a specific filesystem. The two filesystems which have the most usage, however, are EXT4 and XFS. Historically, EXT4 has had more usage, but recent improvements to the XFS filesystem have shown it to have better performance characteristics for Kafka's workload with no compromise in stability.

Comparison testing was performed on a cluster with significant message loads, using a variety of filesystem creation and mount options. The primary metric in Kafka that was monitored was the "Request Local Time", indicating the amount of time append operations were taking. XFS resulted in much better local times (160ms vs. 250ms+ for the best EXT4 configuration), as well as lower average wait times. The XFS performance also showed less variability in disk performance.

### General Filesystem Notes

For any filesystem used for data directories, on Linux systems, the following options are recommended to be used at mount time:

- `noatime`: This option disables updating of a file's atime (last access time) attribute when the file is read. This can eliminate a significant number of filesystem writes, especially in the case of bootstrapping consumers. Kafka does not rely on the atime attributes at all, so it is safe to disable this.

### XFS Notes

The XFS filesystem has a significant amount of auto-tuning in place, so it does not require any change in the default settings, either at filesystem creation time or at mount. The only tuning parameters worth considering are:

- `largeio`: This affects the preferred I/O size reported by the `stat` call. While this can allow for higher performance on larger disk writes, in practice it had minimal or no effect on performance.
- `nobarrier`: For underlying devices that have battery-backed cache, this option can provide a little more performance by disabling periodic write flushes. However, if the underlying device is well-behaved, it will report to the filesystem that it does not require flushes, and this option will have no effect.

### EXT4 Notes

EXT4 is a serviceable choice of filesystem for the Kafka data directories, however getting the most performance out of it will require adjusting several mount options. In addition, these options are generally unsafe in a failure scenario, and will result in much more data loss and corruption. For a single broker failure, this is not much of a concern as the disk can be wiped and the replicas rebuilt from the cluster. In a multiple-failure scenario, such as a power outage, this can mean underlying filesystem (and therefore data) corruption that is not easily recoverable. The following options can be adjusted:

- `data=writeback`: Ext4 defaults to `data=ordered` which puts a strong order on some writes. Kafka does not require this ordering as it does very paranoid data recovery on all unflushed log. This setting removes the ordering constraint and seems to significantly reduce latency.
- `Disabling journaling`: Journaling is a tradeoff: it makes reboots faster after server crashes but it introduces a great deal of additional locking which adds variance to write performance. Those who don't care about reboot time and want to reduce a major source of write latency spikes can turn off journaling entirely.
- `commit=num_secs`: This tunes the frequency with which ext4 commits to its metadata journal. Setting this to a lower value reduces the loss of unflushed data during a crash. Setting this to a higher value will improve throughput.
- `nobh`: This setting controls additional ordering guarantees when using `data=writeback` mode. This should be safe with Kafka as we do not depend on write ordering and improves throughput and latency.

- **delalloc:** Delayed allocation means that the filesystem avoids allocating any blocks until the physical write occurs. This allows ext4 to allocate a large extent instead of smaller pages and helps ensure the data is written sequentially. This feature is great for throughput. It does seem to involve some locking in the filesystem which adds a bit of latency variance.

## Replace KRaft Controller Disk

When Kafka is configured to use KRaft, the controllers store the cluster metadata in the directory specified in `metadata.log.dir` -- or the first log directory, if `metadata.log.dir` is not configured. See the documentation for `metadata.log.dir` for details.

If the data in the cluster metadata directory is lost either because of hardware failure or the hardware needs to be replaced, care should be taken when provisioning the new controller node. The new controller node should not be formatted and started until the majority of the controllers have all of the committed data. To determine if the majority of the controllers have the committed data, run the `kafka-metadata-quorum.sh` tool to describe the replication status:

```
> bin/kafka-metadata-quorum.sh --bootstrap-server broker_host:port describe --replication
  NodeId  LogEndOffset    Lag    LastFetchTimestamp    LastCaughtUpTimestamp
Status
  1        25806            0      1662500992757      1662500992757
Leader
```

Check and wait until the `Lag` is small for a majority of the controllers. If the leader's end offset is not increasing, you can wait until the lag is 0 for a majority; otherwise, you can pick the latest leader end offset and wait until all replicas have reached it. Check and wait until the `LastFetchTimestamp` and `LastCaughtUpTimestamp` are close to each other for the majority of the controllers. At this point it is safer to format the controller's metadata log directory. This can be done by running the `kafka-storage.sh` command.

```
> bin/kafka-storage.sh format --cluster-id uuid --config server_properties
```

It is possible for the `bin/kafka-storage.sh format` command above to fail with a message like `Log directory ... is already formatted`. This can happen when combined mode is used and only the metadata log directory was lost but not the others. In that case and only in that case, can you run the `kafka-storage.sh format` command with the `--ignore-formatted` option.

Start the KRaft controller after formatting the log directories.

```
> /bin/kafka-server-start.sh server.properties
```

## 6.8 Monitoring

Kafka uses Yammer Metrics for metrics reporting in the server. The Java clients use Kafka Metrics, a built-in metrics registry that minimizes transitive dependencies pulled into client applications. Both expose metrics via JMX and can be configured to report stats using pluggable stats reporters to hook up to your monitoring system.

All Kafka rate metrics have a corresponding cumulative count metric with suffix `-total`. For example, `records-consumed-rate` has a corresponding metric named `records-consumed-total`.

The easiest way to see the available metrics is to fire up jconsole and point it at a running kafka client or server; this will allow browsing all metrics with JMX.

## [Security Considerations for Remote Monitoring using JMX](#)

Apache Kafka disables remote JMX by default. You can enable remote monitoring using JMX by setting the environment variable `JMX_PORT` for processes started using the CLI or standard Java system properties to enable remote JMX programmatically. You must enable security when enabling remote JMX in production scenarios to ensure that unauthorized users cannot monitor or control your broker or application as well as the platform on which these are running. Note that authentication is disabled for JMX by default in Kafka and security configs must be overridden for production deployments by setting the environment variable `KAFKA_JMX_OPTS` for processes started using the CLI or by setting appropriate Java system properties. See [Monitoring and Management Using JMX Technology](#) for details on securing JMX.

We do graphing and alerting on the following metrics:

DESCRIPTION	MBEAN NAME	NORMAL VALUE
Message in rate	kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=[-.\w]+)	Incoming message rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Byte in from clients	kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=[-.\w]+)	Byte in (from the clients) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Byte in rate from other brokers	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesInPerSec,topic=[-.\w]+)	Byte in (from the other brokers) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Controller Request rate from Broker	kafka.controller:type=ControllerChannelManager,name=RequestRateAndQueueTimeMs,brokerId=[0-9]+)	The rate (requests per second) at which the ControllerChannelManager takes requests from the queue of the given broker. And the time it takes for a request to stay in this queue before it is taken from the queue.
Controller Event queue size	kafka.controller:type=ControllerEventManager,name=EventQueueSize	Size of the ControllerEventManager's queue.
Controller Event queue time	kafka.controller:type=ControllerEventManager,name=EventQueueTimeMs	Time that takes for any event (except the Idle event) to wait in the ControllerEventManager's queue before being processed
Request rate	kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce FetchConsumer FetchFollower},version=[0-9]+)	
Error rate	kafka.network:type=RequestMetrics,name=ErrorsPerSec,request=[-.\w]+),error=[-.\w]+)	Number of errors in responses counted per-request-type, per-error-code. If a response contains multiple errors, all are counted. error=NONE indicates successful responses.
Produce request rate	kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=[-.\w]+)	Produce request rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Fetch request rate	kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestsPerSec,topic=[-.\w]+)	Fetch request (from clients or followers) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Failed produce request rate	kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=[-.\w]+)	Failed Produce request rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Failed fetch request rate	kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec,topic=[-.\w]+)	Failed Fetch request (from clients or followers) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Request size in bytes	kafka.network:type=RequestMetrics,name=RequestBytes,request=[-.\w]+)	Size of requests for each request type.
Temporary memory size in bytes	kafka.network:type=RequestMetrics,name=TemporaryMemoryBytes,request={Produce Fetch}	Temporary memory used for message format conversions and decompression.
Message conversion time	kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce Fetch}	Time in milliseconds spent on message format conversions.
Message conversion rate	kafka.server:type=BrokerTopicMetrics,name={Produce Fetch}MessageConversionsPerSec,topic=[-.\w]+)	Message format conversion rate, for Produce or Fetch requests, per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Request Queue Size	kafka.network:type=RequestChannel,name=RequestQueueSize	Size of the request queue.
Byte out rate to clients	kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=[-.\w]+)	Byte out (to the clients) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Byte out rate to other brokers	kafka.server:type=BrokerTopicMetrics,name=ReplicationBytesOutPerSec,topic=[-.\w]+)	Byte out (to the other brokers) rate per topic. Omitting 'topic=(...)' will yield the all-topic rate.
Rejected byte rate	kafka.server:type=BrokerTopicMetrics,name=BytesRejectedPerSec,topic=[-.\w]+)	Rejected byte rate per topic, due to the record batch size being greater than max.message.bytes configuration. Omitting 'topic=(...)' will yield the all-topic rate.
Message validation failure rate due to no key specified for compacted topic	kafka.server:type=BrokerTopicMetrics,name=NoKeyCompactedTopicRecordsPerSec	0
Message validation failure rate due to invalid magic number	kafka.server:type=BrokerTopicMetrics,name=InvalidMagicNumberRecordsPerSec	0
Message validation failure rate due to incorrect crc checksum	kafka.server:type=BrokerTopicMetrics,name=InvalidMessageCrcRecordsPerSec	0
Message validation failure rate due to non-contiguous offset or sequence number in batch	kafka.server:type=BrokerTopicMetrics,name=InvalidOffsetOrSequenceRecordsPerSec	0
Log flush rate and time	kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs	
# of offline log directories	kafka.log:type=LogManager,name=OfflineLogDirectoryCount	0
Leader election rate	kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs	non-zero when there are broker failures
Unclean leader election rate	kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec	0
Is controller active on broker	kafka.controller:type=KafkaController,name=ActiveControllerCount	only one broker in the cluster should have 1
Pending topic deletes	kafka.controller:type=KafkaController,name=TopicsToDeleteCount	
Pending replica deletes	kafka.controller:type=KafkaController,name=ReplicasToDeleteCount	
Ineligible pending topic deletes	kafka.controller:type=KafkaController,name=TopicsIneligibleToDeleteCount	
Ineligible pending replica deletes	kafka.controller:type=KafkaController,name=ReplicasIneligibleToDeleteCount	
# of under replicated partitions ( ISR  <  all replicas )	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	0
# of under minlsr partitions ( ISR  < min.insync.replicas)	kafka.server:type=ReplicaManager,name=UnderMinLsrPartitionCount	0
# of at minlsr partitions ( ISR  = min.insync.replicas)	kafka.server:type=ReplicaManager,name=AtMinLsrPartitionCount	0
Partition counts	kafka.server:type=ReplicaManager,name=PartitionCount	mostly even across brokers
Offline Replica counts	kafka.server:type=ReplicaManager,name=OfflineReplicaCount	0
Leader replica counts	kafka.server:type=ReplicaManager,name=LeaderCount	mostly even across brokers
ISR shrink rate	kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	If a broker goes down, ISR for some of the partitions will shrink. When that broker is up again, ISR will be expanded once the replicas are fully caught up. Other than that, the expected value for both ISR shrink rate and expansion rate is 0.
ISR expansion rate	kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	See above
Failed ISR update rate	kafka.server:type=ReplicaManager,name=FailedIsrUpdatesPerSec	0

DESCRIPTION	MBEAN NAME	NORMAL VALUE
Max lag in messages btw follower and leader replicas	kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	lag should be proportional to the maximum batch size of a produce request.
Lag in messages per follower replica	kafka.server:type=FetcherLagMetrics,name=ConsumerLag,clientId=[-.\w]+,topic=[-.\w]+,partition=[0-9]+	lag should be proportional to the maximum batch size of a produce request.
Requests waiting in the producer purgatory	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Produce	non-zero if ack=1 is used
Requests waiting in the fetch purgatory	kafka.server:type=DelayedOperationPurgatory,name=PurgatorySize,delayedOperation=Fetch	size depends on fetch.wait.max.ms in the consumer
Request total time	kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce FetchConsumer FetchFollower}	broken into queue, local, remote and response send time
Time the request waits in the request queue	kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce FetchConsumer FetchFollower}	
Time the request is processed at the leader	kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce FetchConsumer FetchFollower}	
Time the request waits for the follower	kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce FetchConsumer FetchFollower}	non-zero for produce requests when ack=-1
Time the request waits in the response queue	kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce FetchConsumer FetchFollower}	
Time to send the response	kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce FetchConsumer FetchFollower}	
Number of messages the consumer lags behind the producer by. Published by the consumer, not broker.	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=(client-id) Attribute: records-lag-max	
The average fraction of time the network processors are idle	kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent	between 0 and 1, ideally > 0.3
The number of connections disconnected on a processor due to a client not re-authenticating and then using the connection beyond its expiration time for anything other than re-authentication	kafka.server:type=socket-server-metrics,listener=[SASL_PLAINTEXT SASL_SSL],networkProcessor=<#>,name=expired-connections-killed-count	ideally 0 when re-authentication is enabled, implying there are no longer any older, pre-2.2.0 clients connecting to this (listener, processor) combination
The total number of connections disconnected, across all processors, due to a client not re-authenticating and then using the connection beyond its expiration time for anything other than re-authentication	kafka.network:type=SocketServer,name=ExpiredConnectionsKilledCount	ideally 0 when re-authentication is enabled, implying there are no longer any older, pre-2.2.0 clients connecting to this broker
The average fraction of time the request handler threads are idle	kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent	between 0 and 1, ideally > 0.3
Bandwidth quota metrics per (user, client-id), user or client-id	kafka.server:type={Produce Fetch},user=[-.\w]+,client-id=[-.\w]+	Two attributes. throttle-time indicates the amount of time in ms the client was throttled. Ideally = 0, byte-rate indicates the data produce/consume rate of the client in bytes/sec. For (user, client-id) quotas, both user and client-id are specified. If per-client-id quota is applied to the client, user is not specified. If per-user quota is applied, client-id is not specified.
Request quota metrics per (user, client-id), user or client-id	kafka.server:type=Request,user=[-.\w]+,client-id=[-.\w]+	Two attributes. throttle-time indicates the amount of time in ms the client was throttled. Ideally = 0, request-time indicates the percentage of time spent in broker network and I/O threads to process requests from client group. For (user, client-id) quotas, both user and client-id are specified. If per-client-id quota is applied to the client, user is not specified. If per-user quota is applied, client-id is not specified.
Requests exempt from throttling	kafka.server:type=Request	exempt-throttle-time indicates the percentage of time spent in broker network and I/O threads to process requests that are exempt from throttling.
ZooKeeper client request latency	kafka.server:type=ZooKeeperClientMetrics,name=ZooKeeperRequestLatencyMs	Latency in milliseconds for ZooKeeper requests from broker.
ZooKeeper connection status	kafka.server:type=SessionExpireListener,name=SessionState	Connection status of broker's ZooKeeper session which may be one of Disconnected   SyncConnected   AuthFailed   ConnectedReadOnly   SaslAuthenticated   Expired.
Max time to load group metadata	kafka.server:type=group-coordinator-metrics,name=partition-load-time-max	maximum time, in milliseconds, it took to load offsets and group metadata from the consumer offset partitions loaded in the last 30 seconds (including time spent waiting for the loading task to be scheduled)
Avg time to load group metadata	kafka.server:type=group-coordinator-metrics,name=partition-load-time-avg	average time, in milliseconds, it took to load offsets and group metadata from the consumer offset partitions loaded in the last 30 seconds (including time spent waiting for the loading task to be scheduled)
Max time to load transaction metadata	kafka.server:type=transaction-coordinator-metrics,name=partition-load-time-max	maximum time, in milliseconds, it took to load transaction metadata from the consumer offset partitions loaded in the last 30 seconds (including time spent waiting for the loading task to be scheduled)
Avg time to load transaction metadata	kafka.server:type=transaction-coordinator-metrics,name=partition-load-time-avg	average time, in milliseconds, it took to load transaction metadata from the consumer offset partitions loaded in the last 30 seconds (including time spent waiting for the loading task to be scheduled)
Consumer Group Offset Count	kafka.server:type=GroupMetadataManager,name=NumOffsets	Total number of committed offsets for Consumer Groups
Consumer Group Count	kafka.server:type=GroupMetadataManager,name=NumGroups	Total number of Consumer Groups
Consumer Group Count, per State	kafka.server:type=GroupMetadataManager,name=NumGroups[PreparingRebalance,CompletingRebalance,Empty,Stable,Dead]	The number of Consumer Groups in each state: PreparingRebalance, CompletingRebalance, Empty, Stable, Dead
Number of reassigning partitions	kafka.server:type=ReplicaManager,name=ReassigningPartitions	The number of reassigning leader partitions on a broker.

DESCRIPTION	MBEAN NAME	NORMAL VALUE
Outgoing byte rate of reassignment traffic	kafka.server:type=BrokerTopicMetrics,name=ReassignmentBytesOutPerSec	0; non-zero when a partition reassignment is in progress.
Incoming byte rate of reassignment traffic	kafka.server:type=BrokerTopicMetrics,name=ReassignmentBytesInPerSec	0; non-zero when a partition reassignment is in progress.
Size of a partition on disk (in bytes)	kafka.log:type=Log,name=Size,topic={[-\w]+},partition={{[0-9]+}}	The size of a partition on disk, measured in bytes.
Number of log segments in a partition	kafka.log:type=Log,name=NumLogSegments,topic={[-\w]+},partition={{[0-9]+}}	The number of log segments in a partition.
First offset in a partition	kafka.log:type=Log,name=LogStartOffset,topic={[-\w]+},partition={{[0-9]+}}	The first offset in a partition.
Last offset in a partition	kafka.log:type=Log,name=LogEndOffset,topic={[-\w]+},partition={{[0-9]+}}	The last offset in a partition.

## KRaft Monitoring Metrics

The set of metrics that allow monitoring of the KRaft quorum and the metadata log.

Note that some exposed metrics depend on the role of the node as defined by `process.roles`

### KRaft Quorum Monitoring Metrics

These metrics are reported on both Controllers and Brokers in a KRaft Cluster

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
Current State	The current state of this member; possible values are leader, candidate, voted, follower, unattached.	kafka.server:type=raft-metrics,name=current-state
Current Leader	The current quorum leader's id; -1 indicates unknown.	kafka.server:type=raft-metrics,name=current-leader
Current Voted	The current voted leader's id; -1 indicates not voted for anyone.	kafka.server:type=raft-metrics,name=current-vote
Current Epoch	The current quorum epoch.	kafka.server:type=raft-metrics,name=current-epoch
High Watermark	The high watermark maintained on this member; -1 if it is unknown.	kafka.server:type=raft-metrics,name=high-watermark
Log End Offset	The current raft log end offset.	kafka.server:type=raft-metrics,name=log-end-offset
Number of Unknown Voter Connections	Number of unknown voters whose connection information is not cached. This value of this metric is always 0.	kafka.server:type=raft-metrics,name=number-unknown-voter-connections
Average Commit Latency	The average time in milliseconds to commit an entry in the raft log.	kafka.server:type=raft-metrics,name=commit-latency-avg
Maximum Commit Latency	The maximum time in milliseconds to commit an entry in the raft log.	kafka.server:type=raft-metrics,name=commit-latency-max
Average Election Latency	The average time in milliseconds spent on electing a new leader.	kafka.server:type=raft-metrics,name=election-latency-avg
Maximum Election Latency	The maximum time in milliseconds spent on electing a new leader.	kafka.server:type=raft-metrics,name=election-latency-max
Fetch Records Rate	The average number of records fetched from the leader of the raft quorum.	kafka.server:type=raft-metrics,name=fetch-records-rate
Append Records Rate	The average number of records appended per sec by the leader of the raft quorum.	kafka.server:type=raft-metrics,name=append-records-rate
Average Poll Idle Ratio	The average fraction of time the client's poll() is idle as opposed to waiting for the user code to process records.	kafka.server:type=raft-metrics,name=poll-idle-ratio-avg

## [KRaft Controller Monitoring Metrics](#)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
Active Controller Count	The number of Active Controllers on this node. Valid values are '0' or '1'.	kafka.controller:type=KafkaController,name=ActiveControllerCount
Event Queue Time Ms	A Histogram of the time in milliseconds that requests spent waiting in the Controller Event Queue.	kafka.controller:type=ControllerEventManager,name=EventQueueTimeMs
Event Queue Processing Time Ms	A Histogram of the time in milliseconds that requests spent being processed in the Controller Event Queue.	kafka.controller:type=ControllerEventManager,name=EventQueueProcessingTimeMs
Fenced Broker Count	The number of fenced brokers as observed by this Controller.	kafka.controller:type=KafkaController,name=FencedBrokerCount
Active Broker Count	The number of active brokers as observed by this Controller.	kafka.controller:type=KafkaController,name=ActiveBrokerCount
Global Topic Count	The number of global topics as observed by this Controller.	kafka.controller:type=KafkaController,name=GlobalTopicCount
Global Partition Count	The number of global partitions as observed by this Controller.	kafka.controller:type=KafkaController,name=GlobalPartitionCount
Offline Partition Count	The number of offline topic partitions (non-internal) as observed by this Controller.	kafka.controller:type=KafkaController,name=OfflinePartitionCount
Preferred Replica Imbalance Count	The count of topic partitions for which the leader is not the preferred leader.	kafka.controller:type=KafkaController,name=PreferredReplicaImbalanceCount
Metadata Error Count	The number of times this controller node has encountered an error during metadata log processing.	kafka.controller:type=KafkaController,name=MetadataErrorCount
Last Applied Record Offset	The offset of the last record from the cluster metadata partition that was applied by the Controller.	kafka.controller:type=KafkaController,name=LastAppliedRecordOffset
Last Committed Record Offset	The offset of the last record committed to this Controller.	kafka.controller:type=KafkaController,name=LastCommittedRecordOffset
Last Applied Record Timestamp	The timestamp of the last record from the cluster metadata partition that was applied by the Controller.	kafka.controller:type=KafkaController,name=LastAppliedRecordTimestamp
Last Applied Record Lag Ms	The difference between now and the timestamp of the last record from the cluster metadata partition that was applied by the controller. For active Controllers the value of this lag is always zero.	kafka.controller:type=KafkaController,name=LastAppliedRecordLagMs

## [KRaft Broker Monitoring Metrics](#)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
Last Applied Record Offset	The offset of the last record from the cluster metadata partition that was applied by the broker	kafka.server:type=broker-metadata-metrics,name=last-applied-record-offset
Last Applied Record Timestamp	The timestamp of the last record from the cluster metadata partition that was applied by the broker.	kafka.server:type=broker-metadata-metrics,name=last-applied-record-timestamp
Last Applied Record Lag Ms	The difference between now and the timestamp of the last record from the cluster metadata partition that was applied by the broker	kafka.server:type=broker-metadata-metrics,name=last-applied-record-lag-ms
Metadata Load Error Count	The number of errors encountered by the BrokerMetadataListener while loading the metadata log and generating a new MetadataDelta based on it.	kafka.server:type=broker-metadata-metrics,name=metadata-load-error-count
Metadata Apply Error Count	The number of errors encountered by the BrokerMetadataPublisher while applying a new MetadataImage based on the latest MetadataDelta.	kafka.server:type=broker-metadata-metrics,name=metadata-apply-error-count

### [Common monitoring metrics for producer/consumer/connectstreams](#)

The following metrics are available on producer/consumer/connector/stream instances. For specific metrics, please see following sections.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
connection-close-rate	Connections closed per second in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
connection-close-total	Total connections closed in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
connection-creation-rate	New connections established per second in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
connection-creation-total	Total new connections established in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
network-io-rate	The average number of network operations (reads or writes) on all connections per second.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
network-io-total	The total number of network operations (reads or writes) on all connections.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
outgoing-byte-rate	The average number of outgoing bytes sent per second to all servers.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
outgoing-byte-total	The total number of outgoing bytes sent to all servers.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
request-rate	The average number of requests sent per second.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
request-total	The total number of requests sent.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
request-size-avg	The average size of all requests in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
request-size-max	The maximum size of any request sent in the window.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
incoming-byte-rate	Bytes/second read off all sockets.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
incoming-byte-total	Total bytes read off all sockets.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
response-rate	Responses received per second.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
response-total	Total responses received.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
select-rate	Number of times the I/O layer checked for new I/O to perform per second.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
select-total	Total number of times the I/O layer checked for new I/O to perform.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-wait-time-ns-avg	The average length of time the I/O thread spent waiting for a socket ready for reads or writes in nanoseconds.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-wait-time-ns-total	The total time the I/O thread spent waiting in nanoseconds.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-waittime-total	<b>*Deprecated*</b> The total time the I/O thread spent waiting in nanoseconds. Replacement is <code>io-wait-time-ns-total</code> .	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-wait-ratio	The fraction of time the I/O thread spent waiting.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
io-time-ns-avg	The average length of time for I/O per select call in nanoseconds.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-time-ns-total	The total time the I/O thread spent doing I/O in nanoseconds.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
iotime-total	<b>*Deprecated*</b> The total time the I/O thread spent doing I/O in nanoseconds. Replacement is <a href="#">io-time-ns-total</a> .	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
io-ratio	The fraction of time the I/O thread spent doing I/O.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
connection-count	The current number of active connections.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
successful-authentication-rate	Connections per second that were successfully authenticated using SASL or SSL.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
successful-authentication-total	Total connections that were successfully authenticated using SASL or SSL.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
failed-authentication-rate	Connections per second that failed authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
failed-authentication-total	Total connections that failed authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
successful-reauthentication-rate	Connections per second that were successfully re-authenticated using SASL.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
successful-reauthentication-total	Total connections that were successfully re-authenticated using SASL.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
reauthentication-latency-max	The maximum latency in ms observed due to re-authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
reauthentication-latency-avg	The average latency in ms observed due to re-authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
failed-reauthentication-rate	Connections per second that failed re-authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
failed-reauthentication-total	Total connections that failed re-authentication.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)
successful-authentication-no-reauth-total	Total connections that were successfully authenticated by older, pre-2.2.0 SASL clients that do not support re-authentication. May only be non-zero.	kafka. [producer consumer connect]:type=[producer consumer connect]-metrics,client-id=([-.\w]+)

### [Common Per-broker metrics for producer/consumer/connectstreams](#)

The following metrics are available on producer/consumer/connector/stream instances. For specific metrics, please see following sections.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
outgoing-byte-rate	The average number of outgoing bytes sent per second for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
outgoing-byte-total	The total number of outgoing bytes sent for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-rate	The average number of requests sent per second for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-total	The total number of requests sent for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-avg	The average size of all requests in the window for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-size-max	The maximum size of any request sent in the window for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
incoming-byte-rate	The average number of bytes received per second for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
incoming-byte-total	The total number of bytes received for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-latency-avg	The average request latency in ms for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
request-latency-max	The maximum request latency in ms for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
response-rate	Responses received per second for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)
response-total	Total responses received for a node.	kafka.[producer consumer connect]:type=[consumer producer connect]-node-metrics,client-id=([-.\w]+),node-id=([0-9]+)

## Producer monitoring

The following metrics are available on producer instances.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
waiting-threads	The number of user threads blocked waiting for buffer memory to enqueue their records.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-total-bytes	The maximum amount of buffer memory the client can use (whether or not it is currently used).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
buffer-available-bytes	The total amount of buffer memory that is not being used (either unallocated or in the free list).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
bufferpool-wait-time	The fraction of time an appender waits for space allocation.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
bufferpool-wait-time-total	<b>*Deprecated*</b> The total time an appender waits for space allocation in nanoseconds. Replacement is <code>bufferpool-wait-time-ns-total</code>	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
bufferpool-wait-time-ns-total	The total time an appender waits for space allocation in nanoseconds.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
flush-time-ns-total	The total time the Producer spent in Producer.flush in nanoseconds.	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
txn-init-time-ns-total	The total time the Producer spent initializing transactions in nanoseconds (for EOS).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
txn-begin-time-ns-total	The total time the Producer spent in beginTransaction in nanoseconds (for EOS).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
txn-send-offsets-time-ns-total	The total time the Producer spent sending offsets to transactions in nanoseconds (for EOS).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
txn-commit-time-ns-total	The total time the Producer spent committing transactions in nanoseconds (for EOS).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)
txn-abort-time-ns-total	The total time the Producer spent aborting transactions in nanoseconds (for EOS).	kafka.producer:type=producer-metrics,client-id=([-.\w]+)

## Producer Sender Metrics

<b>kafka.producer:type=producer-metrics,client-id="{client-id}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	batch-size-avg	The average number of bytes sent per partition per-request.
	batch-size-max	The max number of bytes sent per partition per-request.
	batch-split-rate	The average number of batch splits per second
	batch-split-total	The total number of batch splits
	compression-rate-avg	The average compression rate of record batches, defined as the average ratio of the compressed batch size over the uncompressed size.
	metadata-age	The age in seconds of the current producer metadata being used.
	produce-throttle-time-avg	The average time in ms a request was throttled by a broker
	produce-throttle-time-max	The maximum time in ms a request was throttled by a broker
	record-error-rate	The average per-second number of record sends that resulted in errors
	record-error-total	The total number of record sends that resulted in errors
	record-queue-time-avg	The average time in ms record batches spent in the send buffer.
	record-queue-time-max	The maximum time in ms record batches spent in the send buffer.
	record-retry-rate	The average per-second number of retried record sends
	record-retry-total	The total number of retried record sends
	record-send-rate	The average number of records sent per second.
	record-send-total	The total number of records sent.

<b>kafka.producer:type=producer-metrics,client-id="{client-id}"</b>		
	record-size-avg	The average record size
	record-size-max	The maximum record size
	records-per-request-avg	The average number of records per request.
	request-latency-avg	The average request latency in ms
	request-latency-max	The maximum request latency in ms
	requests-in-flight	The current number of in-flight requests awaiting a response.
<b>kafka.producer:type=producer-topic-metrics,client-id="{client-id}",topic="{topic}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	byte-rate	The average number of bytes sent per second for a topic.
	byte-total	The total number of bytes sent for a topic.
	compression-rate	The average compression rate of record batches for a topic, defined as the average ratio of the compressed batch size over the uncompressed size.
	record-error-rate	The average per-second number of record sends that resulted in errors for a topic
	record-error-total	The total number of record sends that resulted in errors for a topic
	record-retry-rate	The average per-second number of retried record sends for a topic
	record-retry-total	The total number of retried record sends for a topic
	record-send-rate	The average number of records sent per second for a topic.
	record-send-total	The total number of records sent for a topic.

## [Consumer monitoring](#)

The following metrics are available on consumer instances.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
time-between-poll-avg	The average delay between invocations of poll().	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
time-between-poll-max	The max delay between invocations of poll().	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
last-poll-seconds-ago	The number of seconds since the last poll() invocation.	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
poll-idle-ratio-avg	The average fraction of time the consumer's poll() is idle as opposed to waiting for the user code to process records.	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
committed-time-ns-total	The total time the Consumer spent in committed in nanoseconds.	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)
commit-sync-time-ns-total	The total time the Consumer spent committing offsets in nanoseconds (for AOS).	kafka.consumer:type=consumer-metrics,client-id=([-.\w]+)

## [Consumer Group Metrics](#)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
commit-latency-avg	The average time taken for a commit request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
commit-latency-max	The max time taken for a commit request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
commit-rate	The number of commit calls per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
commit-total	The total number of commit calls	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
assigned-partitions	The number of partitions currently assigned to this consumer	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
heartbeat-response-time-max	The max time taken to receive a response to a heartbeat request	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
heartbeat-rate	The average number of heartbeats per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
heartbeat-total	The total number of heartbeats	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-time-avg	The average time taken for a group rejoin	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-time-max	The max time taken for a group rejoin	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-rate	The number of group joins per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
join-total	The total number of group joins	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
sync-time-avg	The average time taken for a group sync	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
sync-time-max	The max time taken for a group sync	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
sync-rate	The number of group syncs per second	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
sync-total	The total number of group syncs	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
rebalance-latency-avg	The average time taken for a group rebalance	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
rebalance-latency-max	The max time taken for a group rebalance	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
rebalance-latency-total	The total time taken for group rebalances so far	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
rebalance-total	The total number of group rebalances participated	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
rebalance-rate-per-hour	The number of group rebalance participated per hour	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
failed-rebalance-total	The total number of failed group rebalances	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
failed-rebalance-rate-per-hour	The number of failed group rebalance event per hour	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
last-rebalance-seconds-ago	The number of seconds since the last rebalance event	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
last-heartbeat-seconds-ago	The number of seconds since the last controller heartbeat	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
partitions-revoked-latency-avg	The average time taken by the on-partitions-revoked rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
partitions-revoked-latency-max	The max time taken by the on-partitions-revoked rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
partitions-assigned-latency-avg	The average time taken by the on-partitions-assigned rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
partitions-assigned-latency-max	The max time taken by the on-partitions-assigned rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
partitions-lost-latency-avg	The average time taken by the on-partitions-lost rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)
partitions-lost-latency-max	The max time taken by the on-partitions-lost rebalance listener callback	kafka.consumer:type=consumer-coordinator-metrics,client-id=([-.\w]+)

### [Consumer Fetch Metrics](#)

<b>kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	bytes-consumed-rate	The average number of bytes consumed per second
	bytes-consumed-total	The total number of bytes consumed
	fetch-latency-avg	The average time taken for a fetch request.
	fetch-latency-max	The max time taken for any fetch request.
	fetch-rate	The number of fetch requests per second.
	fetch-size-avg	The average number of bytes fetched per request
	fetch-size-max	The maximum number of bytes fetched per request
	fetch-throttle-time-avg	The average throttle time in ms
	fetch-throttle-time-max	The maximum throttle time in ms
	fetch-total	The total number of fetch requests.
	records-consumed-rate	The average number of records consumed per second
	records-consumed-total	The total number of records consumed
	records-lag-max	The maximum lag in terms of number of records for any partition in this window. NOTE: This is based on current offset and not committed offset
	records-lead-min	The minimum lead in terms of number of records for any partition in this window

<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"</code>		
	records-per-request-avg	The average number of records in each request
<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}",topic="{topic}"</code>		
	ATTRIBUTE NAME	DESCRIPTION
	bytes-consumed-rate	The average number of bytes consumed per second for a topic
	bytes-consumed-total	The total number of bytes consumed for a topic
	fetch-size-avg	The average number of bytes fetched per request for a topic
	fetch-size-max	The maximum number of bytes fetched per request for a topic
	records-consumed-rate	The average number of records consumed per second for a topic
	records-consumed-total	The total number of records consumed for a topic
	records-per-request-avg	The average number of records in each request for a topic
<code>kafka.consumer:type=consumer-fetch-manager-metrics,partition="{partition}",topic="{topic}",client-id="{client-id}"</code>		
	ATTRIBUTE NAME	DESCRIPTION
	preferred-read-replica	The current read replica for the partition, or -1 if reading from leader
	records-lag	The latest lag of the partition
	records-lag-avg	The average lag of the partition

<code>kafka.consumer:type=consumer-fetch-manager-metrics,client-id="{client-id}"</code>		
	records-lag-max	The max lag of the partition
	records-lead	The latest lead of the partition
	records-lead-avg	The average lead of the partition
	records-lead-min	The min lead of the partition

## [Connect Monitoring](#)

A Connect worker process contains all the producer and consumer metrics as well as metrics specific to Connect. The worker process itself has a number of metrics, while each connector and task have additional metrics.

[2023-01-19 19:37:36,494] INFO Metrics scheduler closed  
 (org.apache.kafka.common.metrics.Metrics:693) [2023-01-19 19:37:36,497] INFO Metrics reporters closed  
 (org.apache.kafka.common.metrics.Metrics:703)

<b>kafka.connect:type=connect-worker-metrics</b>		
	ATTRIBUTE NAME	DESCRIPTION
	connector-count	The number of connectors run in this worker.
	connector-startup-attempts-total	The total number of connector startups that this worker has attempted.
	connector-startup-failure-percentage	The average percentage of this worker's connectors starts that failed.
	connector-startup-failure-total	The total number of connector starts that failed.
	connector-startup-success-percentage	The average percentage of this worker's connectors starts that succeeded.
	connector-startup-success-total	The total number of connector starts that succeeded.
	task-count	The number of tasks run in this worker.
	task-startup-attempts-total	The total number of task startups that this worker has attempted.
	task-startup-failure-percentage	The average percentage of this worker's tasks starts that failed.
	task-startup-failure-total	The total number of task starts that failed.
	task-startup-success-percentage	The average percentage of this worker's tasks starts that succeeded.
	task-startup-success-total	The total number of task starts that succeeded.
kafka.connect:type=connect-worker-metrics,connector=" {connector}"		
	ATTRIBUTE NAME	DESCRIPTION
	connector-destroyed-task-count	The number of destroyed tasks of the connector on the worker.
	connector-failed-task-count	The number of failed tasks of the connector on the worker.
	connector-paused-task-count	The number of paused tasks of the connector on the worker.

<b>kafka.connect:type=connect-worker-metrics</b>		
	connector-restarting-task-count	The number of restarting tasks of the connector on the worker.
	connector-running-task-count	The number of running tasks of the connector on the worker.
	connector-total-task-count	The number of tasks of the connector on the worker.
	connector-unassigned-task-count	The number of unassigned tasks of the connector on the worker.
<b>kafka.connect:type=connect-worker-rebalance-metrics</b>		
	ATTRIBUTE NAME	DESCRIPTION
	completed-rebalances-total	The total number of rebalances completed by this worker.
	connect-protocol	The Connect protocol used by this cluster
	epoch	The epoch or generation number of this worker.
	leader-name	The name of the group leader.
	rebalance-avg-time-ms	The average time in milliseconds spent by this worker to rebalance.
	rebalance-max-time-ms	The maximum time in milliseconds spent by this worker to rebalance.
	rebalancing	Whether this worker is currently rebalancing.
	time-since-last-rebalance-ms	The time in milliseconds since this worker completed the most recent rebalance.
<b>kafka.connect:type=connector-metrics,connector="{connector}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	connector-class	The name of the connector class.
	connector-type	The type of the connector. One of 'source' or 'sink'.
	connector-version	The version of the connector class, as reported by the connector.
	status	The status of the connector. One of 'unassigned', 'running', 'paused', 'failed', or 'restarting'.
<b>kafka.connect:type=connector-task-metrics,connector="{connector}",task="{task}"</b>		

<b>kafka.connect:type=connect-worker-metrics</b>		
	ATTRIBUTE NAME	DESCRIPTION
	batch-size-avg	The average size of the batches processed by the connector.
	batch-size-max	The maximum size of the batches processed by the connector.
	offset-commit-avg-time-ms	The average time in milliseconds taken by this task to commit offsets.
	offset-commit-failure-percentage	The average percentage of this task's offset commit attempts that failed.
	offset-commit-max-time-ms	The maximum time in milliseconds taken by this task to commit offsets.
	offset-commit-success-percentage	The average percentage of this task's offset commit attempts that succeeded.
	pause-ratio	The fraction of time this task has spent in the pause state.
	running-ratio	The fraction of time this task has spent in the running state.
	status	The status of the connector task. One of 'unassigned', 'running', 'paused', 'failed', or 'restarting'.
<b>kafka.connect:type=sink-task-metrics,connector="{connector}",task="{task}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	offset-commit-completion-rate	The average per-second number of offset commit completions that were completed successfully.
	offset-commit-completion-total	The total number of offset commit completions that were completed successfully.
	offset-commit-seq-no	The current sequence number for offset commits.
	offset-commit-skip-rate	The average per-second number of offset commit completions that were received too late and skipped/ignored.
	offset-commit-skip-total	The total number of offset commit completions that were received too late and skipped/ignored.
	partition-count	The number of topic partitions assigned to this task belonging to the named sink connector in this worker.
	put-batch-avg-time-ms	The average time taken by this task to put a batch of sinks records.

<b>kafka.connect:type=connect-worker-metrics</b>		
	put-batch-max-time-ms	The maximum time taken by this task to put a batch of sinks records.
	sink-record-active-count	The number of records that have been read from Kafka but not yet completely committed/flushed/acknowledged by the sink task.
	sink-record-active-count-avg	The average number of records that have been read from Kafka but not yet completely committed/flushed/acknowledged by the sink task.
	sink-record-active-count-max	The maximum number of records that have been read from Kafka but not yet completely committed/flushed/acknowledged by the sink task.
	sink-record-lag-max	The maximum lag in terms of number of records that the sink task is behind the consumer's position for any topic partitions.
	sink-record-read-rate	The average per-second number of records read from Kafka for this task belonging to the named sink connector in this worker. This is before transformations are applied.
	sink-record-read-total	The total number of records read from Kafka by this task belonging to the named sink connector in this worker, since the task was last restarted.
	sink-record-send-rate	The average per-second number of records output from the transformations and sent/put to this task belonging to the named sink connector in this worker. This is after transformations are applied and excludes any records filtered out by the transformations.
	sink-record-send-total	The total number of records output from the transformations and sent/put to this task belonging to the named sink connector in this worker, since the task was last restarted.
<b>kafka.connect:type=source-task-metrics,connector="{connector}",task="{task}"</b>		
	ATTRIBUTE NAME	DESCRIPTION
	poll-batch-avg-time-ms	The average time in milliseconds taken by this task to poll for a batch of source records.
	poll-batch-max-time-ms	The maximum time in milliseconds taken by this task to poll for a batch of source records.
	source-record-active-count	The number of records that have been produced by this task but not yet completely written to Kafka.
	source-record-active-count-avg	The average number of records that have been produced by this task but not yet completely written to Kafka.

<b>kafka.connect:type=connect-worker-metrics</b>		
	source-record-active-count-max	The maximum number of records that have been produced by this task but not yet completely written to Kafka.
	source-record-poll-rate	The average per-second number of records produced/polled (before transformation) by this task belonging to the named source connector in this worker.
	source-record-poll-total	The total number of records produced/polled (before transformation) by this task belonging to the named source connector in this worker.
	source-record-write-rate	The average per-second number of records output from the transformations and written to Kafka for this task belonging to the named source connector in this worker. This is after transformations are applied and excludes any records filtered out by the transformations.
	source-record-write-total	The number of records output from the transformations and written to Kafka for this task belonging to the named source connector in this worker, since the task was last restarted.
	transaction-size-avg	The average number of records in the transactions the task has committed so far.
	transaction-size-max	The number of records in the largest transaction the task has committed so far.
	transaction-size-min	The number of records in the smallest transaction the task has committed so far.
kafka.connect:type=task-error-metrics,connector="{connector}",task="{task}"		
	ATTRIBUTE NAME	DESCRIPTION
	deadletterqueue-produce-failures	The number of failed writes to the dead letter queue.
	deadletterqueue-produce-requests	The number of attempted writes to the dead letter queue.
	last-error-timestamp	The epoch timestamp when this task last encountered an error.
	total-errors-logged	The number of errors that were logged.
	total-record-errors	The number of record processing errors in this task.
	total-record-failures	The number of record processing failures in this task.

<b>kafka.connect:type=connect-worker-metrics</b>		
	total-records-skipped	The number of records skipped due to errors.
	total-retries	The number of operations retried.

## [Streams Monitoring](#)

A Kafka Streams instance contains all the producer and consumer metrics as well as additional metrics specific to Streams. The metrics have three recording levels: `info`, `debug`, and `trace`.

Note that the metrics have a 4-layer hierarchy. At the top level there are client-level metrics for each started Kafka Streams client. Each client has stream threads, with their own metrics. Each stream thread has tasks, with their own metrics. Each task has a number of processor nodes, with their own metrics. Each task also has a number of state stores and record caches, all with their own metrics.

Use the following configuration option to specify which metrics you want collected:

```
metrics.recording.level="info"
```

### [Client Metrics](#)

All of the following metrics have a recording level of `info`:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
version	The version of the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
commit-id	The version control commit ID of the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
application-id	The application ID of the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
topology-description	The description of the topology executed in the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
state	The state of the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)
failed-stream-threads	The number of failed stream threads since the start of the Kafka Streams client.	kafka.streams:type=stream-metrics,client-id=([-.\w]+)

### [Thread Metrics](#)

All of the following metrics have a recording level of `info`:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
commit-latency-avg	The average execution time in ms, for committing, across all running tasks of this thread.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
commit-latency-max	The maximum execution time in ms, for committing, across all running tasks of this thread.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
poll-latency-avg	The average execution time in ms, for consumer polling.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
poll-latency-max	The maximum execution time in ms, for consumer polling.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
process-latency-avg	The average execution time in ms, for processing.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
process-latency-max	The maximum execution time in ms, for processing.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
punctuate-latency-avg	The average execution time in ms, for punctuating.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
punctuate-latency-max	The maximum execution time in ms, for punctuating.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
commit-rate	The average number of commits per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
commit-total	The total number of commit calls.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
poll-rate	The average number of consumer poll calls per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
poll-total	The total number of consumer poll calls.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
process-rate	The average number of processed records per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
process-total	The total number of processed records.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
punctuate-rate	The average number of punctuate calls per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
punctuate-total	The total number of punctuate calls.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
task-created-rate	The average number of tasks created per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
task-created-total	The total number of tasks created.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
task-closed-rate	The average number of tasks closed per second.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
task-closed-total	The total number of tasks closed.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
blocked-time-ns-total	The total time the thread spent blocked on kafka.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)
thread-start-time	The time that the thread was started.	kafka.streams:type=stream-thread-metrics,thread-id=([-.\w]+)

### Task Metrics

All of the following metrics have a recording level of `debug`, except for the `dropped-records-*` and `active-process-ratio` metrics which have a recording level of `info`:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
process-latency-avg	The average execution time in ns, for processing.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
process-latency-max	The maximum execution time in ns, for processing.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
process-rate	The average number of processed records per second across all source processor nodes of this task.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
process-total	The total number of processed records across all source processor nodes of this task.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
commit-latency-avg	The average execution time in ns, for committing.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
commit-latency-max	The maximum execution time in ns, for committing.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
commit-rate	The average number of commit calls per second.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
commit-total	The total number of commit calls.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
record-lateness-avg	The average observed lateness of records (stream time - record timestamp).	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
record-lateness-max	The max observed lateness of records (stream time - record timestamp).	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
enforced-processing-rate	The average number of enforced processings per second.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
enforced-processing-total	The total number enforced processings.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
dropped-records-rate	The average number of records dropped within this task.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)
dropped-records-total	The total number of records dropped within this task.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
active-process-ratio	The fraction of time the stream thread spent on processing this task among all assigned active tasks.	kafka.streams:type=stream-task-metrics,thread-id=([-.\w]+),task-id=([-.\w]+)

### [Processor Node Metrics](#)

The following metrics are only available on certain types of nodes, i.e., the process-\* metrics are only available for source processor nodes, the suppression-emit-\* metrics are only available for suppression operation nodes, and the record-e2e-latency-\* metrics are only available for source processor nodes and terminal nodes (nodes without successor nodes). All of the metrics have a recording level of `debug`, except for the record-e2e-latency-\* metrics which have a recording level of `info`:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
bytes-consumed-total	The total number of bytes consumed by a source processor node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+),topic=([-.\w]+)
bytes-produced-total	The total number of bytes produced by a sink processor node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+),topic=([-.\w]+)
process-rate	The average number of records processed by a source processor node per second.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
process-total	The total number of records processed by a source processor node per second.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
suppression-emit-rate	The rate at which records that have been emitted downstream from suppression operation nodes.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
suppression-emit-total	The total number of records that have been emitted downstream from suppression operation nodes.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
record-e2e-latency-avg	The average end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
record-e2e-latency-max	The maximum end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
record-e2e-latency-min	The minimum end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+)
records-consumed-total	The total number of records consumed by a source processor node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+),topic=([-.\w]+)
records-produced-total	The total number of records produced by a sink processor node.	kafka.streams:type=stream-processor-node-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),processor-node-id=([-.\w]+),topic=([-.\w]+)

### [State Store Metrics](#)

All of the following metrics have a recording level of `debug`, except for the `record-e2e-latency-*` metrics which have a recording level `trace`. Note that the `store-scope` value is specified in `StoreSupplier#metricsScope()` for user's customized state stores; for built-in state stores, currently we have:

- `in-memory-state`
- `in-memory-lru-state`
- `in-memory-window-state`
- `in-memory-suppression` (for suppression buffers)
- `rocksdb-state` (for RocksDB backed key-value store)
- `rocksdb-window-state` (for RocksDB backed window store)
- `rocksdb-session-state` (for RocksDB backed session store)

Metrics `suppression-buffer-size-avg`, `suppression-buffer-size-max`, `suppression-buffer-count-avg`, and `suppression-buffer-count-max` are only available for suppression buffers. All other metrics are not available for suppression buffers.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
put-latency-avg	The average put execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-latency-max	The maximum put execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-if-absent-latency-avg	The average put-if-absent execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-if-absent-latency-max	The maximum put-if-absent execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
get-latency-avg	The average get execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
get-latency-max	The maximum get execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
delete-latency-avg	The average delete execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
delete-latency-max	The maximum delete execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-all-latency-avg	The average put-all execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-all-latency-max	The maximum put-all execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
all-latency-avg	The average all operation execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
all-latency-max	The maximum all operation execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
range-latency-avg	The average range execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
range-latency-max	The maximum range execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
flush-latency-avg	The average flush execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
flush-latency-max	The maximum flush execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
restore-latency-avg	The average restore execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
restore-latency-max	The maximum restore execution time in ns.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-rate	The average put rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
put-if-absent-rate	The average put-if-absent rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
get-rate	The average get rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]
delete-rate	The average delete rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=[-.\w+],task-id=[-.\w+],[store-scope]-id=[-.\w+]

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
put-all-rate	The average put-all rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
all-rate	The average all operation rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
range-rate	The average range rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
flush-rate	The average flush rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
restore-rate	The average restore rate for this store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
suppression-buffer-size-avg	The average total size, in bytes, of the buffered data over the sampling window.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),in-memory-suppression-id=([-.\w]+)
suppression-buffer-size-max	The maximum total size, in bytes, of the buffered data over the sampling window.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),in-memory-suppression-id=([-.\w]+)
suppression-buffer-count-avg	The average number of records buffered over the sampling window.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),in-memory-suppression-id=([-.\w]+)
suppression-buffer-count-max	The maximum number of records buffered over the sampling window.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),in-memory-suppression-id=([-.\w]+)
record-e2e-latency-avg	The average end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
record-e2e-latency-max	The maximum end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
record-e2e-latency-min	The minimum end-to-end latency of a record, measured by comparing the record timestamp with the system time when it has been fully processed by the node.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)

### [RocksDB Metrics](#)

RocksDB metrics are grouped into statistics-based metrics and properties-based metrics. The former are recorded from statistics that a RocksDB state store collects whereas the latter are recorded from properties that RocksDB exposes. Statistics collected by RocksDB provide cumulative measurements over time, e.g. bytes written to the state store. Properties exposed by RocksDB provide current measurements, e.g., the amount of memory currently used. Note that the `store-scope` for built-in RocksDB state stores are currently the following:

- `rocksdb-state` (for RocksDB backed key-value store)
- `rocksdb-window-state` (for RocksDB backed window store)
- `rocksdb-session-state` (for RocksDB backed session store)

**RocksDB Statistics-based Metrics:** All of the following statistics-based metrics have a recording level of `debug` because collecting statistics in [RocksDB may have an impact on performance](#). Statistics-based metrics are collected every minute from the RocksDB state stores. If a state store consists of multiple RocksDB instances, as is the case for WindowStores and SessionStores, each metric reports an aggregation over the RocksDB instances of the state store.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
bytes-written-rate	The average number of bytes written per second to the RocksDB state store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
bytes-written-total	The total number of bytes written to the RocksDB state store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
bytes-read-rate	The average number of bytes read per second from the RocksDB state store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
bytes-read-total	The total number of bytes read from the RocksDB state store.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-bytes-flushed-rate	The average number of bytes flushed per second from the memtable to disk.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-bytes-flushed-total	The total number of bytes flushed from the memtable to disk.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-hit-ratio	The ratio of memtable hits relative to all lookups to the memtable.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-flush-time-avg	The average duration of memtable flushes to disc in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-flush-time-min	The minimum duration of memtable flushes to disc in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
memtable-flush-time-max	The maximum duration of memtable flushes to disc in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
block-cache-data-hit-ratio	The ratio of block cache hits for data blocks relative to all lookups for data blocks to the block cache.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
block-cache-index-hit-ratio	The ratio of block cache hits for index blocks relative to all lookups for index blocks to the block cache.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
block-cache-filter-hit-ratio	The ratio of block cache hits for filter blocks relative to all lookups for filter blocks to the block cache.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)
write-stall-duration-avg	The average duration of write stalls in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w+]),[store-scope]-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
write-stall-duration-total	The total duration of write stalls in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
bytes-read-compaction-rate	The average number of bytes read per second during compaction.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
bytes-written-compaction-rate	The average number of bytes written per second during compaction.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
compaction-time-avg	The average duration of disc compactions in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
compaction-time-min	The minimum duration of disc compactions in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
compaction-time-max	The maximum duration of disc compactions in ms.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
number-open-files	The number of current open files.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
number-file-errors-total	The total number of file errors occurred.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)

**RocksDB Properties-based Metrics:** All of the following properties-based metrics have a recording level of `info` and are recorded when the metrics are accessed. If a state store consists of multiple RocksDB instances, as is the case for WindowStores and SessionStores, each metric reports the sum over all the RocksDB instances of the state store, except for the block cache metrics `block-cache-*`. The block cache metrics report the sum over all RocksDB instances if each instance uses its own block cache, and they report the recorded value from only one instance if a single block cache is shared among all instances.

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
num-immutable-mem-table	The number of immutable memtables that have not yet been flushed.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
cur-size-active-mem-table	The approximate size of the active memtable in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
cur-size-all-mem-tables	The approximate size of active and unflushed immutable memtables in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
size-all-mem-tables	The approximate size of active, unflushed immutable, and pinned immutable memtables in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-entries-active-mem-table	The number of entries in the active memtable.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-entries-imm-mem-tables	The number of entries in the unflushed immutable memtables.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-deletes-active-mem-table	The number of delete entries in the active memtable.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-deletes-imm-mem-tables	The number of delete entries in the unflushed immutable memtables.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
mem-table-flush-pending	This metric reports 1 if a memtable flush is pending, otherwise it reports 0.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-running-flushes	The number of currently running flushes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
compaction-pending	This metric reports 1 if at least one compaction is pending, otherwise it reports 0.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
num-running-compactions	The number of currently running compactions.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
estimate-pending-compaction-bytes	The estimated total number of bytes a compaction needs to rewrite on disk to get all levels down to under target size (only valid for level compaction).	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
total-sst-files-size	The total size in bytes of all SST files.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
live-sst-files-size	The total size in bytes of all SST files that belong to the latest LSM tree.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
num-live-versions	Number of live versions of the LSM tree.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
block-cache-capacity	The capacity of the block cache in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
block-cache-usage	The memory size of the entries residing in block cache in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
block-cache-pinned-usage	The memory size for the entries being pinned in the block cache in bytes.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
estimate-num-keys	The estimated number of keys in the active and unflushed immutable memtables and storage.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
estimate-table-readers-mem	The estimated memory in bytes used for reading SST tables, excluding memory used in block cache.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)
background-errors	The total number of background errors.	kafka.streams:type=stream-state-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),[store-scope]-id=([-.\w]+)

## Record Cache Metrics

All of the following metrics have a recording level of `debug`:

METRIC/ATTRIBUTE NAME	DESCRIPTION	MBEAN NAME
hit-ratio-avg	The average cache hit ratio defined as the ratio of cache read hits over the total cache read requests.	kafka.streams:type=stream-record-cache-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)
hit-ratio-min	The minimum cache hit ratio.	kafka.streams:type=stream-record-cache-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)
hit-ratio-max	The maximum cache hit ratio.	kafka.streams:type=stream-record-cache-metrics,thread-id=([-.\w]+),task-id=([-.\w]+),record-cache-id=([-.\w]+)

## Others

We recommend monitoring GC time and other stats and various server stats such as CPU utilization, I/O service time, etc. On the client side, we recommend monitoring the message/byte rate (global and per topic), request rate/size/time, and on the consumer side, max lag in messages among all partitions and min fetch request rate. For a consumer to keep up, max lag needs to be less than a threshold and min fetch rate needs to be larger than 0.

## 6.9 ZooKeeper

### Stable version

The current stable branch is 3.5. Kafka is regularly updated to include the latest release in the 3.5 series.

### Operationalizing ZooKeeper

Operationally, we do the following for a healthy ZooKeeper installation:

- Redundancy in the physical/hardware/network layout: try not to put them all in the same rack, decent (but don't go nuts) hardware, try to keep redundant power and network paths, etc. A typical ZooKeeper ensemble has 5 or 7 servers, which tolerates 2 and 3 servers down, respectively. If you have a small deployment, then using 3 servers is acceptable, but keep in mind that you'll only be able to tolerate 1 server down in this case.
- I/O segregation: if you do a lot of write type traffic you'll almost definitely want the transaction logs on a dedicated disk group. Writes to the transaction log are synchronous (but batched for performance), and consequently, concurrent writes can significantly affect performance. ZooKeeper snapshots can be one such a source of concurrent writes, and ideally should be written on a disk group separate from the transaction log. Snapshots are written to disk asynchronously, so it is typically ok to share with the operating system and message log files. You can configure a server to use a separate disk group with the `dataLogDir` parameter.
- Application segregation: Unless you really understand the application patterns of other apps that you want to install on the same box, it can be a good idea to run ZooKeeper in isolation (though this can be a balancing act with the capabilities of the hardware).
- Use care with virtualization: It can work, depending on your cluster layout and read/write patterns and SLAs, but the tiny overheads introduced by the virtualization layer can add up and throw off ZooKeeper, as it can be very time sensitive

- ZooKeeper configuration: It's java, make sure you give it 'enough' heap space (We usually run them with 3-5G, but that's mostly due to the data set size we have here). Unfortunately we don't have a good formula for it, but keep in mind that allowing for more ZooKeeper state means that snapshots can become large, and large snapshots affect recovery time. In fact, if the snapshot becomes too large (a few gigabytes), then you may need to increase the initLimit parameter to give enough time for servers to recover and join the ensemble.
- Monitoring: Both JMX and the 4 letter words (4lw) commands are very useful, they do overlap in some cases (and in those cases we prefer the 4 letter commands, they seem more predictable, or at the very least, they work better with the LI monitoring infrastructure)
- Don't overbuild the cluster: large clusters, especially in a write heavy usage pattern, means a lot of intracluster communication (quorums on the writes and subsequent cluster member updates), but don't underbuild it (and risk swamping the cluster). Having more servers adds to your read capacity.

Overall, we try to keep the ZooKeeper system as small as will handle the load (plus standard growth capacity planning) and as simple as possible. We try not to do anything fancy with the configuration or application layout as compared to the official release as well as keep it as self contained as possible. For these reasons, we tend to skip the OS packaged versions, since it has a tendency to try to put things in the OS standard hierarchy, which can be 'messy', for want of a better way to word it.

## [6.10 KRaft](#)

### [Configuration](#)

#### [Process Roles](#)

In KRaft mode each Kafka server can be configured as a controller, a broker, or both using the `process.roles` property. This property can have the following values:

- If `process.roles` is set to `broker`, the server acts as a broker.
- If `process.roles` is set to `controller`, the server acts as a controller.
- If `process.roles` is set to `broker,controller`, the server acts as both a broker and a controller.
- If `process.roles` is not set at all, it is assumed to be in ZooKeeper mode.

Kafka servers that act as both brokers and controllers are referred to as "combined" servers. Combined servers are simpler to operate for small use cases like a development environment. The key disadvantage is that the controller will be less isolated from the rest of the system. For example, it is not possible to roll or scale the controllers separately from the brokers in combined mode. Combined mode is not recommended in critical deployment environments.

#### [Controllers](#)

In KRaft mode, specific Kafka servers are selected to be controllers (unlike the ZooKeeper-based mode, where any server can become the Controller). The servers selected to be controllers will participate in the metadata quorum. Each controller is either an active or a hot standby for the current active controller.

A Kafka admin will typically select 3 or 5 servers for this role, depending on factors like cost and the number of concurrent failures your system should withstand without availability impact. A majority of the controllers must be alive in order to maintain availability. With 3 controllers, the cluster can tolerate 1 controller failure; with 5 controllers, the cluster can tolerate 2 controller failures.

All of the servers in a Kafka cluster discover the quorum voters using the `controller.quorum.voters` property. This identifies the quorum controller servers that should be used. All the controllers must be enumerated. Each controller is identified with their `id`, `host` and `port` information. For example:

```
controller.quorum.voters=id1@host1:port1,id2@host2:port2,id3@host3:port3
```

If a Kafka cluster has 3 controllers named controller1, controller2 and controller3, then controller1 may have the following configuration:

```
process.roles=controller
node.id=1
listeners=CONTROLLER://controller1.example.com:9093
controller.quorum.voters=1@controller1.example.com:9093,2@controller2.example.com:9093
,3@controller3.example.com:9093
```

Every broker and controller must set the `controller.quorum.voters` property. The node ID supplied in the `controller.quorum.voters` property must match the corresponding id on the controller servers. For example, on controller1, node.id must be set to 1, and so forth. Each node ID must be unique across all the servers in a particular cluster. No two servers can have the same node ID regardless of their `process.roles` values.

## [Storage Tool](#)

The `kafka-storage.sh random-uuid` command can be used to generate a cluster ID for your new cluster. This cluster ID must be used when formatting each server in the cluster with the `kafka-storage.sh format` command.

This is different from how Kafka has operated in the past. Previously, Kafka would format blank storage directories automatically, and also generate a new cluster ID automatically. One reason for the change is that auto-formatting can sometimes obscure an error condition. This is particularly important for the metadata log maintained by the controller and broker servers. If a majority of the controllers were able to start with an empty log directory, a leader might be able to be elected with missing committed data.

## [Debugging](#)

### [Metadata Quorum Tool](#)

The `kafka-metadata-quorum` tool can be used to describe the runtime state of the cluster metadata partition. For example, the following command displays a summary of the metadata quorum:

```
> bin/kafka-metadata-quorum.sh --bootstrap-server broker_host:port describe --
status
ClusterId: fMCL8kv1Swm87L_Md-I2hg
LeaderId: 3002
LeaderEpoch: 2
Highwatermark: 10
MaxFollowerLag: 0
MaxFollowerLagTimeMs: -1
CurrentVoters: [3000,3001,3002]
CurrentObservers: [0,1,2]
```

### [Dump Log Tool](#)

The `kafka-dump-log` tool can be used to debug the log segments and snapshots for the cluster metadata directory. The tool will scan the provided files and decode the metadata records. For example, this command decodes and prints the records in the first log segment:

```
> bin/kafka-dump-log.sh --cluster-metadata-decoder --files
metadata_log_dir/_cluster_metadata-0/00000000000000000000000000000000.log
```

This command decodes and prints the records in the a cluster metadata snapshot:

```
> bin/kafka-dump-log.sh --cluster-metadata-decoder --files
metadata_log_dir/_cluster_metadata-0/0000000000000000100-0000000001.checkpoint
```

## Metadata Shell

The `kafka-metadata-shell` tool can be used to interactively inspect the state of the cluster metadata partition:

```
> bin/kafka-metadata-shell.sh --snapshot metadata_log_dir/_cluster_metadata-0/000000000000000000000000.log
>> ls /
brokers local metadataQuorum topicIds topics
>> ls /topics
foo
>> cat /topics/foo/0/data
{
  "partitionId" : 0,
  "topicId" : "5zoAlv-xEh9xRANKxt1Lbg",
  "replicas" : [ 1 ],
  "isr" : [ 1 ],
  "removingReplicas" : null,
  "addingReplicas" : null,
  "leader" : 1,
  "leaderEpoch" : 0,
  "partitionEpoch" : 0
}
>> exit
```

## Deploying Considerations

- Kafka server's `process.role` should be set to either `broker` or `controller` but not both. Combined mode can be used in development environment but it should be avoided in critical deployment environments.
- For redundancy, a Kafka cluster should use 3 controllers. More than 3 servers is not recommended in critical environments. In the rare case of a partial network failure it is possible for the cluster metadata quorum to become unavailable. This limitation will be addressed in a future release of Kafka.
- The Kafka controllers store all of the metadata for the cluster in memory and on disk. We believe that for a typical Kafka cluster 5GB of main memory and 5GB of disk space on the metadata log director is sufficient.

## Missing Features

The following features are not fully implemented in KRaft mode:

- Configuring SCRAM users via the administrative API
- Supporting JBOD configurations with multiple storage directories
- Modifying certain dynamic configurations on the standalone KRaft controller
- Delegation tokens

## ZooKeeper to KRaft Migration

**The ZooKeeper to KRaft migration feature is considered Early Access in 3.4.0. It is not recommended for production clusters.**

The following features are not yet supported for ZK to KRaft migrations:

- Downgrading to ZooKeeper mode during or after the migration
- Migration of ACLs
- Other features [not yet supported in KRaft](#)

Please report issues with ZooKeeper to KRaft migration using the [project JIRA](#) and the "kraft" component.

# Terminology

In this documentation, we use the term "migration" to refer to the process to changing a Kafka cluster's metadata system from ZooKeeper to KRaft. An "upgrade" refers to installing a newer version of Kafka. It is not recommended to upgrade the software at the same time as performing a migration.

This documentation uses the term "ZK mode" to refer to Kafka brokers which are using ZooKeeper as their metadata system. "KRaft mode" refers Kafka brokers which are using a KRaft controller quorum as their metadata system.

## Preparing for migration

Before beginning the migration, the Kafka brokers must be upgraded to software version 3.4.0 and have the "inter.broker.protocol.version" configuration set to "3.4". See [Upgrading to 3.4.0](#) for upgrade instructions.

It is recommended to enable TRACE level logging for the migration components while the migration is active. This can be done by adding the following log4j configuration to each KRaft controller's "log4j.properties" file.

```
log4j.logger.org.apache.kafka.metadata.migration=TRACE
```

It may generally useful to enable DEBUG logging on the KRaft controllers and the ZK brokers during the migration.

## Provisioning the KRaft controller quorum

Two things are needed before the migration can begin. The brokers must be configured to support the migration and a KRaft controller quorum must be deployed. The KRaft controllers should be provisioned with the same cluster ID as the existing Kafka cluster. This can be found by examining one of the "meta.properties" files in the data directories of the brokers, or by running the following command.

```
./bin/zookeeper-shell.sh localhost:2181 get /cluster/id
```

The KRaft controller quorum should also be provisioned with the latest `metadata.version` of "3.4". For further instructions on KRaft deployment, please refer to [the above documentation](#).

In addition to the standard KRaft configuration, the KRaft controllers will need to enable support for the migration as well as provide ZooKeeper connection configuration.

```
# Sample KRaft cluster controller.properties listening on 9093
process.roles=controller
node.id=3000
controller.quorum.voters=3000@localhost:9093
controller.listener.names=CONTROLLER
listeners=CONTROLLER://:9093

# Enable the migration
zookeeper.metadata.migration.enable=true

# ZooKeeper client configuration
zookeeper.connect=localhost:2181

# Other configs ...
```

*Note: The KRaft cluster `node.id` values must be different from any existing ZK broker `broker.id`. In KRaft-mode, the brokers and controllers share the same Node ID namespace.*

## Enabling the migration on the brokers

Once the KRaft controller quorum has been started, the brokers will need to be reconfigured and restarted. Brokers may be restarted in a rolling fashion to avoid impacting cluster availability. Each broker will need to add the following configurations to allow it to communicate with the KRaft controllers and to enable the migration.

- [controller.quorum.voters](#)
- [controller.listener.names](#)
- The controller.listener.name should also be added to [listener.security.property.map](#)
- [zookeeper.metadata.migration.enable](#)

Here is a sample config for a broker that is ready for migration:

```
# Sample ZK broker server.properties listening on 9092
broker.id=0
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://localhost:9092
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT

# Set the IBP
inter.broker.protocol.version=3.4

# Enable the migration
zookeeper.metadata.migration.enable=true

# ZooKeeper client configuration
zookeeper.connect=localhost:2181

# KRaft controller quorum configuration
controller.quorum.voters=3000@localhost:9093
controller.listener.names=CONTROLLER
```

*Note: Once the final ZK broker has been restarted with the necessary configuration, the migration will automatically begin.* When the migration is complete, a INFO level log can be observed on the active controller.

Completed migration of metadata from zookeeper to KRaft

## Migrating brokers to KRaft

Once the KRaft controller completes the metadata migration, the brokers will still be running in ZK mode. While the KRaft controller is in migration mode, it will continue sending controller RPCs to the ZK mode brokers. This includes RPCs like UpdateMetadata and LeaderAndlsr.

To migrate the brokers to KRaft, they simply need to be reconfigured as KRaft brokers and restarted. Using the above broker configuration as an example, we would replace the `broker.id` with `node.id` and add `process.roles=broker`. It is important that the broker maintain the same Broker/Node ID when it is restarted. The zookeeper configurations should be removed at this point.

```
# Sample KRaft broker server.properties listening on 9092
process.roles=broker
node.id=0
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://localhost:9092
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT

# Don't set the IBP, KRaft uses "metadata.version" feature flag
```

```

# inter.broker.protocol.version=3.4

# Remove the migration enabled flag
# zookeeper.metadata.migration.enable=true

# Remove ZooKeeper client configuration
# zookeeper.connect=localhost:2181

# Keep the KRaft controller quorum configuration
controller.quorum.voters=3000@localhost:9093
controller.listener.names=CONTROLLER

```

Each broker is restarted with a KRaft configuration until the entire cluster is running in KRaft mode.

## Finalizing the migration

Once all brokers have been restarted in KRaft mode, the last step to finalize the migration is to take the KRaft controllers out of migration mode. This is done by removing the "zookeeper.metadata.migration.enable" property from each of their configs and restarting them one at a time.

```

# Sample KRaft cluster controller.properties listening on 9093
process.roles=controller
node.id=3000
controller.quorum.voters=1@localhost:9093
controller.listener.names=CONTROLLER
listeners=CONTROLLER://:9093

# Disable the migration
# zookeeper.metadata.migration.enable=true

# Remove ZooKeeper client configuration
# zookeeper.connect=localhost:2181

# Other configs ...

```

## 7. SECURITY

### 7.1 Security Overview

In release 0.9.0.0, the Kafka community added a number of features that, used either separately or together, increases security in a Kafka cluster. The following security measures are currently supported:

1. Authentication of connections to brokers from clients (producers and consumers), other brokers and tools, using either SSL or SASL. Kafka supports the following SASL mechanisms:
  - o SASL/GSSAPI (Kerberos) - starting at version 0.9.0.0
  - o SASL/PLAIN - starting at version 0.10.0.0
  - o SASL/SCRAM-SHA-256 and SASL/SCRAM-SHA-512 - starting at version 0.10.2.0
  - o SASL/OAUTHBEARER - starting at version 2.0
2. Authentication of connections from brokers to ZooKeeper
3. Encryption of data transferred between brokers and clients, between brokers, or between brokers and tools using SSL (Note that there is a performance degradation when SSL is enabled, the magnitude of which depends on the CPU type and the JVM implementation.)
4. Authorization of read / write operations by clients
5. Authorization is pluggable and integration with external authorization services is supported

It's worth noting that security is optional - non-secured clusters are supported, as well as a mix of authenticated, unauthenticated, encrypted and non-encrypted clients. The guides below explain how to configure and use the security features in both clients and brokers.

## 7.2 Listener Configuration

In order to secure a Kafka cluster, it is necessary to secure the channels that are used to communicate with the servers. Each server must define the set of listeners that are used to receive requests from clients as well as other servers. Each listener may be configured to authenticate clients using various mechanisms and to ensure traffic between the server and the client is encrypted. This section provides a primer for the configuration of listeners.

Kafka servers support listening for connections on multiple ports. This is configured through the `listeners` property in the server configuration, which accepts a comma-separated list of the listeners to enable. At least one listener must be defined on each server. The format of each listener defined in `listeners` is given below:

```
{LISTENER_NAME}://:{hostname}:{port}
```

The `LISTENER_NAME` is usually a descriptive name which defines the purpose of the listener. For example, many configurations use a separate listener for client traffic, so they might refer to the corresponding listener as `CLIENT` in the configuration:

```
listeners=CLIENT://localhost:9092
```

The security protocol of each listener is defined in a separate configuration:

`listener.security.protocol.map`. The value is a comma-separated list of each listener mapped to its security protocol. For example, the follow value configuration specifies that the `CLIENT` listener will use SSL while the `BROKER` listener will use plaintext.

```
listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT
```

Possible options for the security protocol are given below:

1. PLAINTEXT
2. SSL
3. SASL\_PLAINTEXT
4. SASL\_SSL

The plaintext protocol provides no security and does not require any additional configuration. In the following sections, this document covers how to configure the remaining protocols.

If each required listener uses a separate security protocol, it is also possible to use the security protocol name as the listener name in `listeners`. Using the example above, we could skip the definition of the `CLIENT` and `BROKER` listeners using the following definition:

```
listeners=SSL://localhost:9092,PLAINTEXT://localhost:9093
```

However, we recommend users to provide explicit names for the listeners since it makes the intended usage of each listener clearer.

Among the listeners in this list, it is possible to declare the listener to be used for inter-broker communication by setting the `inter.broker.listener.name` configuration to the name of the listener. The primary purpose of the inter-broker listener is partition replication. If not defined, then the inter-broker listener is determined by the security protocol defined by `security.inter.broker.protocol`, which defaults to `PLAINTEXT`.

For legacy clusters which rely on Zookeeper to store cluster metadata, it is possible to declare a separate listener to be used for metadata propagation from the active controller to the brokers. This is defined by `control_plane.listener.name`. The active controller will use this listener when it needs to push metadata updates to the brokers in the cluster. The benefit of using a control plane listener is that it uses a separate processing thread, which makes it less likely for application traffic to impede timely propagation of metadata changes (such as partition leader and ISR updates). Note that the default value is null, which means that the controller will use the same listener defined by `inter.broker.listener`.

In a KRaft cluster, a broker is any server which has the `broker` role enabled in `process.roles` and a controller is any server which has the `controller` role enabled. Listener configuration depends on the role. The listener defined by `inter.broker.listener.name` is used exclusively for requests between brokers. Controllers, on the other hand, must use separate listener which is defined by the `controller.listener.names` configuration. This cannot be set to the same value as the inter-broker listener.

Controllers receive requests both from other controllers and from brokers. For this reason, even if a server does not have the `controller` role enabled (i.e. it is just a broker), it must still define the controller listener along with any security properties that are needed to configure it. For example, we might use the following configuration on a standalone broker:

```
process.roles=broker
listeners=BROKER://localhost:9092
inter.broker.listener.name=BROKER
controller.quorum.voters=0@localhost:9093
controller.listener.names=CONTROLLER
listener.security.protocol.map=BROKER:SASL_SSL,CONTROLLER:SASL_SSL
```

The controller listener is still configured in this example to use the `SASL_SSL` security protocol, but it is not included in `listeners` since the broker does not expose the controller listener itself. The port that will be used in this case comes from the `controller.quorum.voters` configuration, which defines the complete list of controllers.

For KRaft servers which have both the broker and controller role enabled, the configuration is similar. The only difference is that the controller listener must be included in `listeners`:

```
process.roles=broker,controller
listeners=BROKER://localhost:9092,CONTROLLER://localhost:9093
inter.broker.listener.name=BROKER
controller.quorum.voters=0@localhost:9093
controller.listener.names=CONTROLLER
listener.security.protocol.map=BROKER:SASL_SSL,CONTROLLER:SASL_SSL
```

It is a requirement for the port defined in `controller.quorum.voters` to exactly match one of the exposed controller listeners. For example, here the `CONTROLLER` listener is bound to port 9093. The connection string defined by `controller.quorum.voters` must then also use port 9093, as it does here.

The controller will accept requests on all listeners defined by `controller.listener.names`. Typically there would be just one controller listener, but it is possible to have more. For example, this provides a way to change the active listener from one port or security protocol to another through a roll of the cluster (one roll to expose the new listener, and one roll to remove the old listener). When multiple controller listeners are defined, the first one in the list will be used for outbound requests.

It is conventional in Kafka to use a separate listener for clients. This allows the inter-cluster listeners to be isolated at the network level. In the case of the controller listener in KRaft, the listener should be isolated since clients do not work with it anyway. Clients are expected to connect to any other listener configured on a broker. Any requests that are bound for the controller will be forwarded as described [below](#).

In the following [section](#), this document covers how to enable SSL on a listener for encryption as well as authentication. The subsequent [section](#) will then cover additional authentication mechanisms using SASL.

## **7.3 Encryption and Authentication using SSL**

Apache Kafka allows clients to use SSL for encryption of traffic as well as authentication. By default, SSL is disabled but can be turned on if needed. The following paragraphs explain in detail how to set up your own PKI infrastructure, use it to create certificates and configure Kafka to use these.

### **1. Generate SSL key and certificate for each Kafka broker**

The first step of deploying one or more brokers with SSL support is to generate a public/private keypair for every server. Since Kafka expects all keys and certificates to be stored in keystores we will use Java's keytool command for this task. The tool supports two different keystore formats, the Java specific jks format which has been deprecated by now, as well as PKCS12. PKCS12 is the default format as of Java version 9, to ensure this format is being used regardless of the Java version in use all following commands explicitly specify the PKCS12 format.

```
> keytool -keystore {keystorefile} -alias localhost -validity {validity} -genkey -keyalg RSA -storetype pkcs12
```

You need to specify two parameters in the above command:

1. keystorefile: the keystore file that stores the keys (and later the certificate) for this broker. The keystore file contains the private and public keys of this broker, therefore it needs to be kept safe. Ideally this step is run on the Kafka broker that the key will be used on, as this key should never be transmitted/leave the server that it is intended for.
2. validity: the valid time of the key in days. Please note that this differs from the validity period for the certificate, which will be determined in [Signing the certificate](#). You can use the same key to request multiple certificates: if your key has a validity of 10 years, but your CA will only sign certificates that are valid for one year, you can use the same key with 10 certificates over time.

To obtain a certificate that can be used with the private key that was just created a certificate signing request needs to be created. This signing request, when signed by a trusted CA results in the actual certificate which can then be installed in the keystore and used for authentication purposes.

To generate certificate signing requests run the following command for all server keystores created so far.

```
> keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genkey -keyalg RSA -deststoretype pkcs12 -ext SAN=DNS:{FQDN},IP:{IPADDRESS1}
```

This command assumes that you want to add hostname information to the certificate, if this is not the case, you can omit the extension parameter

```
-ext SAN=DNS:{FQDN},IP:{IPADDRESS1}
```

. Please see below for more information on this.

### **Host Name Verification**

Host name verification, when enabled, is the process of checking attributes from the certificate that is presented by the server you are connecting to against the actual hostname or ip address of that server to ensure that you are indeed connecting to the correct server.

The main reason for this check is to prevent man-in-the-middle attacks. For Kafka, this check has been disabled by default for a long time, but as of Kafka 2.0.0 host name verification of servers is enabled by default for client connections as well as inter-broker connections.

Server host name verification may be disabled by setting

```
ssl.endpoint.identification.algorithm
```

to an empty string.

For dynamically configured broker listeners, hostname verification may be disabled using

```
kafka-configs.sh
```

:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9093 --entity-type brokers --  
entity-name 0 --alter --add-config  
"listener.name.internal.ssl.endpoint.identification.algorithm=""
```

**Note:**

Normally there is no good reason to disable hostname verification apart from being the quickest way to "just get it to work" followed by the promise to "fix it later when there is more time"!

Getting hostname verification right is not that hard when done at the right time, but gets much harder once the cluster is up and running - do yourself a favor and do it now!

If host name verification is enabled, clients will verify the server's fully qualified domain name (FQDN) or ip address against one of the following two fields:

1. Common Name (CN)
2. [Subject Alternative Name \(SAN\)](#).

While Kafka checks both fields, usage of the common name field for hostname verification has been

deprecated

since 2000 and should be avoided if possible. In addition the SAN field is much more flexible, allowing for multiple DNS and IP entries to be declared in a certificate.

Another advantage is that if the SAN field is used for hostname verification the common name can be set to a more meaningful value for authorization purposes. Since we need the SAN field to be contained in the signed certificate, it will be specified when generating the signing request. It can also be specified when generating the keypair, but this will not automatically be copied into the signing request.

To add a SAN field append the following argument

```
-ext SAN=DNS:{FQDN},IP:{IPADDRESS}
```

to the keytool command:

```
> keytool -keystore server.keystore.jks -alias localhost -validity {validity} -  
genkey -keyalg RSA -destkeystoretype pkcs12 -ext SAN=DNS:{FQDN},IP:{IPADDRESS1}
```

## 2. [Creating your own CA](#)

After this step each machine in the cluster has a public/private key pair which can already be used to encrypt traffic and a certificate signing request, which is the basis for creating a certificate. To add authentication capabilities this signing request needs to be signed by a trusted authority, which will be created in this step.

A certificate authority (CA) is responsible for signing certificates. CAs work like a government that issues passports - the government stamps (signs) each passport so that the passport becomes difficult to forge. Other governments verify the stamps to ensure the passport is authentic.

Similarly, the CA signs the certificates, and the cryptography guarantees that a signed certificate is computationally difficult to forge. Thus, as long as the CA is a genuine and trusted authority, the clients have a strong assurance that they are connecting to the authentic machines.

For this guide we will be our own Certificate Authority. When setting up a production cluster in a corporate environment these certificates would usually be signed by a corporate CA that is trusted throughout the company. Please see [Common Pitfalls in Production](#) for some things to consider for this case.

Due to a [bug](#) in OpenSSL, the x509 module will not copy requested extension fields from CSRs into the final certificate. Since we want the SAN extension to be present in our certificate to enable hostname verification, we'll use the `ca` module instead. This requires some additional configuration to be in place before we generate our CA keypair.

Save the following listing into a file called `openssl-ca.cnf` and adjust the values for validity and common attributes as necessary.

```
HOME          = .
RANDFILE      = $ENV::HOME/.rnd

#####
[ ca ]
default_ca    = CA_default      # The default ca section

[ CA_default ]

base_dir      = .
certificate   = $base_dir/cacert.pem      # The CA certificate
private_key   = $base_dir/cakey.pem       # The CA private key
new_certs_dir = $base_dir                # Location for new certs after signing
database      = $base_dir/index.txt      # Database index file
serial        = $base_dir/serial.txt      # The current serial number

default_days   = 1000                  # How long to certify for
default_crl_days = 30                  # How long before next CRL
default_md     = sha256                # Use public key default MD
preserve       = no                    # Keep passed DN ordering

x509_extensions = ca_extensions # The extensions to add to the cert

email_in_dn    = no                   # Don't concat the email in the DN
copy_extensions = copy                # Required to copy SANs from CSR to cert

#####
[ req ]
default_bits   = 4096
default_keyfile = cakey.pem
distinguished_name = ca_distinguished_name
```

```

x509_extensions      = ca_extensions
string_mask          = utf8only

#####
[ ca_distinguished_name ]
countryName          = Country Name (2 letter code)
countryName_default  = DE

stateOrProvinceName   = State or Province Name (full name)
stateOrProvinceName_default = Test Province

localityName          = Locality Name (eg, city)
localityName_default  = Test Town

organizationName       = Organization Name (eg, company)
organizationName_default = Test Company

organizationalUnitName = Organizational Unit (eg, division)
organizationalUnitName_default = Test Unit

commonName            = Common Name (e.g. server FQDN or YOUR name)
commonName_default    = Test Name

emailAddress          = Email Address
emailAddress_default  = test@test.com

#####
[ ca_extensions ]

subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints     = critical, CA:true
keyUsage              = keyCertSign, CRLSign

#####
[ signing_policy ]
countryName          = optional
stateOrProvinceName   = optional
localityName          = optional
organizationName       = optional
organizationalUnitName = optional
commonName            = supplied
emailAddress          = optional

#####
[ signing_req ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer
basicConstraints     = CA:FALSE
keyUsage              = digitalSignature, keyEncipherment

```

Then create a database and serial number file, these will be used to keep track of which certificates were signed with this CA. Both of these are simply text files that reside in the same directory as your CA keys.

```

> echo 01 > serial.txt
> touch index.txt

```

With these steps done you are now ready to generate your CA that will be used to sign certificates later.

```
> openssl req -x509 -config openssl-ca.cnf -newkey rsa:4096 -sha256 -nodes -out cacert.pem -outform PEM
```

The CA is simply a public/private key pair and certificate that is signed by itself, and is only intended to sign other certificates.

This keypair should be kept very safe, if someone gains access to it, they can create and sign certificates that will be trusted by your infrastructure, which means they will be able to impersonate anybody when connecting to any service that trusts this CA.

The next step is to add the generated CA to the **clients' truststore** so that the clients can trust this CA:

```
> keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

Note:

If you configure the Kafka brokers to require client authentication by setting ssl.client.auth to be "requested" or "required" in the

Kafka brokers config

then you must provide a truststore for the Kafka brokers as well and it should have all the CA certificates that clients' keys were signed by.

```
> keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

In contrast to the keystore in step 1 that stores each machine's own identity, the truststore of a client stores all the certificates that the client should trust. Importing a certificate into one's truststore also means trusting all certificates that are signed by that certificate. As the analogy above, trusting the government (CA) also means trusting all passports (certificates) that it has issued. This attribute is called the chain of trust, and it is particularly useful when deploying SSL on a large Kafka cluster. You can sign all certificates in the cluster with a single CA, and have all machines share the same truststore that trusts the CA. That way all machines can authenticate all other machines.

### 3. [Signing the certificate](#)

Then sign it with the CA:

```
> openssl ca -config openssl-ca.cnf -policy signing_policy -extensions signing_req -out {server certificate} -infiles {certificate signing request}
```

Finally, you need to import both the certificate of the CA and the signed certificate into the keystore:

```
> keytool -keystore {keystore} -alias CARoot -import -file {CA certificate}  
> keytool -keystore {keystore} -alias localhost -import -file cert-signed
```

The definitions of the parameters are the following:

1. keystore: the location of the keystore
2. CA certificate: the certificate of the CA
3. certificate signing request: the csr created with the server key
4. server certificate: the file to write the signed certificate of the server to

This will leave you with one truststore called

truststore.jks

- this can be the same for all clients and brokers and does not contain any sensitive information, so there is no need to secure this.

Additionally you will have one

server.keystore.jks

file per node which contains that nodes keys, certificate and your CAs certificate, please refer to

Configuring Kafka Brokers

and

Configuring Kafka Clients

for information on how to use these files.

For some tooling assistance on this topic, please check out the [easyRSA](#) project which has extensive scripting in place to help with these steps.

### **SSL key and certificates in PEM format**

From 2.7.0 onwards, SSL key and trust stores can be configured for Kafka brokers and clients directly in the configuration in PEM format. This avoids the need to store separate files on the file system and benefits from password protection features of Kafka configuration. PEM may also be used as the store type for file-based key and trust stores in addition to JKS and PKCS12. To configure PEM key store directly in the broker or client configuration, private key in PEM format should be provided in

`ssl.keystore.key`

and the certificate chain in PEM format should be provided in

`ssl.keystore.certificate.chain`

. To configure trust store, trust certificates, e.g. public certificate of CA, should be provided in

`ssl.truststore.certificates`

. Since PEM is typically stored as multi-line base-64 strings, the configuration value can be included in Kafka configuration as multi-line strings with lines terminating in backslash ('\') for line continuation.

Store password configs `ssl.keystore.password` and `ssl.truststore.password` are not used for PEM. If private key is encrypted using a password, the key password must be provided in `ssl.key.password`. Private keys may be provided in unencrypted form without a password. In production deployments, configs should be encrypted or externalized using password protection feature in Kafka in this case. Note that the default SSL engine factory has limited capabilities for decryption of encrypted private keys when external tools like OpenSSL are used for encryption. Third party libraries like BouncyCastle may be integrated with a custom `SSLEngineFactory` to support a wider range of encrypted private keys.

#### 4. [Common Pitfalls in Production](#)

The above paragraphs show the process to create your own CA and use it to sign certificates for your cluster. While very useful for sandbox, dev, test, and similar systems, this is usually not the correct process to create certificates for a production cluster in a corporate environment. Enterprises will normally operate their own CA and users can send in CSRs to be signed with this CA, which has the benefit of users not being responsible to keep the CA secure as well as a central authority that everybody can trust. However it also takes away a lot of control over the process of signing certificates from the user. Quite often the persons operating corporate CAs will apply tight restrictions on certificates that can cause issues when trying to use these certificates with Kafka.

##### 1. [Extended Key Usage](#)

Certificates may contain an extension field that controls the purpose for which the certificate can be used. If this field is empty, there are no restrictions on the usage, but if any usage is specified in here, valid SSL implementations have to enforce these usages.

Relevant usages for Kafka are:

- Client authentication
- Server authentication

Kafka brokers need both these usages to be allowed, as for intra-cluster communication every broker will behave as both the client and the server towards other brokers. It is not uncommon for corporate CAs to have a signing profile for webservers and use this for Kafka as well, which will only contain the

`serverAuth`

`usage` value and cause the SSL handshake to fail.

##### 2. [Intermediate Certificates](#)

Corporate Root CAs are often kept offline for security reasons. To enable day-to-day usage, so called intermediate CAs are created, which are then used to sign the final certificates. When importing a certificate into the keystore that was signed by an intermediate CA it is necessary to provide the entire chain of trust up to the root CA. This can be done by simply *cating* the certificate files into one combined certificate file and then importing this with `keytool`.

##### 3. Failure to copy extension fields

CA operators are often hesitant to copy and requested extension fields from CSRs and prefer to specify these themselves as this makes it harder for a malicious party to obtain certificates with potentially misleading or fraudulent values. It is adviseable to double check signed certificates, whether these contain all requested SAN fields to enable proper hostname verification. The following command can be used to print certificate details to the console, which should be compared with what was originally requested:

```
> openssl x509 -in certificate.crt -text -noout
```

## 5. Configuring Kafka Brokers

If SSL is not enabled for inter-broker communication (see below for how to enable it), both PLAINTEXT and SSL ports will be necessary.

```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

Following SSL configs are needed on the broker side

```
ssl.keystore.location=/var/private/ssl/server.keystore.jks  
ssl.keystore.password=test1234  
ssl.key.password=test1234  
ssl.truststore.location=/var/private/ssl/server.truststore.jks  
ssl.truststore.password=test1234
```

Note: ssl.truststore.password is technically optional but highly recommended. If a password is not set access to the truststore is still available, but integrity checking is disabled. Optional settings that are worth considering:

1. ssl.client.auth=none ("required" => client authentication is required, "requested" => client authentication is requested and client without certs can still connect. The usage of "requested" is discouraged as it provides a false sense of security and misconfigured clients will still connect successfully.)
2. ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. (Default is an empty list)
3. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1 (list out the SSL protocols that you are going to accept from clients. Do note that SSL is deprecated in favor of TLS and using SSL in production is not recommended)
4. ssl.keystore.type=JKS
5. ssl.truststore.type=JKS
6. ssl.secure.random.implementation=SHA1PRNG

If you want to enable SSL for inter-broker communication, add the following to the server.properties file (it defaults to PLAINTEXT)

```
security.inter.broker.protocol=SSL
```

Due to import regulations in some countries, the Oracle implementation limits the strength of cryptographic algorithms available by default. If stronger algorithms are needed (for example, AES with 256-bit keys), the [JCE Unlimited Strength Jurisdiction Policy Files](#) must be obtained and installed in the JDK/JRE. See the [JCA Providers Documentation](#) for more information.

The JRE/JDK will have a default pseudo-random number generator (PRNG) that is used for cryptography operations, so it is not required to configure the implementation used with the `ssl.secure.random.implementation`. However, there are performance issues with some implementations (notably, the default chosen on Linux systems, `NativePRNG`, utilizes a global lock). In cases where performance of SSL connections becomes an issue, consider explicitly setting the implementation to be used. The `SHA1PRNG` implementation is non-blocking, and has shown very good performance characteristics under heavy load (50 MB/sec of produced messages, plus replication traffic, per-broker).

Once you start the broker you should be able to see in the server.log

```
with addresses: PLAINTEXT ->EndPoint(192.168.64.1,9092,PLAINTEXT),SSL ->  
EndPoint(192.168.64.1,9093,SSL)
```

To check quickly if the server keystore and truststore are setup properly you can run the following command

```
> openssl s_client -debug -connect localhost:9093 -tls1
```

(Note: TLSv1 should be listed under ssl.enabled.protocols)

In the output of this command you should see server's certificate:

```
-----BEGIN CERTIFICATE-----  
{variable sized random bytes}  
-----END CERTIFICATE-----  
subject=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=Sriharsha Chintalapani  
issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddress=test@test.com
```

If the certificate does not show up or if there are any other error messages then your keystore is not setup properly.

## 6. [Configuring Kafka Clients](#)

SSL is supported only for the new Kafka Producer and Consumer, the older API is not supported. The configs for SSL will be the same for both producer and consumer.

If client authentication is not required in the broker, then the following is a minimal configuration example:

```
security.protocol=SSL  
ssl.truststore.location=/var/private/ssl/client.truststore.jks  
ssl.truststore.password=test1234
```

Note: ssl.truststore.password is technically optional but highly recommended. If a password is not set access to the truststore is still available, but integrity checking is disabled. If client authentication is required, then a keystore must be created like in step 1 and the following must also be configured:

```
ssl.keystore.location=/var/private/ssl/client.keystore.jks  
ssl.keystore.password=test1234  
ssl.key.password=test1234
```

Other configuration settings that may also be needed depending on our requirements and the broker configuration:

1. ssl.provider (Optional). The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
2. ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.
3. ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. It should list at least one of the protocols configured on the broker side
4. ssl.truststore.type=JKS
5. ssl.keystore.type=JKS

Examples using console-producer and console-consumer:

```
> kafka-console-producer.sh --bootstrap-server localhost:9093 --topic test --  
producer.config client-ssl.properties  
> kafka-console-consumer.sh --bootstrap-server localhost:9093 --topic test --  
consumer.config client-ssl.properties
```

## 7.4 Authentication using SASL

### 1. JAAS configuration

Kafka uses the Java Authentication and Authorization Service ([JAAS](#)) for SASL configuration.

#### 1. JAAS configuration for Kafka brokers

`KafkaServer` is the section name in the JAAS file used by each KafkaServer/Broker. This section provides SASL configuration options for the broker including any SASL client connections made by the broker for inter-broker communication. If multiple listeners are configured to use SASL, the section name may be prefixed with the listener name in lower-case followed by a period, e.g. `sasl_ssl.KafkaServer`.

`Client` section is used to authenticate a SASL connection with zookeeper. It also allows the brokers to set SASL ACL on zookeeper nodes which locks these nodes down so that only the brokers can modify it. It is necessary to have the same principal name across all brokers. If you want to use a section name other than Client, set the system property `zookeeper.sasl.clientconfig` to the appropriate name (e.g., `-Dzookeeper.sasl.clientconfig=zkcClient`).

ZooKeeper uses "zookeeper" as the service name by default. If you want to change this, set the system property `zookeeper.sasl.client.username` to the appropriate name (e.g., `-Dzookeeper.sasl.client.username=zk`).

Brokers may also configure JAAS using the broker configuration property `sasl.jaas.config`. The property name must be prefixed with the listener prefix including the SASL mechanism, i.e. `listener.name.{listenerName}.{saslMechanism}.sasl.jaas.config`. Only one login module may be specified in the config value. If multiple mechanisms are configured on a listener, configs must be provided for each mechanism using the listener and mechanism prefix. For example,

```
listener.name.sasl_ssl.scram-sha-
256.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required \
    username="admin" \
    password="admin-secret";
listener.name.sasl_ssl.plain.sasl.jaas.config=org.apache.kafka.common.security
.plain.PlainLoginModule required \
    username="admin" \
    password="admin-secret" \
    user_admin="admin-secret" \
    user_alice="alice-secret";
```

If JAAS configuration is defined at different levels, the order of precedence used is:

- Broker configuration property `listener.name.{listenerName}.{saslMechanism}.sasl.jaas.config`
- `{listenerName}.KafkaServer` section of static JAAS configuration
- `KafkaServer` section of static JAAS configuration

Note that ZooKeeper JAAS config may only be configured using static JAAS configuration.

See [GSSAPI \(Kerberos\)](#), [PLAIN](#), [SCRAM](#) or [OAUTHBEARER](#) for example broker configurations.

#### 2. JAAS configuration for Kafka clients

Clients may configure JAAS using the client configuration property `sasl.jaas.config` or using the [static JAAS config file](#) similar to brokers.

## 1. JAAS configuration using client configuration property

Clients may specify JAAS configuration as a producer or consumer property without creating a physical configuration file. This mode also enables different producers and consumers within the same JVM to use different credentials by specifying different properties for each client. If both static JAAS configuration system property `java.security.auth.login.config` and client property `sasl.jaas.config` are specified, the client property will be used.

See [GSSAPI \(Kerberos\)](#), [PLAIN](#), [SCRAM](#) or [OAUTHBEARER](#) for example configurations.

## 2. JAAS configuration using static config file

To configure SASL authentication on the clients using static JAAS config file:

1. Add a JAAS config file with a client login section named

KafkaClient

. Configure a login module in

KafkaClient

for the selected mechanism as described in the examples for setting up

GSSAPI (Kerberos)

,

PLAIN

,

SCRAM

or

OAUTHBEARER

. For example,

GSSAPI

credentials may be configured as:

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_client.keytab"  
    principal="kafka-client-1@EXAMPLE.COM";  
};
```

2. Pass the JAAS config file location as JVM parameter to each client JVM. For example:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
```

## 2. [SASL configuration](#)

SASL may be used with PLAINTEXT or SSL as the transport layer using the security protocol SASL\_PLAINTEXT or SASL\_SSL respectively. If SASL\_SSL is used, then [SSL must also be configured](#).

### 1. [SASL mechanisms](#)

Kafka supports the following SASL mechanisms:

- [GSSAPI](#) (Kerberos)
- [PLAIN](#)
- [SCRAM-SHA-256](#)
- [SCRAM-SHA-512](#)
- [OAUTHBEARER](#)

### 2. [SASL configuration for Kafka brokers](#)

1. Configure a SASL port in server.properties, by adding at least one of SASL\_PLAINTEXT or SASL\_SSL to the

listeners

parameter, which contains one or more comma-separated values:

```
listeners=SASL_PLAINTEXT://host.name:port
```

If you are only configuring a SASL port (or if you want the Kafka brokers to authenticate each other using SASL) then make sure you set the same SASL protocol for inter-broker communication:

```
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
```

2. Select one or more [supported mechanisms](#) to enable in the broker and follow the steps to configure SASL for the mechanism. To enable multiple mechanisms in the broker, follow the steps [here](#).

### 3. [SASL configuration for Kafka clients](#)

SASL authentication is only supported for the new Java Kafka producer and consumer, the older API is not supported.

To configure SASL authentication on the clients, select a SASL [mechanism](#) that is enabled in the broker for client authentication and follow the steps to configure SASL for the selected mechanism.

## 3. [Authentication using SASL/Kerberos](#)

### 1. [Prerequisites](#)

#### 1. Kerberos

If your organization is already using a Kerberos server (for example, by using Active Directory), there is no need to install a new server just for Kafka. Otherwise you will need to install one, your Linux vendor likely has packages for Kerberos and a short guide on how to install and configure it ([Ubuntu](#), [Redhat](#)). Note that if you are using Oracle Java, you will need to download JCE policy files for your Java version and copy them to \$JAVA\_HOME/jre/lib/security.

2. Create Kerberos Principals

If you are using the organization's Kerberos or Active Directory server, ask your Kerberos administrator for a principal for each Kafka broker in your cluster and for every operating system user that will access Kafka with Kerberos authentication (via clients and tools).

If you have installed your own Kerberos, you will need to create these principals yourself using the following commands:

```
> sudo /usr/sbin/kadmin.local -q 'addprinc -randkey  
kafka/{hostname}@{REALM}'  
> sudo /usr/sbin/kadmin.local -q "ktadd -k  
/etc/security/keytabs/{keytabname}.keytab kafka/{hostname}@{REALM}"
```

3. **Make sure all hosts can be reachable using hostnames** - it is a Kerberos requirement that all your hosts can be resolved with their FQDNs.

## 2. [Configuring Kafka Brokers](#)

1. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka\_server\_jaas.conf for this example (note that each broker should have its own keytab):

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
};  
  
// zookeeper client authentication  
client {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
};
```

KafkaServer

section in the JAAS file tells the broker which principal to use and the location of the keytab where this principal is stored. It allows the broker to login using the keytab specified in this section. See

notes

for more details on Zookeeper SASL configuration.

2. Pass the JAAS and optionally the krb5 file locations as JVM parameters to each Kafka broker (see

here

for more details):

```
-Djava.security.krb5.conf=/etc/kafka/krb5.conf  
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. Make sure the keytabs configured in the JAAS file are readable by the operating system user who is starting kafka broker.
4. Configure SASL port and SASL mechanisms in server.properties as described here

. For example:

```
listeners=SASL_PLAINTEXT://host.name:port  
security.inter.broker.protocol=SASL_PLAINTEXT  
sasl.mechanism.inter.broker.protocol=GSSAPI  
sasl.enabled.mechanisms=GSSAPI
```

We must also configure the service name in server.properties, which should match the principal name of the kafka brokers. In the above example, principal is "kafka/[kafka1.host.name.com@EXAMPLE.com](#)", so:

```
sasl.kerberos.service.name=kafka
```

### 3. [Configuring Kafka Clients](#)

To configure SASL authentication on the clients:

1. Clients (producers, consumers, connect workers, etc) will authenticate to the cluster with their own principal (usually with the same name as the user running the client), so obtain or create these principals as needed. Then configure the JAAS configuration property for each client. Different clients within a JVM may run as different users by specifying different principals. The property

```
sasl.jaas.config
```

in producer.properties or consumer.properties describes how clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client using a keytab (recommended for long-running processes):

```
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \  
useKeyTab=true \  
storeKey=true \  
keyTab="/etc/security/keytabs/kafka_client.keytab" \  
principal="kafka-client-1@EXAMPLE.COM";
```

For command-line utilities like kafka-console-consumer or kafka-console-producer, kinit can be used along with "useTicketCache=true" as in:

```
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \  
useTicketCache=true;
```

JAAS configuration for clients may alternatively be specified as a JVM parameter similar to brokers as described

here

. Clients use the login section named

KafkaClient

. This option allows only one user for all client connections from a JVM.

2. Make sure the keytabs configured in the JAAS configuration are readable by the operating system user who is starting kafka client.
3. Optionally pass the krb5 file locations as JVM parameters to each client JVM (see

here

for more details):

```
-Djava.security.krb5.conf=/etc/kafka/krb5.conf
```

4. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=kafka
```

#### 4. [Authentication using SASL/PLAIN](#)

SASL/PLAIN is a simple username/password authentication mechanism that is typically used with TLS for encryption to implement secure authentication. Kafka supports a default implementation for SASL/PLAIN which can be extended for production use as described [here](#).

Under the default implementation of

```
principal.builder.class
```

, the username is used as the authenticated

```
Principal
```

for configuration of ACLs etc.

#### 1. [Configuring Kafka Brokers](#)

1. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka\_server\_jaas.conf for this example:

```
kafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
        username="admin"
        password="admin-secret"
        user_admin="admin-secret"
        user_alice="alice-secret";
};
```

This configuration defines two users (

admin

and

alice

). The properties

username

and

password

in the

KafkaServer

section are used by the broker to initiate connections to other brokers. In this example,

admin

is the user for inter-broker communication. The set of properties

*user\_userName*

defines the passwords for all users that connect to the broker and the broker validates all client connections including those from other brokers using these properties.

2. Pass the JAAS config file location as JVM parameter to each Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. Configure SASL port and SASL mechanisms in server.properties as described

here

. For example:

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```

## 2. [Configuring Kafka Clients](#)

To configure SASL authentication on the clients:

1. Configure the JAAS configuration property for each client in producer.properties or consumer.properties. The login module describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client for the PLAIN mechanism:

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \
    username="alice" \
    password="alice-secret";
```

The options `username` and `password` are used by clients to configure the user for client connections. In this example, clients connect to the broker as user *alice*. Different clients within a JVM may connect as different users by specifying different user names and passwords in `sasl.jaas.config`.

JAAS configuration for clients may alternatively be specified as a JVM parameter similar to brokers as described [here](#). Clients use the login section named `kafkaclient`. This option allows only one user for all client connections from a JVM.

2. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
```

## 3. [Use of SASL/PLAIN in production](#)

- SASL/PLAIN should be used only with SSL as transport layer to ensure that clear passwords are not transmitted on the wire without encryption.
- The default implementation of SASL/PLAIN in Kafka specifies usernames and passwords in the JAAS configuration file as shown [here](#). From Kafka version 2.0 onwards, you can avoid storing clear passwords on disk by configuring your own callback handlers that obtain username and password from an external source using the configuration options `sasl.server.callback.handler.class` and `sasl.client.callback.handler.class`.
- In production systems, external authentication servers may implement password authentication. From Kafka version 2.0 onwards, you can plug in your own callback handlers that use external authentication servers for password verification by configuring `sasl.server.callback.handler.class`.

## 5. [Authentication using SASL/SCRAM](#)

Salted Challenge Response Authentication Mechanism (SCRAM) is a family of SASL mechanisms that addresses the security concerns with traditional mechanisms that perform username/password authentication like PLAIN and DIGEST-MD5. The mechanism is defined in [RFC 5802](#). Kafka supports [SCRAM-SHA-256](#) and SCRAM-SHA-512 which can be used with TLS to perform secure authentication. Under the default implementation of `principal.builder.class`, the username is used as the authenticated `Principal` for configuration of ACLs etc. The default SCRAM implementation in Kafka stores SCRAM credentials in Zookeeper and is suitable for use in Kafka installations where Zookeeper is on a private network. Refer to [Security Considerations](#) for more details.

## 1. [Creating SCRAM Credentials](#)

The SCRAM implementation in Kafka uses Zookeeper as credential store. Credentials can be created in Zookeeper using `kafka-configs.sh`. For each SCRAM mechanism enabled, credentials must be created by adding a config with the mechanism name. Credentials for inter-broker communication must be created before Kafka brokers are started. Client credentials may be created and updated dynamically and updated credentials will be used to authenticate new connections.

Create SCRAM credentials for user `alice` with password `alice-secret`:

```
> bin/kafka-configs.sh --zookeeper localhost:2182 --zk-tls-config-file zk_tls_config.properties --alter --add-config 'SCRAM-SHA-256=[iterations=8192,password=alice-secret],SCRAM-SHA-512=[password=alice-secret]' --entity-type users --entity-name alice
```

The default iteration count of 4096 is used if iterations are not specified. A random salt is created and the SCRAM identity consisting of salt, iterations, StoredKey and ServerKey are stored in Zookeeper. See [RFC 5802](#) for details on SCRAM identity and the individual fields.

The following examples also require a user `admin` for inter-broker communication which can be created using:

```
> bin/kafka-configs.sh --zookeeper localhost:2182 --zk-tls-config-file zk_tls_config.properties --alter --add-config 'SCRAM-SHA-256=[password=admin-secret],SCRAM-SHA-512=[password=admin-secret]' --entity-type users --entity-name admin
```

Existing credentials may be listed using the `--describe` option:

```
> bin/kafka-configs.sh --zookeeper localhost:2182 --zk-tls-config-file zk_tls_config.properties --describe --entity-type users --entity-name alice
```

Credentials may be deleted for one or more SCRAM mechanisms using the `--alter --delete-config` option:

```
> bin/kafka-configs.sh --zookeeper localhost:2182 --zk-tls-config-file zk_tls_config.properties --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name alice
```

## 2. [Configuring Kafka Brokers](#)

1. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it `kafka_server_jaas.conf` for this example:

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
        username="admin"  
        password="admin-secret";  
};
```

The properties

username

and

password

in the

KafkaServer

section are used by the broker to initiate connections to other brokers. In this example,

admin

is the user for inter-broker communication.

2. Pass the JAAS config file location as JVM parameter to each Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. Configure SASL port and SASL mechanisms in server.properties as described

here

. For example:

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-256 (or SCRAM-SHA-512)
sasl.enabled.mechanisms=SCRAM-SHA-256 (or SCRAM-SHA-512)
```

### 3. [Configuring Kafka Clients](#)

To configure SASL authentication on the clients:

1. Configure the JAAS configuration property for each client in producer.properties or consumer.properties. The login module describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client for the SCRAM mechanisms:

```
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required \
    username="alice" \
    password="alice-secret";
```

The options `username` and `password` are used by clients to configure the user for client connections. In this example, clients connect to the broker as user *alice*. Different clients within a JVM may connect as different users by specifying different user names and passwords in `sasl.jaas.config`.

JAAS configuration for clients may alternatively be specified as a JVM parameter similar to brokers as described [here](#). Clients use the login section named `kafkaclient`. This option allows only one user for all client connections from a JVM.

2. Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_SSL  
sasl.mechanism=SCRAM-SHA-256 (or SCRAM-SHA-512)
```

#### 4. [Security Considerations for SASL/SCRAM](#)

- The default implementation of SASL/SCRAM in Kafka stores SCRAM credentials in Zookeeper. This is suitable for production use in installations where Zookeeper is secure and on a private network.
- Kafka supports only the strong hash functions SHA-256 and SHA-512 with a minimum iteration count of 4096. Strong hash functions combined with strong passwords and high iteration counts protect against brute force attacks if Zookeeper security is compromised.
- SCRAM should be used only with TLS-encryption to prevent interception of SCRAM exchanges. This protects against dictionary or brute force attacks and against impersonation if Zookeeper is compromised.
- From Kafka version 2.0 onwards, the default SASL/SCRAM credential store may be overridden using custom callback handlers by configuring `sasl.server.callback.handler.class` in installations where Zookeeper is not secure.
- For more details on security considerations, refer to [RFC 5802](#).

#### 6. [Authentication using SASL/OAUTHBEARER](#)

The [OAuth 2 Authorization Framework](#) "enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf." The SASL OAUTHBEARER mechanism enables the use of the framework in a SASL (i.e. a non-HTTP) context; it is defined in [RFC 7628](#). The default OAUTHBEARER implementation in Kafka creates and validates [Unsecured JSON Web Tokens](#) and is only suitable for use in non-production Kafka installations. Refer to [Security Considerations](#) for more details.

Under the default implementation of

```
principal.builder.class
```

, the principalName of OAuthBearerToken is used as the authenticated

```
Principal
```

for configuration of ACLs etc.

##### 1. [Configuring Kafka Brokers](#)

1. Add a suitably modified JAAS file similar to the one below to each Kafka broker's config directory, let's call it kafka\_server\_jaas.conf for this example:

```
KafkaServer {  
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule  
    required  
    unsecuredLoginStringClaim_sub="admin";  
};
```

The property

```
unsecuredLoginStringClaim_sub
```

in the

## KafkaServer

section is used by the broker when it initiates connections to other brokers. In this example,

admin

will appear in the subject (  
sub  
) claim and will be the user for inter-broker communication.

- Pass the JAAS config file location as JVM parameter to each Kafka broker:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

- Configure SASL port and SASL mechanisms in server.properties as described

here

. For example:

```
listeners=SASL_SSL://host.name:port (or SASL_PLAINTEXT if non-production)
security.inter.broker.protocol=SASL_SSL (or SASL_PLAINTEXT if non-
production)
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
sasl.enabled.mechanisms=OAUTHBEARER
```

## 2. [Configuring Kafka Clients](#)

To configure SASL authentication on the clients:

- Configure the JAAS configuration property for each client in producer.properties or consumer.properties. The login module describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client for the OAUTHBEARER mechanisms:

```
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerL
oginModule required \
unsecuredLoginStringClaim_sub="alice";
```

The option `unsecuredLoginStringClaim_sub` is used by clients to configure the subject (`sub`) claim, which determines the user for client connections. In this example, clients connect to the broker as user `alice`. Different clients within a JVM may connect as different users by specifying different subject (`sub`) claims in `sasl.jaas.config`.

JAAS configuration for clients may alternatively be specified as a JVM parameter similar to brokers as described [here](#). Clients use the login section named `kafkaclient`. This option allows only one user for all client connections from a JVM.

- Configure the following properties in producer.properties or consumer.properties:

```
security.protocol=SASL_SSL (or SASL_PLAINTEXT if non-production)
sasl.mechanism=OAUTHBEARER
```

3. The default implementation of SASL/OAUTHBEARER depends on the jackson-databind library. Since it's an optional dependency, users have to configure it as a dependency via their build tool.

### 3. [Unsecured Token Creation Options for SASL/OAUTHBEARER](#)

- The default implementation of SASL/OAUTHBEARER in Kafka creates and validates [Unsecured JSON Web Tokens](#). While suitable only for non-production use, it does provide the flexibility to create arbitrary tokens in a DEV or TEST environment.
- Here are the various supported JAAS module options on the client side (and on the broker side if OAUTHBEARER is the inter-broker protocol):

JAAS Module Option for Unsecured Token Creation	Documentation
<code>unsecuredLoginStringClaim_&lt;claimname&gt;="value"</code>	Creates a <code>String</code> claim with the given name and value. Any valid claim name can be specified except ' <code>iat</code> ' and ' <code>exp</code> ' (these are automatically generated).
<code>unsecuredLoginNumberClaim_&lt;claimname&gt;="value"</code>	Creates a <code>Number</code> claim with the given name and value. Any valid claim name can be specified except ' <code>iat</code> ' and ' <code>exp</code> ' (these are automatically generated).
<code>unsecuredLoginListClaim_&lt;claimname&gt;="value"</code>	Creates a <code>String List</code> claim with the given name and values parsed from the given value where the first character is taken as the delimiter. For example: <code>unsecuredLoginListClaim_fubar=" value1 value2"</code> . Any valid claim name can be specified except ' <code>iat</code> ' and ' <code>exp</code> ' (these are automatically generated).
<code>unsecuredLoginExtension_&lt;extensionname&gt;="value"</code>	Creates a <code>String</code> extension with the given name and value. For example: <code>unsecuredLoginExtension_traceId="123"</code> . A valid extension name is any sequence of lowercase or uppercase alphabet characters. In addition, the "auth" extension name is reserved. A valid extension value is any combination of characters with ASCII codes 1-127.
<code>unsecuredLoginPrincipalClaimName</code>	Set to a custom claim name if you wish the name of the <code>String</code> claim holding the principal name to be something other than 'sub'.
<code>unsecuredLoginLifetimeSeconds</code>	Set to an integer value if the token expiration is to be set to something other than the default value of 3600 seconds (which is 1 hour). The ' <code>exp</code> ' claim will be set to reflect the expiration time.
<code>unsecuredLoginScopeClaimName</code>	Set to a custom claim name if you wish the name of the <code>String</code> or <code>String List</code> claim holding any token scope to be something other than 'scope'.

### 4. [Unsecured Token Validation Options for SASL/OAUTHBEARER](#)

- Here are the various supported JAAS module options on the broker side for

Unsecured JSON Web Token

validation:

JAAS Module Option for Unsecured Token Validation	Documentation
<code>unsecuredValidatorPrincipalClaimName="value"</code>	Set to a non-empty value if you wish a particular <code>String</code> claim holding a principal name to be checked for existence; the default is to check for the existence of the 'sub' claim.
<code>unsecuredValidatorScopeClaimName="value"</code>	Set to a custom claim name if you wish the name of the <code>String</code> or <code>String List</code> claim holding any token scope to be something other than 'scope'.
<code>unsecuredValidatorRequiredScope="value"</code>	Set to a space-delimited list of scope values if you wish the <code>String/String List</code> claim holding the token scope to be checked to make sure it contains certain values.
<code>unsecuredValidatorAllowableClockSkewMs="value"</code>	Set to a positive integer value if you wish to allow up to some number of positive milliseconds of clock skew (the default is 0).

- The default unsecured SASL/OAUTHBEARER implementation may be overridden (and must be overridden in production environments) using custom login and SASL Server callback handlers.
- For more details on security considerations, refer to [RFC 6749, Section 10](#).

## 5. [Token Refresh for SASL/OAUTHBEARER](#)

Kafka periodically refreshes any token before it expires so that the client can continue to make connections to brokers. The parameters that impact how the refresh algorithm operates are specified as part of the producer/consumer/broker configuration and are as follows. See the documentation for these properties elsewhere for details. The default values are usually reasonable, in which case these configuration parameters would not need to be explicitly set.

Producer/Consumer/Broker Configuration Property
<code>sasl.login.refresh.window.factor</code>
<code>sasl.login.refresh.window.jitter</code>
<code>sasl.login.refresh.min.period.seconds</code>
<code>sasl.login.refresh.min.buffer.seconds</code>

## 6. [Secure/Production Use of SASL/OAUTHBEARER](#)

Production use cases will require writing an implementation of

```
org.apache.kafka.common.security.auth.AuthenticateCallbackHandler
```

that can handle an instance of

`org.apache.kafka.common.security.oauthbearer.OAuthBearerTokenCallback`

and declaring it via either the

`sasl.login.callback.handler.class`

configuration option for a non-broker client or via the

`listener.name.sasl_ssl.oauthbearer.sasl.login.callback.handler.class`

configuration option for brokers (when SASL/OAUTHBEARER is the inter-broker protocol).

Production use cases will also require writing an implementation of

`org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` that can handle an instance of

`org.apache.kafka.common.security.oauthbearer OAuthBearervalidatorCallback` and declaring it via the

`listener.name.sasl_ssl.oauthbearer.sasl.server.callback.handler.class` broker configuration option.

## 7. [Security Considerations for SASL/OAUTHBEARER](#)

- The default implementation of SASL/OAUTHBEARER in Kafka creates and validates [Unsecured JSON Web Tokens](#). This is suitable only for non-production use.
- OAUTHBEARER should be used in production environments only with TLS-encryption to prevent interception of tokens.
- The default unsecured SASL/OAUTHBEARER implementation may be overridden (and must be overridden in production environments) using custom login and SASL Server callback handlers as described above.
- For more details on OAuth 2 security considerations in general, refer to [RFC 6749, Section 10](#).

## 7. [Enabling multiple SASL mechanisms in a broker](#)

1. Specify configuration for the login modules of all enabled mechanisms in the

KafkaServer

section of the JAAS config file. For example:

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_server.keytab"  
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";  
  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
    username="admin"  
    password="admin-secret"  
    user_admin="admin-secret"  
    user_alice="alice-secret";  
};
```

2. Enable the SASL mechanisms in server.properties:

```
sasl.enabled.mechanisms=GSSAPI,PLAIN,SCRAM-SHA-256,SCRAM-SHA-512,OAUTHBEARER
```

3. Specify the SASL security protocol and mechanism for inter-broker communication in server.properties if required:

```
security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
sasl.mechanism.inter.broker.protocol=GSSAPI (or one of the other enabled
mechanisms)
```

4. Follow the mechanism-specific steps in [GSSAPI \(Kerberos\)](#), [PLAIN](#), [SCRAM](#) and [OAUTHBEARER](#) to configure SASL for the enabled mechanisms.

## 8. [Modifying SASL mechanism in a Running Cluster](#)

SASL mechanism can be modified in a running cluster using the following sequence:

1. Enable new SASL mechanism by adding the mechanism to `sasl.enabled.mechanisms` in server.properties for each broker. Update JAAS config file to include both mechanisms as described [here](#). Incrementally bounce the cluster nodes.
2. Restart clients using the new mechanism.
3. To change the mechanism of inter-broker communication (if this is required), set `sasl.mechanism.inter.broker.protocol` in server.properties to the new mechanism and incrementally bounce the cluster again.
4. To remove old mechanism (if this is required), remove the old mechanism from `sasl.enabled.mechanisms` in server.properties and remove the entries for the old mechanism from JAAS config file. Incrementally bounce the cluster again.

## 9. [Authentication using Delegation Tokens](#)

Delegation token based authentication is a lightweight authentication mechanism to complement existing SASL/SSL methods. Delegation tokens are shared secrets between kafka brokers and clients. Delegation tokens will help processing frameworks to distribute the workload to available workers in a secure environment without the added cost of distributing Kerberos TGT/keytabs or keystores when 2-way SSL is used. See [KIP-48](#) for more details.

Under the default implementation of

```
principal.builder.class
```

, the owner of delegation token is used as the authenticated

```
Principal
```

for configuration of ACLs etc.

Typical steps for delegation token usage are:

1. User authenticates with the Kafka cluster via SASL or SSL, and obtains a delegation token. This can be done using Admin APIs or using `kafka-delegation-tokens.sh` script.
2. User securely passes the delegation token to Kafka clients for authenticating with the Kafka cluster.
3. Token owner/renewer can renew/expire the delegation tokens.

## 1. [Token Management](#)

A secret is used to generate and verify delegation tokens. This is supplied using config option `delegation.token.secret.key`. The same secret key must be configured across all the brokers. If the secret is not set or set to empty string, brokers will disable the delegation token authentication.

In the current implementation, token details are stored in Zookeeper and is suitable for use in Kafka installations where Zookeeper is on a private network. Also currently, this secret is stored as plain text in the `server.properties` config file. We intend to make these configurable in a future Kafka release.

A token has a current life, and a maximum renewable life. By default, tokens must be renewed once every 24 hours for up to 7 days. These can be configured using `delegation.token.expiry.time.ms` and `delegation.token.max.lifetime.ms` config options.

Tokens can also be cancelled explicitly. If a token is not renewed by the token's expiration time or if token is beyond the max life time, it will be deleted from all broker caches as well as from zookeeper.

## 2. [Creating Delegation Tokens](#)

Tokens can be created by using Admin APIs or using `kafka-delegation-tokens.sh` script. Delegation token requests (create/renew/expire/describe) should be issued only on SASL or SSL authenticated channels. Tokens can not be requests if the initial authentication is done through delegation token. A token can be created by the user for that user or others as well by specifying the `--owner-principal` parameter. Owner/Renewers can renew or expire tokens. Owner/renewers can always describe their own tokens. To describe other tokens, a `DESCRIBE_TOKEN` permission needs to be added on the User resource representing the owner of the token. `kafka-delegation-tokens.sh` script examples are given below.

Create a delegation token:

```
> bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 --create  
--max-life-time-period -1 --command-config client.properties --renewer-  
principal user:user1
```

Create a delegation token for a different owner:

```
> bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 --create  
--max-life-time-period -1 --command-config client.properties --renewer-  
principal user:user1 --owner-principal user:owner1
```

Renew a delegation token:

```
> bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 --renew  
--renew-time-period -1 --command-config client.properties --hmac ABCDEFGHIJK
```

Expire a delegation token:

```
> bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 --expire  
--expiry-time-period -1 --command-config client.properties --hmac  
ABCDEFGHIJK
```

Existing tokens can be described using the `--describe` option:

```
> bin/kafka-delegation-tokens.sh --bootstrap-server localhost:9092 --describe  
--command-config client.properties --owner-principal User:user1
```

### 3. [Token Authentication](#)

Delegation token authentication piggybacks on the current SASL/SCRAM authentication mechanism. We must enable SASL/SCRAM mechanism on Kafka cluster as described in [here](#).

Configuring Kafka Clients:

1. Configure the JAAS configuration property for each client in producer.properties or consumer.properties. The login module describes how the clients like producer and consumer can connect to the Kafka Broker. The following is an example configuration for a client for the token authentication:

```
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule  
required \  
username="tokenID123" \  
password="1AYYSFmLs4bTjf+1TZ1LCHR/ZZFNA==" \  
tokenauth="true";
```

The options `username` and `password` are used by clients to configure the token id and token HMAC. And the option `tokenauth` is used to indicate the server about token authentication. In this example, clients connect to the broker using token id: `tokenID123`. Different clients within a JVM may connect using different tokens by specifying different token details in `sasl.jaas.config`.

JAAS configuration for clients may alternatively be specified as a JVM parameter similar to brokers as described [here](#). Clients use the login section named `kafkaClient`. This option allows only one user for all client connections from a JVM.

### 4. [Procedure to manually rotate the secret:](#)

We require a re-deployment when the secret needs to be rotated. During this process, already connected clients will continue to work. But any new connection requests and renew/expire requests with old tokens can fail. Steps are given below.

1. Expire all existing tokens.
2. Rotate the secret by rolling upgrade, and
3. Generate new tokens

We intend to automate this in a future Kafka release.

## 7.5 Authorization and ACLs

Kafka ships with a pluggable authorization framework, which is configured with the `authorizer.class.name` property in the server configuration. Configured implementations must extend `org.apache.kafka.server.authorizer.Authorizer`. Kafka provides default implementations which store ACLs in the cluster metadata (either Zookeeper or the KRaft metadata log). For Zookeeper-based clusters, the provided implementation is configured as follows:

```
authorizer.class.name=kafka.security.authorizer.AclAuthorizer
```

For KRaft clusters, use the following configuration on all nodes (brokers, controllers, or combined broker/controller nodes):

```
authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

Kafka ACLs are defined in the general format of "Principal {P} is [Allowed | Denied] Operation {O} From Host {H} on any Resource {R} matching ResourcePattern {RP}" . You can read more about the ACL structure in [KIP-11](#) and resource patterns in [KIP-290](#). In order to add, remove, or list ACLs, you can use the Kafka ACL CLI `kafka-acls.sh`. By default, if no ResourcePatterns match a specific Resource R, then R has no associated ACLs, and therefore no one other than super users is allowed to access R. If you want to change that behavior, you can include the following in `server.properties`.

```
allow.everyone.if.no.acl.found=true
```

One can also add super users in `server.properties` like the following (note that the delimiter is semicolon since SSL user names may contain comma). Default PrincipalType string "User" is case sensitive.

```
super.users=User:Bob;User:Alice
```

## [KRaft Principal Forwarding](#)

In KRaft clusters, admin requests such as `CreateTopics` and `DeleteTopics` are sent to the broker listeners by the client. The broker then forwards the request to the active controller through the first listener configured in `controller.listener.names`. Authorization of these requests is done on the controller node. This is achieved by way of an `Envelope` request which packages both the underlying request from the client as well as the client principal. When the controller receives the forwarded `Envelope` request from the broker, it first authorizes the `Envelope` request using the authenticated broker principal. Then it authorizes the underlying request using the forwarded principal. All of this implies that Kafka must understand how to serialize and deserialize the client principal. The authentication framework allows for customized principals by overriding the `principal.builder.class` configuration. In order for customized principals to work with KRaft, the configured class must implement `org.apache.kafka.common.security.auth.KafkaPrincipalSerde` so that Kafka knows how to serialize and deserialize the principals. The default implementation `org.apache.kafka.common.security.authenticator.DefaultKafkaPrincipalBuilder` uses the Kafka RPC format defined in the source code: `clients/src/main/resources/common/message/DefaultPrincipalData.json`. For more detail about request forwarding in KRaft, see [KIP-590](#)

## [Customizing SSL User Name](#)

By default, the SSL user name will be of the form "CN=writeuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown". One can change that by setting `ssl.principal.mapping.rules` to a customized rule in `server.properties`. This config allows a list of rules for mapping X.500 distinguished name to short name. The rules are evaluated in order and the first rule that matches a distinguished name is used to map it to a short name. Any later rules in the list are ignored.

The format of `ssl.principal.mapping.rules` is a list where each rule starts with "RULE:" and contains an expression as the following formats. Default rule will return string representation of the X.500 certificate distinguished name. If the distinguished name matches the pattern, then the replacement command will be run over the name. This also supports lowercase/uppercase options, to force the translated result to be all lower/uppercase case. This is done by adding a "/L" or "/U" to the end of the rule.

```
RULE:pattern/replacement/
RULE:pattern/replacement/[LU]
```

Example `ssl.principal.mapping.rules` values are:

```
RULE:^CN=(.*?),OU=ServiceUsers.*$/\$1/,  
RULE:^CN=(.*?),OU=(.*?),O=(.*?),L=(.*?),ST=(.*?),C=(.*?)$/\$1@\$2/L,  
RULE:^.*[Cc][Nn]=([a-zA-Z0-9.]*).*/\$1/L,  
DEFAULT
```

Above rules translate distinguished name

"CN=serviceuser,OU=ServiceUsers,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" to "serviceuser"  
and "CN=adminUser,OU=Admin,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" to  
"adminuser@admin".

For advanced use cases, one can customize the name by setting a customized PrincipalBuilder in server.properties like the following.

```
principal.builder.class=CustomizedPrincipalBuilderClass
```

### [Customizing SASL User Name](#)

By default, the SASL user name will be the primary part of the Kerberos principal. One can change that by setting `sasl.kerberos.principal.to.local.rules` to a customized rule in server.properties. The format of `sasl.kerberos.principal.to.local.rules` is a list where each rule works in the same way as the auth\_to\_local in [Kerberos configuration file \(krb5.conf\)](#). This also support additional lowercase/uppercase rule, to force the translated result to be all lowercase/uppercase. This is done by adding a "/L" or "/U" to the end of the rule. check below formats for syntax. Each rules starts with RULE: and contains an expression as the following formats. See the kerberos documentation for more details.

```
RULE:[n:string](regexp)s/pattern/replacement/  
RULE:[n:string](regexp)s/pattern/replacement/g  
RULE:[n:string](regexp)s/pattern/replacement//L  
RULE:[n:string](regexp)s/pattern/replacement/g/L  
RULE:[n:string](regexp)s/pattern/replacement//U  
RULE:[n:string](regexp)s/pattern/replacement/g/U
```

An example of adding a rule to properly translate [user@MYDOMAIN.COM](#) to user while also keeping the default rule in place is:

```
sasl.kerberos.principal.to.local.rules=RULE:[1:$1@$0](.*@MYDOMAIN.COM)s/@.*/,,DEFAULT
```

### [Command Line Interface](#)

Kafka Authorization management CLI can be found under bin directory with all the other CLIs. The CLI script is called **kafka-acls.sh**. Following lists all the options that the script supports:

OPTION	DESCRIPTION	DEFAULT	OPTION TYPE
-add	Indicates to the script that user is trying to add an acl.		Action
-remove	Indicates to the script that user is trying to remove an acl.		Action
-list	Indicates to the script that user is trying to list acls.		Action
--bootstrap-server	A list of host/port pairs to use for establishing the connection to the Kafka cluster. Only one of --bootstrap-server or --authorizer option must be specified.		Configuration
--command-config	A property file containing configs to be passed to Admin Client. This option can only be used with --bootstrap-server option.		Configuration
--cluster	Indicates to the script that the user is trying to interact with acls on the singular cluster resource.		ResourcePattern
--topic [topic-name]	Indicates to the script that the user is trying to interact with acls on topic resource pattern(s).		ResourcePattern
--group [group-name]	Indicates to the script that the user is trying to interact with acls on consumer-group resource pattern(s)		ResourcePattern
--transactional-id [transactional-id]	The transactionalid to which ACLs should be added or removed. A value of * indicates the ACLs should apply to all transactionalids.		ResourcePattern
--delegation-token [delegation-token]	Delegation token to which ACLs should be added or removed. A value of * indicates ACL should apply to all tokens.		ResourcePattern
--user-principal [user-principal]	A user resource to which ACLs should be added or removed. This is currently supported in relation with delegation tokens. A value of * indicates ACL should apply to all users.		ResourcePattern
--resource-pattern-type [pattern-type]	Indicates to the script the type of resource pattern, (for --add), or resource pattern filter, (for --list and --remove), the user wishes to use. When adding acls, this should be a specific pattern type, e.g. 'literal' or 'prefixed'. When listing or removing acls, a specific pattern type filter can be used to list or remove acls from a specific type of resource pattern, or the filter values of 'any' or 'match' can be used, where 'any' will match any pattern type, but will match the resource name exactly, and 'match' will perform pattern matching to list or remove all acls that affect the supplied resource(s). WARNING: 'match', when used in combination with the '--remove' switch, should be used with care.	literal	Configuration
--allow-principal	Principal is in PrincipalType:name format that will be added to ACL with Allow permission. Default PrincipalType string "User" is case sensitive. You can specify multiple --allow-principal in a single command.		Principal
--deny-principal	Principal is in PrincipalType:name format that will be added to ACL with Deny permission. Default PrincipalType string "User" is case sensitive. You can specify multiple --deny-principal in a single command.		Principal
--principal	Principal is in PrincipalType:name format that will be used along with --list option. Default PrincipalType string "User" is case sensitive. This will list the ACLs for the specified principal. You can specify multiple --principal in a single command.		Principal
--allow-host	IP address from which principals listed in --allow-principal will have access.	if --allow-principal is specified defaults to * which translates to "all hosts"	Host
--deny-host	IP address from which principals listed in --deny-principal will be denied access.	if --deny-principal is specified defaults to * which translates to "all hosts"	Host
--operation	Operation that will be allowed or denied. Valid values are:ReadWriteCreateDeleteAlterDescribeClusterActionDescribeConfigsAlterConfigsIdempotentWriteCreateTokensDescribeTokensAll	All	Operation
--producer	Convenience option to add/remove acls for producer role. This will generate acls that allows WRITE, DESCRIBE and CREATE on topic.		Convenience
--consumer	Convenience option to add/remove acls for consumer role. This will generate acls that allows READ, DESCRIBE on topic and READ on consumer-group.		Convenience
--idempotent	Enable idempotence for the producer. This should be used in combination with the --producer option. Note that idempotence is enabled automatically if the producer is authorized to a particular transactional-id.		Convenience
--force	Convenience option to assume yes to all queries and do not prompt.		Convenience
--authorizer	(DEPRECATED: not supported in KRaft) Fully qualified class name of the authorizer.	kafka.security.authorizer.AclAuthorizer	Configuration
--authorizer-properties	(DEPRECATED: not supported in KRaft) key=val pairs that will be passed to authorizer for initialization. For the default authorizer in ZK clusters, the example values are: zookeeper.connect=localhost:2181		Configuration
--zk-tls-config-file	(DEPRECATED: not supported in KRaft) Identifies the file where ZooKeeper client TLS connectivity properties for the authorizer are defined. Any properties other than the following (with or without an "authorizer." prefix) are ignored: zookeeper.clientCnxnSocket, zookeeper.ssl.cipher.suites, zookeeper.ssl.client.enable, zookeeper.ssl.crl.enable, zookeeper.ssl.enabled.protocols, zookeeper.ssl.endpoint.identification.algorithm, zookeeper.ssl.keystore.location, zookeeper.ssl.keystore.password, zookeeper.ssl.keystore.type, zookeeper.ssl.ocsp.enable, zookeeper.ssl.protocol, zookeeper.ssl.truststore.location, zookeeper.ssl.truststore.password, zookeeper.ssl.truststore.type		Configuration

## Examples

- Adding Acls

Suppose you want to add an acl "Principals User:Bob and User:Alice are allowed to perform Operation Read and Write on Topic Test-Topic from IP 198.51.100.0 and IP 198.51.100.1". You can do that by executing the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100.0 --allow-host 198.51.100.1 --operation Read --operation write --topic Test-topic
```

By default, all principals that don't have an explicit acl that allows access for an operation to a resource are denied. In rare cases where an allow acl is defined that allows access to all but some principal we will have to use the --deny-principal and --deny-host option. For example, if we want to allow all users to Read from Test-topic but only deny User:BadBob from IP 198.51.100.3 we can do so using following commands:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal User:'*' --allow-host '*' --deny-principal User:BadBob --deny-host 198.51.100.3 --operation Read --topic Test-topic
```

Note that

```
--allow-host
```

and

```
--deny-host
```

only support IP addresses (hostnames are not supported). Above examples add acls to a topic by specifying --topic [topic-name] as the resource pattern option. Similarly user can add acls to cluster by specifying --cluster and to a consumer group by specifying --group [group-name]. You can add acls on any resource of a certain type, e.g. suppose you wanted to add an acl "Principal User:Peter is allowed to produce to any Topic from IP 198.51.200.0" You can do that by using the wildcard resource '\*', e.g. by executing the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal User:Peter --allow-host 198.51.200.1 --producer --topic '*'
```

You can add acls on prefixed resource patterns, e.g. suppose you want to add an acl "Principal User:Jane is allowed to produce to any Topic whose name starts with 'Test-' from any host". You can do that by executing the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal User:Jane --producer --topic Test- --resource-pattern-type prefixed
```

Note, --resource-pattern-type defaults to 'literal', which only affects resources with the exact same name or, in the case of the wildcard resource name '\*', a resource with any name.

- Removing Acls

Removing acls is pretty much the same. The only difference is instead of --add option users will have to specify --remove option. To remove the acls added by the first example above we can execute the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --allow-principal User:Bob --allow-principal User:Alice --allow-host 198.51.100.0 --allow-host 198.51.100.1 --operation Read --operation Write --topic Test-topic
```

If you want to remove the acl added to the prefixed resource pattern above we can execute the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --allow-principal User:Jane --producer --topic Test- --resource-pattern-type Prefixed
```

- List Acls

We can list acls for any resource by specifying the --list option with the resource. To list all acls on the literal resource pattern Test-topic, we can execute the CLI with following options:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --list --topic Test-topic
```

However, this will only return the acls that have been added to this exact resource pattern. Other acls can exist that affect access to the topic, e.g. any acls on the topic wildcard '\*', or any acls on prefixed resource patterns. Acls on the wildcard resource pattern can be queried explicitly:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --list --topic '*'
```

However, it is not necessarily possible to explicitly query for acls on prefixed resource patterns that match Test-topic as the name of such patterns may not be known. We can list

all

acls affecting Test-topic by using '--resource-pattern-type match', e.g.

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --list --topic Test-topic --  
resource-pattern-type match
```

This will list acls on all matching literal, wildcard and prefixed resource patterns.

- Adding or removing a principal as producer or consumer

The most common use case for acl management are adding/removing a principal as producer or consumer so we added convenience options to handle these cases. In order to add User:Bob as a producer of Test-topic we can execute the following command:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal  
User:Bob --producer --topic Test-topic
```

Similarly to add Alice as a consumer of Test-topic with consumer group Group-1 we just have to pass --consumer option:

```
> bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --allow-principal  
User:Bob --consumer --topic Test-topic --group Group-1
```

Note that for consumer option we must also specify the consumer group. In order to remove a principal from producer or consumer role we just need to pass --remove option.

- Admin API based acl management

Users having Alter permission on ClusterResource can use Admin API for ACL management. kafka-acls.sh script supports AdminClient API to manage ACLs without interacting with zookeeper/authorizer directly. All the above examples can be executed by using

--bootstrap-server

option. For example:

```

bin/kafka-acls.sh --bootstrap-server localhost:9092 --command-config
/tmp/adminclient-configs.conf --add --allow-principal User:Bob --producer --topic
Test-topic
bin/kafka-acls.sh --bootstrap-server localhost:9092 --command-config
/tmp/adminclient-configs.conf --add --allow-principal User:Bob --consumer --topic
Test-topic --group Group-1
bin/kafka-acls.sh --bootstrap-server localhost:9092 --command-config
/tmp/adminclient-configs.conf --list --topic Test-topic
bin/kafka-acls.sh --bootstrap-server localhost:9092 --command-config
/tmp/adminclient-configs.conf --add --allow-principal User:tokenRequester --
operation CreateTokens --user-principal "owner1"

```

## [Authorization Primitives](#)

Protocol calls are usually performing some operations on certain resources in Kafka. It is required to know the operations and resources to set up effective protection. In this section we'll list these operations and resources, then list the combination of these with the protocols to see the valid scenarios.

### [Operations in Kafka](#)

There are a few operation primitives that can be used to build up privileges. These can be matched up with certain resources to allow specific protocol calls for a given user. These are:

- Read
- Write
- Create
- Delete
- Alter
- Describe
- ClusterAction
- DescribeConfigs
- AlterConfigs
- IdempotentWrite
- CreateTokens
- DescribeTokens
- All

### [Resources in Kafka](#)

The operations above can be applied on certain resources which are described below.

- **Topic:** this simply represents a Topic. All protocol calls that are acting on topics (such as reading, writing them) require the corresponding privilege to be added. If there is an authorization error with a topic resource, then a TOPIC\_AUTHORIZATION\_FAILED (error code: 29) will be returned.
- **Group:** this represents the consumer groups in the brokers. All protocol calls that are working with consumer groups, like joining a group must have privileges with the group in subject. If the privilege is not given then a GROUP\_AUTHORIZATION\_FAILED (error code: 30) will be returned in the protocol response.
- **Cluster:** this resource represents the cluster. Operations that are affecting the whole cluster, like controlled shutdown are protected by privileges on the Cluster resource. If there is an authorization problem on a cluster resource, then a CLUSTER\_AUTHORIZATION\_FAILED (error code: 31) will be returned.
- **TransactionalId:** this resource represents actions related to transactions, such as committing. If any error occurs, then a TRANSACTIONAL\_ID\_AUTHORIZATION\_FAILED (error code: 53) will be returned by brokers.
- **DelegationToken:** this represents the delegation tokens in the cluster. Actions, such as describing delegation tokens could be protected by a privilege on the DelegationToken resource. Since these

objects have a little special behavior in Kafka it is recommended to read [KIP-48](#) and the related upstream documentation at [Authentication using Delegation Tokens](#).

- **User:** CreateToken and DescribeToken operations can be granted to User resources to allow creating and describing tokens for other users. More info can be found in [KIP-373](#).

### [Operations and Resources on Protocols](#)

In the below table we'll list the valid operations on resources that are executed by the Kafka API protocols.

PROTOCOL (API KEY)	OPERATION	RESOURCE	NOTE
PRODUCE (0)	Write	TransactionalId	An transactional producer which has its transactional.id set requires this privilege.
PRODUCE (0)	IdempotentWrite	Cluster	An idempotent produce action requires this privilege.
PRODUCE (0)	Write	Topic	This applies to a normal produce action.
FETCH (1)	ClusterAction	Cluster	A follower must have ClusterAction on the Cluster resource in order to fetch partition data.
FETCH (1)	Read	Topic	Regular Kafka consumers need READ permission on each partition they are fetching.
LIST_OFFSETS (2)	Describe	Topic	
METADATA (3)	Describe	Topic	
METADATA (3)	Create	Cluster	If topic auto-creation is enabled, then the broker-side API will check for the existence of a Cluster level privilege. If it's found then it'll allow creating the topic, otherwise it'll iterate through the Topic level privileges (see the next one).
METADATA (3)	Create	Topic	This authorizes auto topic creation if enabled but the given user doesn't have a cluster level permission (above).
LEADER_AND_ISR (4)	ClusterAction	Cluster	
STOP_REPLICA (5)	ClusterAction	Cluster	
UPDATE_METADATA (6)	ClusterAction	Cluster	
CONTROLLED_SHUTDOWN (7)	ClusterAction	Cluster	
OFFSET_COMMIT (8)	Read	Group	An offset can only be committed if it's authorized to the given group and the topic too (see below). Group access is checked first, then Topic access.
OFFSET_COMMIT (8)	Read	Topic	Since offset commit is part of the consuming process, it needs privileges for the read action.
OFFSET_FETCH (9)	Describe	Group	Similarly to OFFSET_COMMIT, the application must have privileges on group and topic level too to be able to fetch. However in this case it requires describe access instead of read. Group access is checked first, then Topic access.
OFFSET_FETCH (9)	Describe	Topic	
FIND_COORDINATOR (10)	Describe	Group	The FIND_COORDINATOR request can be of "Group" type in which case it is looking for consumergroup coordinators. This privilege would represent the Group mode.
FIND_COORDINATOR (10)	Describe	TransactionalId	This applies only on transactional producers and checked when a producer tries to find the transaction coordinator.
JOIN_GROUP (11)	Read	Group	
HEARTBEAT (12)	Read	Group	
LEAVE_GROUP (13)	Read	Group	
SYNC_GROUP (14)	Read	Group	
DESCRIBE_GROUPS (15)	Describe	Group	
LIST_GROUPS (16)	Describe	Cluster	When the broker checks to authorize a list_groups request it first checks for this cluster level authorization. If none found then it proceeds to check the groups individually. This operation doesn't return CLUSTER_AUTHORIZATION_FAILED.
LIST_GROUPS (16)	Describe	Group	If none of the groups are authorized, then just an empty response will be sent back instead of an error. This operation doesn't return CLUSTER_AUTHORIZATION_FAILED. This is applicable from the 2.1 release.

PROTOCOL (API KEY)	OPERATION	RESOURCE	NOTE
SASL_HANDSHAKE (17)			The SASL handshake is part of the authentication process and therefore it's not possible to apply any kind of authorization here.
API_VERSIONS (18)			The API_VERSIONS request is part of the Kafka protocol handshake and happens on connection and before any authentication. Therefore it's not possible to control this with authorization.
CREATE_TOPICS (19)	Create	Cluster	If there is no cluster level authorization then it won't return CLUSTER_AUTHORIZATION_FAILED but fall back to use topic level, which is just below. That'll throw error if there is a problem.
CREATE_TOPICS (19)	Create	Topic	This is applicable from the 2.0 release.
DELETE_TOPICS (20)	Delete	Topic	
DELETE_RECORDS (21)	Delete	Topic	
INIT_PRODUCER_ID (22)	Write	TransactionalId	
INIT_PRODUCER_ID (22)	IdempotentWrite	Cluster	
OFFSET_FOR_LEADER_EPOCH (23)	ClusterAction	Cluster	If there is no cluster level privilege for this operation, then it'll check for topic level one.
OFFSET_FOR_LEADER_EPOCH (23)	Describe	Topic	This is applicable from the 2.1 release.
ADD_PARTITIONS_TO_TXN (24)	Write	TransactionalId	This API is only applicable to transactional requests. It first checks for the Write action on the TransactionalId resource, then it checks the Topic in subject (below).
ADD_PARTITIONS_TO_TXN (24)	Write	Topic	
ADD_OFFSETS_TO_TXN (25)	Write	TransactionalId	Similarly to ADD_PARTITIONS_TO_TXN this is only applicable to transactional request. It first checks for Write action on the TransactionalId resource, then it checks whether it can Read on the given group (below).
ADD_OFFSETS_TO_TXN (25)	Read	Group	
END_TXN (26)	Write	TransactionalId	
WRITE_TXN_MARKERS (27)	ClusterAction	Cluster	
TXN_OFFSET_COMMIT (28)	Write	TransactionalId	
TXN_OFFSET_COMMIT (28)	Read	Group	
TXN_OFFSET_COMMIT (28)	Read	Topic	
DESCRIBE_ACLS (29)	Describe	Cluster	
CREATE_ACLS (30)	Alter	Cluster	
DELETE_ACLS (31)	Alter	Cluster	
DESCRIBE_CONFIGS (32)	DescribeConfigs	Cluster	If broker configs are requested, then the broker will check cluster level privileges.
DESCRIBE_CONFIGS (32)	DescribeConfigs	Topic	If topic configs are requested, then the broker will check topic level privileges.
ALTER_CONFIGS (33)	AlterConfigs	Cluster	If broker configs are altered, then the broker will check cluster level privileges.
ALTER_CONFIGS (33)	AlterConfigs	Topic	If topic configs are altered, then the broker will check topic level privileges.
ALTER_REPLICA_LOG_DIRS (34)	Alter	Cluster	
DESCRIBE_LOG_DIRS (35)	Describe	Cluster	An empty response will be returned on authorization failure.
SASL_AUTHENTICATE (36)			SASL_AUTHENTICATE is part of the authentication process and therefore it's not possible to apply any kind of authorization here.
CREATE_PARTITIONS (37)	Alter	Topic	

PROTOCOL (API KEY)	OPERATION	RESOURCE	NOTE
CREATE_DELEGATION_TOKEN (38)			Creating delegation tokens has special rules, for this please see the <a href="#">Authentication using Delegation Tokens</a> section.
CREATE_DELEGATION_TOKEN (38)	CreateTokens	User	Allows creating delegation tokens for the User resource.
RENEW_DELEGATION_TOKEN (39)			Renewing delegation tokens has special rules, for this please see the <a href="#">Authentication using Delegation Tokens</a> section.
EXPIRE_DELEGATION_TOKEN (40)			Expiring delegation tokens has special rules, for this please see the <a href="#">Authentication using Delegation Tokens</a> section.
DESCRIBE_DELEGATION_TOKEN (41)	Describe	DelegationToken	Describing delegation tokens has special rules, for this please see the <a href="#">Authentication using Delegation Tokens</a> section.
DESCRIBE_DELEGATION_TOKEN (41)	DescribeTokens	User	Allows describing delegation tokens of the User resource.
DELETE_GROUPS (42)	Delete	Group	
ELECT_PREFERRED_LEADERS (43)	ClusterAction	Cluster	
INCREMENTAL_ALTER_CONFIGS (44)	AlterConfigs	Cluster	If broker configs are altered, then the broker will check cluster level privileges.
INCREMENTAL_ALTER_CONFIGS (44)	AlterConfigs	Topic	If topic configs are altered, then the broker will check topic level privileges.
ALTER_PARTITION_REASSIGNMENTS (45)	Alter	Cluster	
LIST_PARTITION_REASSIGNMENTS (46)	Describe	Cluster	
OFFSET_DELETE (47)	Delete	Group	
OFFSET_DELETE (47)	Read	Topic	

## 7.6 Incorporating Security Features in a Running Cluster

You can secure a running cluster via one or more of the supported protocols discussed previously. This is done in phases:

- Incrementally bounce the cluster nodes to open additional secured port(s).
- Restart clients using the secured rather than PLAINTEXT port (assuming you are securing the client-broker connection).
- Incrementally bounce the cluster again to enable broker-to-broker security (if this is required)
- A final incremental bounce to close the PLAINTEXT port.

The specific steps for configuring SSL and SASL are described in sections [7.3](#) and [7.4](#). Follow these steps to enable security for your desired protocol(s).

The security implementation lets you configure different protocols for both broker-client and broker-broker communication. These must be enabled in separate bounces. A PLAINTEXT port must be left open throughout so brokers and/or clients can continue to communicate.

When performing an incremental bounce stop the brokers cleanly via a SIGTERM. It's also good practice to wait for restarted replicas to return to the ISR list before moving onto the next node.

As an example, say we wish to encrypt both broker-client and broker-broker communication with SSL. In the first incremental bounce, an SSL port is opened on each node:

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
```

We then restart the clients, changing their config to point at the newly opened, secured port:

```
bootstrap.servers = [broker1:9092,...]
security.protocol = SSL
...etc
```

In the second incremental server bounce we instruct Kafka to use SSL as the broker-broker protocol (which will use the same SSL port):

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092
security.inter.broker.protocol=SSL
```

In the final bounce we secure the cluster by closing the PLAINTEXT port:

```
listeners=SSL://broker1:9092
security.inter.broker.protocol=SSL
```

Alternatively we might choose to open multiple ports so that different protocols can be used for broker-broker and broker-client communication. Say we wished to use SSL encryption throughout (i.e. for broker-broker and broker-client communication) but we'd like to add SASL authentication to the broker-client connection also. We would achieve this by opening two additional ports during the first bounce:

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://broker1:9093
```

We would then restart the clients, changing their config to point at the newly opened, SASL & SSL secured port:

```
bootstrap.servers = [broker1:9093,...]
security.protocol = SASL_SSL
...etc
```

The second server bounce would switch the cluster to use encrypted broker-broker communication via the SSL port we previously opened on port 9092:

```
listeners=PLAINTEXT://broker1:9091,SSL://broker1:9092,SASL_SSL://broker1:9093
security.inter.broker.protocol=SSL
```

The final bounce secures the cluster by closing the PLAINTEXT port.

```
listeners=SSL://broker1:9092,SASL_SSL://broker1:9093
security.inter.broker.protocol=SSL
```

ZooKeeper can be secured independently of the Kafka cluster. The steps for doing this are covered in section [7.7.2](#).

## [7.7 ZooKeeper Authentication](#)

ZooKeeper supports mutual TLS (mTLS) authentication beginning with the 3.5.x versions. Kafka supports authenticating to ZooKeeper with SASL and mTLS -- either individually or both together -- beginning with version 2.5. See [KIP-515: Enable ZK client to use the new TLS supported authentication](#) for more details.

When using mTLS alone, every broker and any CLI tools (such as the [ZooKeeper Security Migration Tool](#)) should identify itself with the same Distinguished Name (DN) because it is the DN that is ACL'ed. This can be changed as described below, but it involves writing and deploying a custom ZooKeeper authentication provider. Generally each certificate should have the same DN but a different Subject Alternative Name (SAN) so that hostname verification of the brokers and any CLI tools by ZooKeeper will succeed.

When using SASL authentication to ZooKeeper together with mTLS, both the SASL identity and either the DN that created the znode (i.e. the creating broker's certificate) or the DN of the Security Migration Tool (if migration was performed after the znode was created) will be ACL'ed, and all brokers and CLI tools will be authorized even if they all use different DNs because they will all use the same ACL'ed SASL identity. It is only when using mTLS authentication alone that all the DNs must match (and SANs become critical -- again, in the absence of writing and deploying a custom ZooKeeper authentication provider as described below).

Use the broker properties file to set TLS configs for brokers as described below.

Use the `--zk-tls-config-file <file>` option to set TLS configs in the Zookeeper Security Migration Tool. The `kafka-acls.sh` and `kafka-configs.sh` CLI tools also support the `--zk-tls-config-file <file>` option.

Use the `-zk-tls-config-file <file>` option (note the single-dash rather than double-dash) to set TLS configs for the `zookeeper-shell.sh` CLI tool.

### [7.7.1 New clusters](#)

#### [7.7.1.1 ZooKeeper SASL Authentication](#)

To enable ZooKeeper SASL authentication on brokers, there are two necessary steps:

1. Create a JAAS login file and set the appropriate system property to point to it as described above
2. Set the configuration property `zookeeper.set.ac1` in each broker to true

The metadata stored in ZooKeeper for the Kafka cluster is world-readable, but can only be modified by the brokers. The rationale behind this decision is that the data stored in ZooKeeper is not sensitive, but inappropriate manipulation of that data can cause cluster disruption. We also recommend limiting the access to ZooKeeper via network segmentation (only brokers and some admin tools need access to ZooKeeper).

#### [7.7.1.2 ZooKeeper Mutual TLS Authentication](#)

ZooKeeper mTLS authentication can be enabled with or without SASL authentication. As mentioned above, when using mTLS alone, every broker and any CLI tools (such as the [ZooKeeper Security Migration Tool](#)) must generally identify itself with the same Distinguished Name (DN) because it is the DN that is ACL'ed, which means each certificate should have an appropriate Subject Alternative Name (SAN) so that hostname verification of the brokers and any CLI tool by ZooKeeper will succeed.

It is possible to use something other than the DN for the identity of mTLS clients by writing a class that extends `org.apache.zookeeper.server.auth.X509AuthenticationProvider` and overrides the method `protected String getClientId(X509Certificate clientCert)`. Choose a scheme name and set `authProvider.[scheme]` in ZooKeeper to be the fully-qualified class name of the custom implementation; then set `ssl.authProvider=[scheme]` to use it.

Here is a sample (partial) ZooKeeper configuration for enabling TLS authentication. These configurations are described in the [ZooKeeper Admin Guide](#).

```
secureClientPort=2182
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider
ssl.keyStore.location=/path/to/zk/keystore.jks
ssl.keyStore.password=zk-ks-passwd
ssl.trustStore.location=/path/to/zk/truststore.jks
ssl.trustStore.password=zk-ts-passwd
```

**IMPORTANT:** ZooKeeper does not support setting the key password in the ZooKeeper server keystore to a value different from the keystore password itself. Be sure to set the key password to be the same as the keystore password.

Here is a sample (partial) Kafka Broker configuration for connecting to ZooKeeper with mTLS authentication. These configurations are described above in [Broker Configs](#).

```
# connect to the ZooKeeper port configured for TLS
zookeeper.connect=zk1:2182,zk2:2182,zk3:2182
# required to use TLS to ZooKeeper (default is false)
zookeeper.ssl.client.enable=true
# required to use TLS to ZooKeeper
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
# define key/trust stores to use TLS to ZooKeeper; ignored unless
zookeeper.ssl.client.enable=true
zookeeper.ssl.keystore.location=/path/to/kafka/keystore.jks
zookeeper.ssl.keystore.password=kafka-ks-passwd
zookeeper.ssl.truststore.location=/path/to/kafka/truststore.jks
zookeeper.ssl.truststore.password=kafka-ts-passwd
# tell broker to create ACLs on znodes
zookeeper.set.acl=true
```

**IMPORTANT:** ZooKeeper does not support setting the key password in the ZooKeeper client (i.e. broker) keystore to a value different from the keystore password itself. Be sure to set the key password to be the same as the keystore password.

## [7.7.2 Migrating clusters](#)

If you are running a version of Kafka that does not support security or simply with security disabled, and you want to make the cluster secure, then you need to execute the following steps to enable ZooKeeper authentication with minimal disruption to your operations:

1. Enable SASL and/or mTLS authentication on ZooKeeper. If enabling mTLS, you would now have both a non-TLS port and a TLS port, like this:

```
clientPort=2181
secureClientPort=2182
serverCnxnFactory=org.apache.zookeeper.server.NettyServerCnxnFactory
authProvider.x509=org.apache.zookeeper.server.auth.X509AuthenticationProvider
ssl.keyStore.location=/path/to/zk/keystore.jks
ssl.keyStore.password=zk-ks-passwd
ssl.trustStore.location=/path/to/zk/truststore.jks
ssl.trustStore.password=zk-ts-passwd
```

2. Perform a rolling restart of brokers setting the JAAS login file and/or defining ZooKeeper mutual TLS configurations (including connecting to the TLS-enabled ZooKeeper port) as required, which enables brokers to authenticate to ZooKeeper. At the end of the rolling restart, brokers are able to manipulate znodes with strict ACLs, but they will not create znodes with those ACLs
3. If you enabled mTLS, disable the non-TLS port in ZooKeeper

4. Perform a second rolling restart of brokers, this time setting the configuration parameter `zookeeper.set.acl` to true, which enables the use of secure ACLs when creating znodes
5. Execute the ZkSecurityMigrator tool. To execute the tool, there is this script: `bin/zookeeper-security-migration.sh` with `zookeeper.acl` set to secure. This tool traverses the corresponding sub-trees changing the ACLs of the znodes. Use the `--zk-tls-config-file <file>` option if you enable mTLS.

It is also possible to turn off authentication in a secure cluster. To do it, follow these steps:

1. Perform a rolling restart of brokers setting the JAAS login file and/or defining ZooKeeper mutual TLS configurations, which enables brokers to authenticate, but setting `zookeeper.set.acl` to false. At the end of the rolling restart, brokers stop creating znodes with secure ACLs, but are still able to authenticate and manipulate all znodes
2. Execute the ZkSecurityMigrator tool. To execute the tool, run this script `bin/zookeeper-security-migration.sh` with `zookeeper.acl` set to unsecure. This tool traverses the corresponding sub-trees changing the ACLs of the znodes. Use the `--zk-tls-config-file <file>` option if you need to set TLS configuration.
3. If you are disabling mTLS, enable the non-TLS port in ZooKeeper
4. Perform a second rolling restart of brokers, this time omitting the system property that sets the JAAS login file and/or removing ZooKeeper mutual TLS configuration (including connecting to the non-TLS-enabled ZooKeeper port) as required
5. If you are disabling mTLS, disable the TLS port in ZooKeeper

Here is an example of how to run the migration tool:

```
> bin/zookeeper-security-migration.sh --zookeeper.acl=secure --
zookeeper.connect=localhost:2181
```

Run this to see the full list of parameters:

```
> bin/zookeeper-security-migration.sh --help
```

### [7.7.3 Migrating the ZooKeeper ensemble](#)

It is also necessary to enable SASL and/or mTLS authentication on the ZooKeeper ensemble. To do it, we need to perform a rolling restart of the server and set a few properties. See above for mTLS information. Please refer to the ZooKeeper documentation for more detail:

1. [Apache ZooKeeper documentation](#)
2. [Apache ZooKeeper wiki](#)

### [7.7.4 ZooKeeper Quorum Mutual TLS Authentication](#)

It is possible to enable mTLS authentication between the ZooKeeper servers themselves. Please refer to the [ZooKeeper documentation](#) for more detail.

## [7.8 ZooKeeper Encryption](#)

ZooKeeper connections that use mutual TLS are encrypted. Beginning with ZooKeeper version 3.5.7 (the version shipped with Kafka version 2.5) ZooKeeper supports a sever-side config `ssl.clientAuth` (case-insensitively: `want` / `need` / `none` are the valid options, the default is `need`), and setting this value to `none` in ZooKeeper allows clients to connect via a TLS-encrypted connection without presenting their own certificate. Here is a sample (partial) Kafka Broker configuration for connecting to ZooKeeper with just TLS encryption. These configurations are described above in [Broker Configs](#).

```
# connect to the ZooKeeper port configured for TLS
zookeeper.connect=zk1:2182,zk2:2182,zk3:2182
# required to use TLS to ZooKeeper (default is false)
zookeeper.ssl.client.enable=true
# required to use TLS to ZooKeeper
zookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty
# define trust stores to use TLS to ZooKeeper; ignored unless
zookeeper.ssl.client.enable=true
# no need to set keystore information assuming ssl.clientAuth=none on ZooKeeper
zookeeper.ssl.truststore.location=/path/to/kafka/truststore.jks
zookeeper.ssl.truststore.password=kafka-ts-passwd
# tell broker to create ACLs on znodes (if using SASL authentication, otherwise do not
set this)
zookeeper.set.acl=true
```

## 8. KAFKA CONNECT

### 8.1 Overview

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It makes it simple to quickly define *connectors* that move large collections of data into and out of Kafka. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency. An export job can deliver data from Kafka topics into secondary storage and query systems or into batch systems for offline analysis.

Kafka Connect features include:

- **A common framework for Kafka connectors** - Kafka Connect standardizes integration of other data systems with Kafka, simplifying connector development, deployment, and management
- **Distributed and standalone modes** - scale up to a large, centrally managed service supporting an entire organization or scale down to development, testing, and small production deployments
- **REST interface** - submit and manage connectors to your Kafka Connect cluster via an easy to use REST API
- **Automatic offset management** - with just a little information from connectors, Kafka Connect can manage the offset commit process automatically so connector developers do not need to worry about this error prone part of connector development
- **Distributed and scalable by default** - Kafka Connect builds on the existing group management protocol. More workers can be added to scale up a Kafka Connect cluster.
- **Streaming/batch integration** - leveraging Kafka's existing capabilities, Kafka Connect is an ideal solution for bridging streaming and batch data systems

### 8.2 User Guide

The [quickstart](#) provides a brief example of how to run a standalone version of Kafka Connect. This section describes how to configure, run, and manage Kafka Connect in more detail.

#### Running Kafka Connect

Kafka Connect currently supports two modes of execution: standalone (single process) and distributed.

In standalone mode all work is performed in a single process. This configuration is simpler to setup and get started with and may be useful in situations where only one worker makes sense (e.g. collecting log files), but it does not benefit from some of the features of Kafka Connect such as fault tolerance. You can start a standalone process with the following command:

```
> bin/connect-standalone.sh config/connect-standalone.properties
[connector1.properties connector2.properties ...]
```

The first parameter is the configuration for the worker. This includes settings such as the Kafka connection parameters, serialization format, and how frequently to commit offsets. The provided example should work well with a local cluster running with the default configuration provided by `config/server.properties`. It will require tweaking to use with a different configuration or production deployment. All workers (both standalone and distributed) require a few configs:

- `bootstrap.servers` - List of Kafka servers used to bootstrap connections to Kafka
- `key.converter` - Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.
- `value.converter` - Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values in messages written to or read from Kafka, and since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON and Avro.
- `plugin.path` (default `empty`) - a list of paths that contain Connect plugins (connectors, converters, transformations). Before running quick starts, users must add the absolute path that contains the example `FileStreamSourceConnector` and `FileStreamSinkConnector` packaged in `connect-file-version.jar`, because these connectors are not included by default to the `CLASSPATH` or the `plugin.path` of the Connect worker (see [plugin.path](#) property for examples).

The important configuration options specific to standalone mode are:

- `offset.storage.file.filename` - File to store source connector offsets

The parameters that are configured here are intended for producers and consumers used by Kafka Connect to access the configuration, offset and status topics. For configuration of the producers used by Kafka source tasks and the consumers used by Kafka sink tasks, the same parameters can be used but need to be prefixed with `producer.` and `consumer.` respectively. The only Kafka client parameter that is inherited without a prefix from the worker configuration is `bootstrap.servers`, which in most cases will be sufficient, since the same cluster is often used for all purposes. A notable exception is a secured cluster, which requires extra parameters to allow connections. These parameters will need to be set up to three times in the worker configuration, once for management access, once for Kafka sources and once for Kafka sinks.

Starting with 2.3.0, client configuration overrides can be configured individually per connector by using the prefixes `producer.override.` and `consumer.override.` for Kafka sources or Kafka sinks respectively. These overrides are included with the rest of the connector's configuration properties.

The remaining parameters are connector configuration files. You may include as many as you want, but all will execute within the same process (on different threads). You can also choose not to specify any connector configuration files on the command line, and instead use the REST API to create connectors at runtime after your standalone worker starts.

Distributed mode handles automatic balancing of work, allows you to scale up (or down) dynamically, and offers fault tolerance both in the active tasks and for configuration and offset commit data. Execution is very similar to standalone mode:

```
> bin/connect-distributed.sh config/connect-distributed.properties
```

The difference is in the class which is started and the configuration parameters which change how the Kafka Connect process decides where to store configurations, how to assign work, and where to store offsets and task statuses. In the distributed mode, Kafka Connect stores the offsets, configs and task statuses in Kafka topics. It is recommended to manually create the topics for offset, configs and statuses in order to achieve the desired the number of partitions and replication factors. If the topics are not yet created when starting Kafka Connect, the topics will be auto created with default number of partitions and replication factor, which may not be best suited for its usage.

In particular, the following configuration parameters, in addition to the common settings mentioned above, are critical to set before starting your cluster:

- `group.id` (default `connect-cluster`) - unique name for the cluster, used in forming the Connect cluster group; note that this **must not conflict** with consumer group IDs
- `config.storage.topic` (default `connect-configs`) - topic to use for storing connector and task configurations; note that this should be a single partition, highly replicated, compacted topic. You may need to manually create the topic to ensure the correct configuration as auto created topics may have multiple partitions or be automatically configured for deletion rather than compaction
- `offset.storage.topic` (default `connect-offsets`) - topic to use for storing offsets; this topic should have many partitions, be replicated, and be configured for compaction
- `status.storage.topic` (default `connect-status`) - topic to use for storing statuses; this topic can have multiple partitions, and should be replicated and configured for compaction

Note that in distributed mode the connector configurations are not passed on the command line. Instead, use the REST API described below to create, modify, and destroy connectors.

## [Configuring Connectors](#)

Connector configurations are simple key-value mappings. In both standalone and distributed mode, they are included in the JSON payload for the REST request that creates (or modifies) the connector. In standalone mode these can also be defined in a properties file and passed to the Connect process on the command line.

Most configurations are connector dependent, so they can't be outlined here. However, there are a few common options:

- `name` - Unique name for the connector. Attempting to register again with the same name will fail.
- `connector.class` - The Java class for the connector
- `tasks.max` - The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot achieve this level of parallelism.
- `key.converter` - (optional) Override the default key converter set by the worker.
- `value.converter` - (optional) Override the default value converter set by the worker.

The `connector.class` config supports several formats: the full name or alias of the class for this connector. If the connector is `org.apache.kafka.connect.file.FileStreamSinkConnector`, you can either specify this full name or use `FileStreamSink` or `FileStreamSinkConnector` to make the configuration a bit shorter.

Sink connectors also have a few additional options to control their input. Each sink connector must set one of the following:

- `topics` - A comma-separated list of topics to use as input for this connector
- `topics.regex` - A Java regular expression of topics to use as input for this connector

For any other options, you should consult the documentation for the connector.

## [Transformations](#)

Connectors can be configured with transformations to make lightweight message-at-a-time modifications. They can be convenient for data massaging and event routing.

A transformation chain can be specified in the connector configuration.

- `transforms` - List of aliases for the transformation, specifying the order in which the transformations will be applied.
- `transforms.$alias.type` - Fully qualified class name for the transformation.
- `transforms.$alias.$transformationSpecificConfig` Configuration properties for the transformation

For example, lets take the built-in file source connector and use a transformation to add a static field.

Throughout the example we'll use schemaless JSON data format. To use schemaless format, we changed the following two lines in `connect-standalone.properties` from true to false:

```
key.converter.schemas.enable  
value.converter.schemas.enable
```

The file source connector reads each line as a String. We will wrap each line in a Map and then add a second field to identify the origin of the event. To do this, we use two transformations:

- **HoistField** to place the input line inside a Map
- **InsertField** to add the static field. In this example we'll indicate that the record came from a file connector

After adding the transformations, `connect-file-source.properties` file looks as following:

```
name=local-file-source  
connector.class=FilestreamSource  
tasks.max=1  
file=test.txt  
topic=connect-test  
transforms=MakeMap, InsertSource  
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$value  
transforms.MakeMap.field=line  
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$value  
transforms.InsertSource.static.field=data_source  
transforms.InsertSource.static.value=test-file-source
```

All the lines starting with `transforms` were added for the transformations. You can see the two transformations we created: "InsertSource" and "MakeMap" are aliases that we chose to give the transformations. The transformation types are based on the list of built-in transformations you can see below. Each transformation type has additional configuration: HoistField requires a configuration called "field", which is the name of the field in the map that will include the original String from the file. InsertField transformation lets us specify the field name and the value that we are adding.

When we ran the file source connector on my sample file without the transformations, and then read them using `kafka-console-consumer.sh`, the results were:

```
"foo"  
"bar"  
"hello world"
```

We then create a new file connector, this time after adding the transformations to the configuration file. This time, the results will be:

```
{"line":"foo","data_source":"test-file-source"}  
{"line":"bar","data_source":"test-file-source"}  
{"line":"hello world","data_source":"test-file-source"}
```

You can see that the lines we've read are now part of a JSON map, and there is an extra field with the static value we specified. This is just one example of what you can do with transformations.

### [Included transformations](#)

Several widely-applicable data and routing transformations are included with Kafka Connect:

- InsertField - Add a field using either static data or record metadata
- ReplaceField - Filter or rename fields

- MaskField - Replace field with valid null value for the type (0, empty string, etc) or custom replacement (non-empty string or numeric value only)
- ValueToKey - Replace the record key with a new key formed from a subset of fields in the record value
- HoistField - Wrap the entire event as a single field inside a Struct or a Map
- ExtractField - Extract a specific field from Struct and Map and include only this field in results
- SetSchemaMetadata - modify the schema name or version
- TimestampRouter - Modify the topic of a record based on original topic and timestamp. Useful when using a sink that needs to write to different tables or indexes based on timestamps
- RegexRouter - modify the topic of a record based on original topic, replacement string and a regular expression
- Filter - Removes messages from all further processing. This is used with a [predicate](#) to selectively filter certain messages.
- InsertHeader - Add a header using static data
- HeadersFrom - Copy or move fields in the key or value to the record headers
- DropHeaders - Remove headers by name

Details on how to configure each transformation are listed below:

#### [`org.apache.kafka.connect.transforms.InsertField`](#)

Insert field(s) using attributes from the record metadata or a configured static value.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.InsertField$Key`) or value (`org.apache.kafka.connect.transforms.InsertField$value`).

- [`offset.field`](#)

Field name for Kafka offset - only applicable to sink connectors.

Suffix with `!` to make this a required field, or `?` to keep it optional (the default).

Type:	<code>string</code>
Default:	null
Valid Values:	
Importance:	medium

- [`partition.field`](#)

Field name for Kafka partition. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).

Type:	<code>string</code>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*static.field\*\*](#)

Field name for static data field. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*static.value\*\*](#)

Static field value, if field name configured.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*timestamp.field\*\*](#)

Field name for record timestamp. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

- [\*\*topic.field\*\*](#)

Field name for Kafka topic. Suffix with `!` to make this a required field, or `?` to keep it optional (the default).

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	medium

## [org.apache.kafka.connect.transforms.ReplaceField](#)

Filter or rename fields.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.ReplaceField$Key`) or value (`org.apache.kafka.connect.transforms.ReplaceField$value`).

- [exclude](#)

Fields to exclude. This takes precedence over the fields to include.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	medium

- [include](#)

Fields to include. If specified, only these fields will be used.

Type:	<b>list</b>
Default:	""
Valid Values:	
Importance:	medium

- [renames](#)

Field rename mappings.

Type:	<b>list</b>
Default:	""
Valid Values:	list of colon-delimited pairs, e.g. <code>foo:bar,abc:xyz</code>
Importance:	medium

- [blacklist](#)

Deprecated. Use exclude instead.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

- [whitelist](#)

Deprecated. Use include instead.

Type:	<b>list</b>
Default:	null
Valid Values:	
Importance:	low

## [org.apache.kafka.connect.transforms.MaskField](#)

Mask specified fields with a valid null value for the field type (i.e. 0, false, empty string, and so on).

For numeric and string fields, an optional replacement value can be specified that is converted to the correct type.

Use the concrete transformation type designed for the record key  
(`org.apache.kafka.connect.transforms.MaskField$Key`) or value  
(`org.apache.kafka.connect.transforms.MaskField$value`).

- [\*\*fields\*\*](#)

Names of fields to mask.

Type:	<b>list</b>
Default:	
Valid Values:	non-empty list
Importance:	high

- [\*\*replacement\*\*](#)

Custom value replacement, that will be applied to all 'fields' values (numeric or non-empty string values only).

Type:	<b>string</b>
Default:	null
Valid Values:	non-empty string
Importance:	low

## [org.apache.kafka.connect.transforms.ValueToKey](#)

Replace the record key with a new key formed from a subset of fields in the record value.

- [\*\*fields\*\*](#)

Field names on the record value to extract as the record key.

Type:	<b>list</b>
Default:	
Valid Values:	non-empty list
Importance:	high

## [`org.apache.kafka.connect.transforms.HoistField`](#)

Wrap data using the specified field name in a Struct when schema present, or a Map in the case of schemaless data.

Use the concrete transformation type designed for the record key  
(`org.apache.kafka.connect.transforms.HoistField$Key`) or value  
(`org.apache.kafka.connect.transforms.HoistField$value`).

- [\*\*field\*\*](#)

Field name for the single field that will be created in the resulting Struct or Map.

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	medium

## [`org.apache.kafka.connect.transforms.ExtractField`](#)

Extract the specified field from a Struct when schema present, or a Map in the case of schemaless data. Any null values are passed through unmodified.

Use the concrete transformation type designed for the record key  
(`org.apache.kafka.connect.transforms.ExtractField$Key`) or value  
(`org.apache.kafka.connect.transforms.ExtractField$value`).

- [\*\*field\*\*](#)

Field name to extract.

Type:	<b>string</b>
Default:	
Valid Values:	
Importance:	medium

## [`org.apache.kafka.connect.transforms.SetSchemaMetadata`](#)

Set the schema name, version or both on the record's key  
(`org.apache.kafka.connect.transforms.SetSchemaMetadata$key`) or value  
(`org.apache.kafka.connect.transforms.SetSchemaMetadata$value`) schema.

- [schema.name](#)

Schema name to set.

Type:	<b>string</b>
Default:	null
Valid Values:	
Importance:	high

- [schema.version](#)

Schema version to set.

Type:	<b>int</b>
Default:	null
Valid Values:	
Importance:	high

## [org.apache.kafka.connect.transforms.TimestampRouter](#)

Update the record's topic field as a function of the original topic value and the record timestamp.

This is mainly useful for sink connectors, since the topic field is often used to determine the equivalent entity name in the destination system(e.g. database table or search index name).

- [timestamp.format](#)

Format string for the timestamp that is compatible with `java.text.SimpleDateFormat`.

Type:	<b>string</b>
Default:	yyyyMMdd
Valid Values:	
Importance:	high

- [topic.format](#)

Format string which can contain  `${topic}` and  `${timestamp}` as placeholders for the topic and timestamp, respectively.

Type:	<b>string</b>
Default:	<code> \${topic}-\${timestamp}</code>
Valid Values:	
Importance:	high

## [org.apache.kafka.connect.transforms.RegexRouter](#)

Update the record topic using the configured regular expression and replacement string.

Under the hood, the regex is compiled to a `java.util.regex.Pattern`. If the pattern matches the input topic, `java.util.regex.Matcher#replaceFirst()` is used with the replacement string to obtain the new topic.

- [regex](#)

Regular expression to use for matching.

Type:	string
Default:	
Valid Values:	valid regex
Importance:	high

- [replacement](#)

Replacement string.

Type:	string
Default:	
Valid Values:	
Importance:	high

## [org.apache.kafka.connect.transforms.Flatten](#)

Flatten a nested data structure, generating names for each field by concatenating the field names at each level with a configurable delimiter character. Applies to Struct when schema present, or a Map in the case of schemaless data. Array fields and their contents are not modified. The default delimiter is '.'.

Use the concrete transformation type designed for the record key (`org.apache.kafka.connect.transforms.Flatten$Key`) or value (`org.apache.kafka.connect.transforms.Flatten$value`).

- [delimiter](#)

Delimiter to insert between field names from the input record when generating field names for the output record

Type:	string
Default:	.
Valid Values:	
Importance:	medium

## [org.apache.kafka.connect.transforms.Cast](#)

Cast fields or the entire key or value to a specific type, e.g. to force an integer field to a smaller width. Cast from integers, floats, boolean and string to any other type, and cast binary to string (base64 encoded).

Use the concrete transformation type designed for the record key  
(`org.apache.kafka.connect.transforms.Cast$Key`) or value  
(`org.apache.kafka.connect.transforms.Cast$value`).

- [spec](#)

List of fields and the type to cast them to of the form field1:type,field2:type to cast fields of Maps or Structs. A single type to cast the entire value. Valid types are int8, int16, int32, int64, float32, float64, boolean, and string. Note that binary fields can only be cast to string.

Type:	<b>list</b>
Default:	
Valid Values:	list of colon-delimited pairs, e.g. <code>foo:bar,abc:xyz</code>
Importance:	high

## [org.apache.kafka.connect.transforms.TimestampConverter](#)

Convert timestamps between different formats such as Unix epoch, strings, and Connect Date/Timestamp types. Applies to individual fields or to the entire value.

Use the concrete transformation type designed for the record key  
(`org.apache.kafka.connect.transforms.TimestampConverter$key`) or value  
(`org.apache.kafka.connect.transforms.TimestampConverter$value`).

- [target.type](#)

The desired timestamp representation: string, unix, Date, Time, or Timestamp

Type:	<b>string</b>
Default:	
Valid Values:	[string, unix, Date, Time, Timestamp]
Importance:	high

- [field](#)

The field containing the timestamp, or empty if the entire value is a timestamp

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	high

- [format](#)

A SimpleDateFormat-compatible format for the timestamp. Used to generate the output when type=string or used to parse the input if the input is a string.

Type:	<b>string</b>
Default:	""
Valid Values:	
Importance:	medium

- [unix.precision](#)

The desired Unix precision for the timestamp: seconds, milliseconds, microseconds, or nanoseconds. Used to generate the output when type=unix or used to parse the input if the input is a Long. Note: This SMT will cause precision loss during conversions from, and to, values with sub-millisecond components.

Type:	<b>string</b>
Default:	milliseconds
Valid Values:	[nanoseconds, microseconds, milliseconds, seconds]
Importance:	low

## [org.apache.kafka.connect.transforms.Filter](#)

Drops all records, filtering them from subsequent transformations in the chain. This is intended to be used conditionally to filter out records matching (or not matching) a particular Predicate.

## [org.apache.kafka.connect.transforms.InsertHeader](#)

Add a header to each record.

- [header](#)

The name of the header.

Type:	<b>string</b>
Default:	
Valid Values:	non-null string
Importance:	high

- [value.literal](#)

The literal value that is to be set as the header value on all records.

Type:	<b>string</b>
Default:	
Valid Values:	non-null string
Importance:	high

### [org.apache.kafka.connect.transforms.DropHeaders](#)

Removes one or more headers from each record.

- [headers](#)

The name of the headers to be removed.

Type:	<b>list</b>
Default:	
Valid Values:	non-empty list
Importance:	high

### [org.apache.kafka.connect.transforms.HeaderFrom](#)

Moves or copies fields in the key/value of a record into that record's headers. Corresponding elements of `fields` and `headers` together identify a field and the header it should be moved or copied to. Use the concrete transformation type designed for the record key (`(org.apache.kafka.connect.transforms.HeaderFrom$Key)` or value (`(org.apache.kafka.connect.transforms.HeaderFrom$value)`).

- [fields](#)

Field names in the record whose values are to be copied or moved to headers.

Type:	<b>list</b>
Default:	
Valid Values:	non-empty list
Importance:	high

- [headers](#)

Header names, in the same order as the field names listed in the fields configuration property.

Type:	<b>list</b>
Default:	
Valid Values:	non-empty list
Importance:	high

- [operation](#)

Either `move` if the fields are to be moved to the headers (removed from the key/value), or `copy` if the fields are to be copied to the headers (retained in the key/value).

Type:	<code>string</code>
Default:	
Valid Values:	[move, copy]
Importance:	high

## [Predicates](#)

Transformations can be configured with predicates so that the transformation is applied only to messages which satisfy some condition. In particular, when combined with the **Filter** transformation predicates can be used to selectively filter out certain messages.

Predicates are specified in the connector configuration.

- `predicates` - Set of aliases for the predicates to be applied to some of the transformations.
- `predicates.$alias.type` - Fully qualified class name for the predicate.
- `predicates.$alias.$predicateSpecificConfig` - Configuration properties for the predicate.

All transformations have the implicit config properties `predicate` and `negate`. A particular predicate is associated with a transformation by setting the transformation's `predicate` config to the predicate's alias. The predicate's value can be reversed using the `negate` configuration property.

For example, suppose you have a source connector which produces messages to many different topics and you want to:

- filter out the messages in the 'foo' topic entirely
- apply the ExtractField transformation with the field name 'other\_field' to records in all topics *except* the topic 'bar'

To do this we need first to filter out the records destined for the topic 'foo'. The Filter transformation removes records from further processing, and can use the TopicNameMatches predicate to apply the transformation only to records in topics which match a certain regular expression. TopicNameMatches's only configuration property is `pattern`, which is a Java regular expression for matching against the topic name. The configuration would look like this:

```
transforms=Filter
transforms.Filter.type=org.apache.kafka.connect.transforms.Filter
transforms.Filter.predicate=IsFoo

predicates=IsFoo
predicates.IsFoo.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
predicates.IsFoo.pattern=foo
```

Next we need to apply ExtractField only when the topic name of the record is not 'bar'. We can't just use TopicNameMatches directly, because that would apply the transformation to matching topic names, not topic names which do *not* match. The transformation's implicit `negate` config properties allows us to invert the set of records which a predicate matches. Adding the configuration for this to the previous example we arrive at:

```
transforms=Filter,Extract
transforms.Filter.type=org.apache.kafka.connect.transforms.Filter
transforms.Filter.predicate=IsFoo
```

```

transforms.Extract.type=org.apache.kafka.connect.transforms.ExtractField$Key
transforms.Extract.field=other_field
transforms.Extract.predicate=IsBar
transforms.Extract.negate=true

predicates=IsFoo,IsBar
predicates.IsFoo.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
predicates.IsFoo.pattern=foo

predicates.IsBar.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
predicates.IsBar.pattern=bar

```

Kafka Connect includes the following predicates:

- `TopicNameMatches` - matches records in a topic with a name matching a particular Java regular expression.
- `HasHeaderKey` - matches records which have a header with the given key.
- `RecordIsTombstone` - matches tombstone records, that is records with a null value.

Details on how to configure each predicate are listed below:

#### [org.apache.kafka.connect.transforms.predicates.HasHeaderKey](#)

A predicate which is true for records with at least one header with the configured name.

- [name](#)

The header name.

Type:	<code>string</code>
Default:	
Valid Values:	non-empty string
Importance:	medium

#### [org.apache.kafka.connect.transforms.predicates.RecordIsTombstone](#)

A predicate which is true for records which are tombstones (i.e. have null value).

#### [org.apache.kafka.connect.transforms.predicates.TopicNameMatches](#)

A predicate which is true for records with a topic name that matches the configured regular expression.

- [pattern](#)

A Java regular expression for matching against the name of a record's topic.

Type:	<code>string</code>
Default:	
Valid Values:	non-empty string, valid regex
Importance:	medium

## [REST API](#)

Since Kafka Connect is intended to be run as a service, it also provides a REST API for managing connectors. This REST API is available in both standalone and distributed mode. The REST API server can be configured using the `listeners` configuration option. This field should contain a list of listeners in the following format: `protocol://host:port,protocol2://host2:port2`. Currently supported protocols are `http` and `https`. For example:

```
listeners=http://localhost:8080,https://localhost:8443
```

By default, if no `listeners` are specified, the REST server runs on port 8083 using the HTTP protocol. When using HTTPS, the configuration has to include the SSL configuration. By default, it will use the `ssl.*` settings. In case it is needed to use different configuration for the REST API than for connecting to Kafka brokers, the fields can be prefixed with `listeners.https`. When using the prefix, only the prefixed options will be used and the `ssl.*` options without the prefix will be ignored. Following fields can be used to configure HTTPS for the REST API:

- `ssl.keystore.location`
- `ssl.keystore.password`
- `ssl.keystore.type`
- `ssl.key.password`
- `ssl.truststore.location`
- `ssl.truststore.password`
- `ssl.truststore.type`
- `ssl.enabled.protocols`
- `ssl.provider`
- `ssl.protocol`
- `ssl.cipher.suites`
- `ssl.keymanager.algorithm`
- `ssl.secure.random.implementation`
- `ssl.trustmanager.algorithm`
- `ssl.endpoint.identification.algorithm`
- `ssl.client.auth`

The REST API is used not only by users to monitor / manage Kafka Connect. In distributed mode, it is also used for the Kafka Connect cross-cluster communication. Some requests received on the follower nodes REST API will be forwarded to the leader node REST API. In case the URL under which is given host reachable is different from the URI which it listens on, the configuration options `rest.advertised.host.name`, `rest.advertised.port` and `rest.advertised.listener` can be used to change the URI which will be used by the follower nodes to connect with the leader. When using both HTTP and HTTPS listeners, the `rest.advertised.listener` option can be also used to define which listener will be used for the cross-cluster communication. When using HTTPS for communication between nodes, the same `ssl.*` or `listeners.https` options will be used to configure the HTTPS client.

The following are the currently supported REST API endpoints:

- `GET /connectors` - return a list of active connectors
- `POST /connectors` - create a new connector; the request body should be a JSON object containing a string `name` field and an object `config` field with the connector configuration parameters
- `GET /connectors/{name}` - get information about a specific connector
- `GET /connectors/{name}/config` - get the configuration parameters for a specific connector
- `PUT /connectors/{name}/config` - update the configuration parameters for a specific connector

- `GET /connectors/{name}/status` - get current status of the connector, including if it is running, failed, paused, etc., which worker it is assigned to, error information if it has failed, and the state of all its tasks
- `GET /connectors/{name}/tasks` - get a list of tasks currently running for a connector
- `GET /connectors/{name}/tasks/{taskId}/status` - get current status of the task, including if it is running, failed, paused, etc., which worker it is assigned to, and error information if it has failed
- `PUT /connectors/{name}/pause` - pause the connector and its tasks, which stops message processing until the connector is resumed
- `PUT /connectors/{name}/resume` - resume a paused connector (or do nothing if the connector is not paused)
- `POST /connectors/{name}/restart?includeTasks=<true|false>&onlyFailed=<true|false>`

- restart a connector and its tasks instances.

- the "includeTasks" parameter specifies whether to restart the connector instance and task instances ("includeTasks=true") or just the connector instance ("includeTasks=false"), with the default ("false") preserving the same behavior as earlier versions.
- the "onlyFailed" parameter specifies whether to restart just the instances with a FAILED status ("onlyFailed=true") or all instances ("onlyFailed=false"), with the default ("false") preserving the same behavior as earlier versions.
- `POST /connectors/{name}/tasks/{taskId}/restart` - restart an individual task (typically because it has failed)
- `DELETE /connectors/{name}` - delete a connector, halting all tasks and deleting its configuration
- `GET /connectors/{name}/topics` - get the set of topics that a specific connector is using since the connector was created or since a request to reset its set of active topics was issued
- `PUT /connectors/{name}/topics/reset` - send a request to empty the set of active topics of a connector

Kafka Connect also provides a REST API for getting information about connector plugins:

- `GET /connector-plugins` - return a list of connector plugins installed in the Kafka Connect cluster. Note that the API only checks for connectors on the worker that handles the request, which means you may see inconsistent results, especially during a rolling upgrade if you add new connector jars
- `PUT /connector-plugins/{connector-type}/config/validate` - validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

The following is a supported REST request at the top-level (root) endpoint:

- `GET /` - return basic information about the Kafka Connect cluster such as the version of the Connect worker that serves the REST request (including git commit ID of the source code) and the Kafka cluster ID that is connected to.

For the complete specification of the REST API, see the [OpenAPI documentation](#)

## [Error Reporting in Connect](#)

Kafka Connect provides error reporting to handle errors encountered along various stages of processing. By default, any error encountered during conversion or within transformations will cause the connector to fail. Each connector configuration can also enable tolerating such errors by skipping them, optionally writing each error and the details of the failed operation and problematic record (with various levels of detail) to the Connect application log. These mechanisms also capture errors when a sink connector is processing the messages consumed from its Kafka topics, and all of the errors can be written to a configurable "dead letter queue" (DLQ) Kafka topic.

To report errors within a connector's converter, transforms, or within the sink connector itself to the log, set `errors.log.enable=true` in the connector configuration to log details of each error and problem record's topic, partition, and offset. For additional debugging purposes, set `errors.log.include.messages=true` to also log the problem record key, value, and headers to the log (note this may log sensitive information).

To report errors within a connector's converter, transforms, or within the sink connector itself to a dead letter queue topic, set `errors.deadletterqueue.topic.name`, and optionally `errors.deadletterqueue.context.headers.enable=true`.

By default connectors exhibit "fail fast" behavior immediately upon an error or exception. This is equivalent to adding the following configuration properties with their defaults to a connector configuration:

```
# disable retries on failure
errors.retry.timeout=0

# do not log the error and their contexts
errors.log.enable=false

# do not record errors in a dead letter queue topic
errors.deadletterqueue.topic.name=

# Fail on first error
errors.tolerance=none
```

These and other related connector configuration properties can be changed to provide different behavior. For example, the following configuration properties can be added to a connector configuration to setup error handling with multiple retries, logging to the application logs and the `my-connector-errors` Kafka topic, and tolerating all errors by reporting them rather than failing the connector task:

```
# retry for at most 10 minutes times waiting up to 30 seconds between consecutive
failures
errors.retry.timeout=600000
errors.retry.delay.max.ms=30000

# log error context along with application logs, but do not include configs and
messages
errors.log.enable=true
errors.log.include.messages=false

# produce error context into the Kafka topic
errors.deadletterqueue.topic.name=my-connector-errors

# Tolerate all errors.
errors.tolerance=all
```

## Exactly-once support

Kafka Connect is capable of providing exactly-once semantics for sink connectors (as of version 0.11.0) and source connectors (as of version 3.3.0). Please note that **support for exactly-once semantics is highly dependent on the type of connector you run**. Even if you set all the correct worker properties in the configuration for each node in a cluster, if a connector is not designed to, or cannot take advantage of the capabilities of the Kafka Connect framework, exactly-once may not be possible.

## [Sink connectors](#)

If a sink connector supports exactly-once semantics, to enable exactly-once at the Connect worker level, you must ensure its consumer group is configured to ignore records in aborted transactions. You can do this by setting the worker property `consumer.isolation.level` to `read_committed` or, if running a version of Kafka Connect that supports it, using a [connector client config override policy](#) that allows the `consumer.override.isolation.level` property to be set to `read_committed` in individual connector configs. There are no additional ACL requirements.

## [Source connectors](#)

If a source connector supports exactly-once semantics, you must configure your Connect cluster to enable framework-level support for exactly-once source connectors. Additional ACLs may be necessary if running against a secured Kafka cluster. Note that exactly-once support for source connectors is currently only available in distributed mode; standalone Connect workers cannot provide exactly-once semantics.

### Worker configuration

For new Connect clusters, set the `exactly.once.source.support` property to `enabled` in the worker config for each node in the cluster. For existing clusters, two rolling upgrades are necessary. During the first upgrade, the `exactly.once.source.support` property should be set to `preparing`, and during the second, it should be set to `enabled`.

### ACL requirements

With exactly-once source support enabled, the principal for each Connect worker will require the following ACLs:

OPERATION	RESOURCE TYPE	RESOURCE NAME	NOTE
-----------	---------------	---------------	------

Write	TransactionalId	<code>connect-cluster-\${groupId}</code> , where <code>\${groupId}</code> is the <code>group.id</code> of the cluster	
Describe	TransactionalId	<code>connect-cluster-\${groupId}</code> , where <code>\${groupId}</code> is the <code>group.id</code> of the cluster	
IdempotentWrite	Cluster	ID of the Kafka cluster that hosts the worker's config topic	The IdempotentWrite ACL has been deprecated as of 2.8 and will only be necessary for Connect clusters running on pre-2.8 Kafka clusters

And the principal for each individual connector will require the following ACLs:

OPERATION	RESOURCE TYPE	RESOURCE NAME	NOTE
Write	TransactionalId	<code>/\${groupId}-\${connector}-\${taskId}</code> , for each task that the connector will create, where <code>(groupId)</code> is the <code>group.id</code> of the Connect cluster, <code>(connector)</code> is the name of the connector, and <code>(taskId)</code> is the ID of the task (starting from zero)	A wildcard prefix of <code>/\${groupId}-\${connector}*` can be used for convenience if there is no risk of conflict with other transactional IDs or if conflicts are acceptable to the user.</code>
Describe	TransactionalId	<code>/\${groupId}-\${connector}-\${taskId}</code> , for each task that the connector will create, where <code>(groupId)</code> is the <code>group.id</code> of the Connect cluster, <code>(connector)</code> is the name of the connector, and <code>(taskId)</code> is the ID of the task (starting from zero)	A wildcard prefix of <code>/\${groupId}-\${connector}*` can be used for convenience if there is no risk of conflict with other transactional IDs or if conflicts are acceptable to the user.</code>
Write	Topic	Offsets topic used by the connector, which is either the value of the <code>offsets.storage.topic</code> property in the connector's configuration if provided, or the value of the <code>offsets.storage.topic</code> property in the worker's configuration if not.	
Read	Topic	Offsets topic used by the connector, which is either the value of the <code>offsets.storage.topic</code> property in the connector's configuration if provided, or the value of the <code>offsets.storage.topic</code> property in the worker's configuration if not.	
Describe	Topic	Offsets topic used by the connector, which is either the value of the <code>offsets.storage.topic</code> property in the connector's configuration if provided, or the value of the <code>offsets.storage.topic</code> property in the worker's configuration if not.	
Create	Topic	Offsets topic used by the connector, which is either the value of the <code>offsets.storage.topic</code> property in the connector's configuration if provided, or the value of the <code>offsets.storage.topic</code> property in the worker's configuration if not.	Only necessary if the offsets topic for the connector does not exist yet
IdempotentWrite	Cluster	ID of the Kafka cluster that the source connector writes to	The IdempotentWrite ACL has been deprecated as of 2.8 and will only be necessary for Connect clusters running on pre-2.8 Kafka clusters

## 8.3 Connector Development Guide

This guide describes how developers can write new connectors for Kafka Connect to move data between Kafka and other systems. It briefly reviews a few key concepts and then describes how to create a simple connector.

### [Core Concepts and APIs](#)

#### [Connectors and Tasks](#)

To copy data between Kafka and another system, users create a `Connector` for the system they want to pull data from or push data to. Connectors come in two flavors: `SourceConnectors` import data from another system (e.g. `JDBCSourceConnector` would import a relational database into Kafka) and `SinkConnectors` export data (e.g. `HDFSSinkConnector` would export the contents of a Kafka topic to an HDFS file).

`Connectors` do not perform any data copying themselves: their configuration describes the data to be copied, and the `Connector` is responsible for breaking that job into a set of `Tasks` that can be distributed to workers. These `Tasks` also come in two corresponding flavors: `sourceTask` and `sinkTask`.

With an assignment in hand, each `Task` must copy its subset of the data to or from Kafka. In Kafka Connect, it should always be possible to frame these assignments as a set of input and output streams consisting of records with consistent schemas. Sometimes this mapping is obvious: each file in a set of log files can be considered a stream with each parsed line forming a record using the same schema and offsets stored as byte offsets in the file. In other cases it may require more effort to map to this model: a JDBC connector can map each table to a stream, but the offset is less clear. One possible mapping uses a timestamp column to generate queries incrementally returning new data, and the last queried timestamp can be used as the offset.

## [Streams and Records](#)

Each stream should be a sequence of key-value records. Both the keys and values can have complex structure -- many primitive types are provided, but arrays, objects, and nested data structures can be represented as well. The runtime data format does not assume any particular serialization format; this conversion is handled internally by the framework.

In addition to the key and value, records (both those generated by sources and those delivered to sinks) have associated stream IDs and offsets. These are used by the framework to periodically commit the offsets of data that have been processed so that in the event of failures, processing can resume from the last committed offsets, avoiding unnecessary reprocessing and duplication of events.

## [Dynamic Connectors](#)

Not all jobs are static, so `Connector` implementations are also responsible for monitoring the external system for any changes that might require reconfiguration. For example, in the `JDBCSourceConnector` example, the `Connector` might assign a set of tables to each `Task`. When a new table is created, it must discover this so it can assign the new table to one of the `Tasks` by updating its configuration. When it notices a change that requires reconfiguration (or a change in the number of `Tasks`), it notifies the framework and the framework updates any corresponding `Tasks`.

## [Developing a Simple Connector](#)

Developing a connector only requires implementing two interfaces, the `Connector` and `Task`. A simple example is included with the source code for Kafka in the `file` package. This connector is meant for use in standalone mode and has implementations of a `SourceConnector`/`SourceTask` to read each line of a file and emit it as a record and a `SinkConnector`/`SinkTask` that writes each record to a file.

The rest of this section will walk through some code to demonstrate the key steps in creating a connector, but developers should also refer to the full example source code as many details are omitted for brevity.

### [Connector Example](#)

We'll cover the `SourceConnector` as a simple example. `SinkConnector` implementations are very similar. Start by creating the class that inherits from `SourceConnector` and add a field that will store the configuration information to be propagated to the task(s) (the topic to send data to, and optionally - the filename to read from and the maximum batch size):

```
public class FileStreamSourceConnector extends SourceConnector {  
    private Map<String, String> props;
```

The easiest method to fill in is `taskClass()`, which defines the class that should be instantiated in worker processes to actually read the data:

```
@Override  
public Class<? extends Task> taskClass() {  
    return FileStreamSourceTask.class;  
}
```

We will define the `FilestreamSourceTask` class below. Next, we add some standard lifecycle methods, `start()` and `stop()`:

```
@Override
public void start(Map<String, String> props) {
    // Initialization logic and setting up of resources can take place in this method.
    // This connector doesn't need to do any of that, but we do log a helpful message
    // to the user.

    this.props = props;
    AbstractConfig config = new AbstractConfig(CONFIG_DEF, props);
    String filename = config.getString(FILE_CONFIG);
    filename = (filename == null || filename.isEmpty()) ? "standard input" :
    config.getString(FILE_CONFIG);
    log.info("Starting file source connector reading from {}", filename);
}

@Override
public void stop() {
    // Nothing to do since no background monitoring is required.
}
```

Finally, the real core of the implementation is in `taskConfigs()`. In this case we are only handling a single file, so even though we may be permitted to generate more tasks as per the `maxTasks` argument, we return a list with only one entry:

```
@Override
public List<Map<String, String>> taskConfigs(int maxTasks) {
    // Note that the task configs could contain configs additional to or different
    // from the connector configs if needed. For instance,
    // if different tasks have different responsibilities, or if different tasks are
    // meant to process different subsets of the source data stream).
    ArrayList<Map<String, String>> configs = new ArrayList<>();
    // Only one input stream makes sense.
    configs.add(props);
    return configs;
}
```

Even with multiple tasks, this method implementation is usually pretty simple. It just has to determine the number of input tasks, which may require contacting the remote service it is pulling data from, and then divvy them up. Because some patterns for splitting work among tasks are so common, some utilities are provided in `ConnectorUtils` to simplify these cases.

Note that this simple example does not include dynamic input. See the discussion in the next section for how to trigger updates to task configs.

### [Task Example - Source Task](#)

Next we'll describe the implementation of the corresponding `SourceTask`. The implementation is short, but too long to cover completely in this guide. We'll use pseudo-code to describe most of the implementation, but you can refer to the source code for the full example.

Just as with the connector, we need to create a class inheriting from the appropriate base `Task` class. It also has some standard lifecycle methods:

```
public class FilestreamSourceTask extends SourceTask {
    private String filename;
    private InputStream stream;
    private String topic;
```

```

private int batchsize;

@Override
public void start(Map<String, String> props) {
    filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
    stream = openOrThrowError(filename);
    topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    batchsize = props.get(FileStreamSourceConnector.TASK_BATCH_SIZE_CONFIG);
}

@Override
public synchronized void stop() {
    stream.close();
}

```

These are slightly simplified versions, but show that these methods should be relatively simple and the only work they should perform is allocating or freeing resources. There are two points to note about this implementation. First, the `start()` method does not yet handle resuming from a previous offset, which will be addressed in a later section. Second, the `stop()` method is synchronized. This will be necessary because `sourceTasks` are given a dedicated thread which they can block indefinitely, so they need to be stopped with a call from a different thread in the Worker.

Next, we implement the main functionality of the task, the `poll()` method which gets events from the input system and returns a `List<SourceRecord>`:

```

@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition =
Collections.singletonMap("filename", filename);
                Map<String, Object> sourceoffset =
Collections.singletonMap("position", streamOffset);
                records.add(new SourceRecord(sourcePartition, sourceoffset, topic,
Schema.STRING_SCHEMA, line));
                if (records.size() >= batchsize) {
                    return records;
                }
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling stop. Allow
        to return
        // null, and driving thread will handle any shutdown if necessary.
    }
    return null;
}

```

Again, we've omitted some details, but we can see the important steps: the `poll()` method is going to be called repeatedly, and for each call it will loop trying to read records from the file. For each line it reads, it also tracks the file offset. It uses this information to create an output `SourceRecord` with four pieces of information: the source partition (there is only one, the single file being read), source offset (byte offset in the file), output topic name, and output value (the line, and we include a schema indicating

this value will always be a string). Other variants of the `SourceRecord` constructor can also include a specific output partition, a key, and headers.

Note that this implementation uses the normal Java `InputStream` interface and may sleep if data is not available. This is acceptable because Kafka Connect provides each task with a dedicated thread. While task implementations have to conform to the basic `poll()` interface, they have a lot of flexibility in how they are implemented. In this case, an NIO-based implementation would be more efficient, but this simple approach works, is quick to implement, and is compatible with older versions of Java.

Although not used in the example, `SourceTask` also provides two APIs to commit offsets in the source system: `commit` and `commitRecord`. The APIs are provided for source systems which have an acknowledgement mechanism for messages. Overriding these methods allows the source connector to acknowledge messages in the source system, either in bulk or individually, once they have been written to Kafka. The `commit` API stores the offsets in the source system, up to the offsets that have been returned by `poll`. The implementation of this API should block until the commit is complete. The `commitRecord` API saves the offset in the source system for each `SourceRecord` after it is written to Kafka. As Kafka Connect will record offsets automatically, `SourceTask`s are not required to implement them. In cases where a connector does need to acknowledge messages in the source system, only one of the APIs is typically required.

## [Sink Tasks](#)

The previous section described how to implement a simple `SourceTask`. Unlike `SourceConnector` and `SinkConnector`, `SourceTask` and `SinkTask` have very different interfaces because `SourceTask` uses a pull interface and `SinkTask` uses a push interface. Both share the common lifecycle methods, but the `SinkTask` interface is quite different:

```
public abstract class SinkTask implements Task {  
    public void initialize(SinkTaskContext context) {  
        this.context = context;  
    }  
  
    public abstract void put(Collection<SinkRecord> records);  
  
    public void flush(Map<TopicPartition, OffsetAndMetadata> currentOffsets) {  
    }  
}
```

The `SinkTask` documentation contains full details, but this interface is nearly as simple as the `SourceTask`. The `put()` method should contain most of the implementation, accepting sets of `SinkRecords`, performing any required translation, and storing them in the destination system. This method does not need to ensure the data has been fully written to the destination system before returning. In fact, in many cases internal buffering will be useful so an entire batch of records can be sent at once, reducing the overhead of inserting events into the downstream data store. The `SinkRecords` contain essentially the same information as `SourceRecords`: Kafka topic, partition, offset, the event key and value, and optional headers.

The `flush()` method is used during the offset commit process, which allows tasks to recover from failures and resume from a safe point such that no events will be missed. The method should push any outstanding data to the destination system and then block until the write has been acknowledged. The `offsets` parameter can often be ignored, but is useful in some cases where implementations want to store offset information in the destination store to provide exactly-once delivery. For example, an HDFS connector could do this and use atomic move operations to make sure the `flush()` operation atomically commits the data and offsets to a final location in HDFS.

## [Errant Record Reporter](#)

When [error reporting](#) is enabled for a connector, the connector can use an `ErrantRecordReporter` to report problems with individual records sent to a sink connector. The following example shows how a connector's `SinkTask` subclass might obtain and use the `ErrantRecordReporter`, safely handling a null reporter when the DLQ is not enabled or when the connector is installed in an older Connect runtime that doesn't have this reporter feature:

```
private ErrantRecordReporter reporter;

@Override
public void start(Map<String, String> props) {
    ...
    try {
        reporter = context.errantRecordReporter(); // may be null if DLQ not enabled
    } catch (NoSuchMethodException | NoClassDefFoundError e) {
        // will occur in Connect runtimes earlier than 2.6
        reporter = null;
    }
}

@Override
public void put(Collection<SinkRecord> records) {
    for (SinkRecord record: records) {
        try {
            // attempt to process and send record to data sink
            process(record);
        } catch(Exception e) {
            if (reporter != null) {
                // Send errant record to error reporter
                reporter.report(record, e);
            } else {
                // There's no error reporter, so fail
                throw new ConnectException("Failed on record", e);
            }
        }
    }
}
```

## [Resuming from Previous Offsets](#)

The `SourceTask` implementation included a stream ID (the input filename) and offset (position in the file) with each record. The framework uses this to commit offsets periodically so that in the case of a failure, the task can recover and minimize the number of events that are reprocessed and possibly duplicated (or to resume from the most recent offset if Kafka Connect was stopped gracefully, e.g. in standalone mode or due to a job reconfiguration). This commit process is completely automated by the framework, but only the connector knows how to seek back to the right position in the input stream to resume from that location.

To correctly resume upon startup, the task can use the `SourceContext` passed into its `initialize()` method to access the offset data. In `initialize()`, we would add a bit more code to read the offset (if it exists) and seek to that position:

```

stream = new FileInputStream(filename);
Map<String, Object> offset =
context.offsetStorageReader().offset(collections.singletonMap(FILENAME_FIELD,
filename));
if (offset != null) {
    Long lastRecordedOffset = (Long) offset.get("position");
    if (lastRecordedOffset != null)
        seekToOffset(stream, lastRecordedOffset);
}

```

Of course, you might need to read many keys for each of the input streams. The `OffsetStorageReader` interface also allows you to issue bulk reads to efficiently load all offsets, then apply them by seeking each input stream to the appropriate position.

## [Exactly-once source connectors](#)

### Supporting exactly-once

With the passing of [KIP-618](#), Kafka Connect supports exactly-once source connectors as of version 3.3.0. In order for a source connector to take advantage of this support, it must be able to provide meaningful source offsets for each record that it emits, and resume consumption from the external system at the exact position corresponding to any of those offsets without dropping or duplicating messages.

### Defining transaction boundaries

By default, the Kafka Connect framework will create and commit a new Kafka transaction for each batch of records that a source task returns from its `poll` method. However, connectors can also define their own transaction boundaries, which can be enabled by users by setting the `transaction.boundary` property to `connector` in the config for the connector.

If enabled, the connector's tasks will have access to a `TransactionContext` from their `SourceTaskContext`, which they can use to control when transactions are aborted and committed.

For example, to commit a transaction at least every ten records:

```

private int recordssent;

@Override
public void start(Map<String, String> props) {
    this.recordssent = 0;
}

@Override
public List<SourceRecord> poll() {
    List<SourceRecord> records = fetchRecords();
    boolean shouldCommit = false;
    for (SourceRecord record : records) {
        if (++this.recordssent >= 10) {
            shouldCommit = true;
        }
    }
    if (shouldCommit) {
        this.recordssent = 0;
        this.context.transactionContext().commitTransaction();
    }
    return records;
}

```

Or to commit a transaction for exactly every tenth record:

```

private int recordsSent;

@Override
public void start(Map<String, String> props) {
    this.recordsSent = 0;
}

@Override
public List<SourceRecord> poll() {
    List<SourceRecord> records = fetchRecords();
    for (SourceRecord record : records) {
        if (++this.recordsSent % 10 == 0) {
            this.context.transactionContext().commitTransaction(record);
        }
    }
    return records;
}

```

Most connectors do not need to define their own transaction boundaries. However, it may be useful if files or objects in the source system are broken up into multiple source records, but should be delivered atomically. Additionally, it may be useful if it is impossible to give each source record a unique source offset, if every record with a given offset is delivered within a single transaction.

Note that if the user has not enabled connector-defined transaction boundaries in the connector configuration, the `TransactionContext` returned by `context.transactionContext()` will be `null`.

## Validation APIs

A few additional preflight validation APIs can be implemented by source connector developers.

Some users may require exactly-once semantics from a connector. In this case, they may set the `exactly.once.support` property to `required` in the configuration for the connector. When this happens, the Kafka Connect framework will ask the connector whether it can provide exactly-once semantics with the specified configuration. This is done by invoking the `exactlyOnceSupport` method on the connector.

If a connector doesn't support exactly-once semantics, it should still implement this method to let users know for certain that it cannot provide exactly-once semantics:

```

@Override
public ExactlyOnceSupport exactlyOnceSupport(Map<String, String> props) {
    // This connector cannot provide exactly-once semantics under any conditions
    return ExactlyOnceSupport.UNSUPPORTED;
}

```

Otherwise, a connector should examine the configuration, and return `ExactlyOnceSupport.SUPPORTED` if it can provide exactly-once semantics:

```

@Override
public ExactlyOnceSupport exactlyOnceSupport(Map<String, String> props) {
    // This connector can always provide exactly-once semantics
    return ExactlyOnceSupport.SUPPORTED;
}

```

Additionally, if the user has configured the connector to define its own transaction boundaries, the Kafka Connect framework will ask the connector whether it can define its own transaction boundaries with the specified configuration, using the `canDefineTransactionBoundaries` method:

```
@Override  
public ConnectorTransactionBoundaries canDefineTransactionBoundaries(Map<String,  
String> props) {  
    // This connector can always define its own transaction boundaries  
    return ConnectorTransactionBoundaries.SUPPORTED;  
}
```

This method should only be implemented for connectors that can define their own transaction boundaries in some cases. If a connector is never able to define its own transaction boundaries, it does not need to implement this method.

## [Dynamic Input/Output Streams](#)

Kafka Connect is intended to define bulk data copying jobs, such as copying an entire database rather than creating many jobs to copy each table individually. One consequence of this design is that the set of input or output streams for a connector can vary over time.

Source connectors need to monitor the source system for changes, e.g. table additions/deletions in a database. When they pick up changes, they should notify the framework via the `ConnectorContext` object that reconfiguration is necessary. For example, in a `SourceConnector`:

```
if (inputsChanged())  
    this.context.requestTaskReconfiguration();
```

The framework will promptly request new configuration information and update the tasks, allowing them to gracefully commit their progress before reconfiguring them. Note that in the `SourceConnector` this monitoring is currently left up to the connector implementation. If an extra thread is required to perform this monitoring, the connector must allocate it itself.

Ideally this code for monitoring changes would be isolated to the `Connector` and tasks would not need to worry about them. However, changes can also affect tasks, most commonly when one of their input streams is destroyed in the input system, e.g. if a table is dropped from a database. If the `Task` encounters the issue before the `Connector`, which will be common if the `Connector` needs to poll for changes, the `Task` will need to handle the subsequent error. Thankfully, this can usually be handled simply by catching and handling the appropriate exception.

`SinkConnectors` usually only have to handle the addition of streams, which may translate to new entries in their outputs (e.g., a new database table). The framework manages any changes to the Kafka input, such as when the set of input topics changes because of a regex subscription. `SinkTasks` should expect new input streams, which may require creating new resources in the downstream system, such as a new table in a database. The trickiest situation to handle in these cases may be conflicts between multiple `SinkTasks` seeing a new input stream for the first time and simultaneously trying to create the new resource. `SinkConnectors`, on the other hand, will generally require no special code for handling a dynamic set of streams.

## [Connect Configuration Validation](#)

Kafka Connect allows you to validate connector configurations before submitting a connector to be executed and can provide feedback about errors and recommended values. To take advantage of this, connector developers need to provide an implementation of `config()` to expose the configuration definition to the framework.

The following code in `FilestreamSourceConnector` defines the configuration and exposes it to the framework.

```

static final ConfigDef CONFIG_DEF = new ConfigDef()
    .define(FILE_CONFIG, Type.STRING, null, Importance.HIGH, "Source filename. If not
specified, the standard input will be used")
    .define(TOPIC_CONFIG, Type.STRING, ConfigDef.NO_DEFAULT_VALUE, new
ConfigDef.NonEmptyString(), Importance.HIGH, "The topic to publish data to")
    .define(TASK_BATCH_SIZE_CONFIG, Type.INT, DEFAULT_TASK_BATCH_SIZE, Importance.LOW,
        "The maximum number of records the source task can read from the file each
time it is polled");

public ConfigDef config() {
    return CONFIG_DEF;
}

```

`ConfigDef` class is used for specifying the set of expected configurations. For each configuration, you can specify the name, the type, the default value, the documentation, the group information, the order in the group, the width of the configuration value and the name suitable for display in the UI. Plus, you can provide special validation logic used for single configuration validation by overriding the `Validator` class. Moreover, as there may be dependencies between configurations, for example, the valid values and visibility of a configuration may change according to the values of other configurations. To handle this, `ConfigDef` allows you to specify the dependents of a configuration and to provide an implementation of `Recommender` to get valid values and set visibility of a configuration given the current configuration values.

Also, the `validate()` method in `Connector` provides a default validation implementation which returns a list of allowed configurations together with configuration errors and recommended values for each configuration. However, it does not use the recommended values for configuration validation. You may provide an override of the default implementation for customized configuration validation, which may use the recommended values.

## [Working with Schemas](#)

The FileStream connectors are good examples because they are simple, but they also have trivially structured data -- each line is just a string. Almost all practical connectors will need schemas with more complex data formats.

To create more complex data, you'll need to work with the Kafka Connect `data` API. Most structured records will need to interact with two classes in addition to primitive types: `Schema` and `Struct`.

The API documentation provides a complete reference, but here is a simple example creating a `Schema` and `Struct`:

```

Schema schema = SchemaBuilder.struct().name(NAME)
    .field("name", Schema.STRING_SCHEMA)
    .field("age", Schema.INT_SCHEMA)
    .field("admin", SchemaBuilder.bool().defaultValue(false).build())
    .build();

Struct struct = new Struct(schema)
    .put("name", "Barbara Liskov")
    .put("age", 75);

```

If you are implementing a source connector, you'll need to decide when and how to create schemas. Where possible, you should avoid recomputing them as much as possible. For example, if your connector is guaranteed to have a fixed schema, create it statically and reuse a single instance.

However, many connectors will have dynamic schemas. One simple example of this is a database connector. Considering even just a single table, the schema will not be predefined for the entire connector (as it varies from table to table). But it also may not be fixed for a single table over the lifetime of the connector since the user may execute an `ALTER TABLE` command. The connector must be able to detect these changes and react appropriately.

Sink connectors are usually simpler because they are consuming data and therefore do not need to create schemas. However, they should take just as much care to validate that the schemas they receive have the expected format. When the schema does not match -- usually indicating the upstream producer is generating invalid data that cannot be correctly translated to the destination system -- sink connectors should throw an exception to indicate this error to the system.

## [Kafka Connect Administration](#)

Kafka Connect's [REST layer](#) provides a set of APIs to enable administration of the cluster. This includes APIs to view the configuration of connectors and the status of their tasks, as well as to alter their current behavior (e.g. changing configuration and restarting tasks).

When a connector is first submitted to the cluster, a rebalance is triggered between the Connect workers in order to distribute the load that consists of the tasks of the new connector. This same rebalancing procedure is also used when connectors increase or decrease the number of tasks they require, when a connector's configuration is changed, or when a worker is added or removed from the group as part of an intentional upgrade of the Connect cluster or due to a failure.

In versions prior to 2.3.0, the Connect workers would rebalance the full set of connectors and their tasks in the cluster as a simple way to make sure that each worker has approximately the same amount of work. This behavior can be still enabled by setting `connect.protocol=eager`.

Starting with 2.3.0, Kafka Connect is using by default a protocol that performs [incremental cooperative rebalancing](#) that incrementally balances the connectors and tasks across the Connect workers, affecting only tasks that are new, to be removed, or need to move from one worker to another. Other tasks are not stopped and restarted during the rebalance, as they would have been with the old protocol.

If a Connect worker leaves the group, intentionally or due to a failure, Connect waits for `scheduled.rebalance.max.delay.ms` before triggering a rebalance. This delay defaults to five minutes (`300000ms`) to tolerate failures or upgrades of workers without immediately redistributing the load of a departing worker. If this worker returns within the configured delay, it gets its previously assigned tasks in full. However, this means that the tasks will remain unassigned until the time specified by `scheduled.rebalance.max.delay.ms` elapses. If a worker does not return within that time limit, Connect will reassigned those tasks among the remaining workers in the Connect cluster.

The new Connect protocol is enabled when all the workers that form the Connect cluster are configured with `connect.protocol=compatible`, which is also the default value when this property is missing. Therefore, upgrading to the new Connect protocol happens automatically when all the workers upgrade to 2.3.0. A rolling upgrade of the Connect cluster will activate incremental cooperative rebalancing when the last worker joins on version 2.3.0.

You can use the REST API to view the current status of a connector and its tasks, including the ID of the worker to which each was assigned. For example, the `GET /connectors/file-source/status` request shows the status of a connector named `file-source`:

```
{  
  "name": "file-source",  
  "connector": {  
    "state": "RUNNING",  
    "worker_id": "192.168.1.208:8083"  
  },  
  "tasks": [  
    {  
      "id": 0,  
      "worker_id": "192.168.1.208:8083",  
      "state": "RUNNING",  
      "offsets_committed": 1000000,  
      "offsets_current": 1000000  
    }  
  ]  
}
```

```

        "id": 0,
        "state": "RUNNING",
        "worker_id": "192.168.1.209:8083"
    }
]
}

```

Connectors and their tasks publish status updates to a shared topic (configured with `status.storage.topic`) which all workers in the cluster monitor. Because the workers consume this topic asynchronously, there is typically a (short) delay before a state change is visible through the status API. The following states are possible for a connector or one of its tasks:

- **UNASSIGNED**: The connector/task has not yet been assigned to a worker.
- **RUNNING**: The connector/task is running.
- **PAUSED**: The connector/task has been administratively paused.
- **FAILED**: The connector/task has failed (usually by raising an exception, which is reported in the status output).
- **RESTARTING**: The connector/task is either actively restarting or is expected to restart soon

In most cases, connector and task states will match, though they may be different for short periods of time when changes are occurring or if tasks have failed. For example, when a connector is first started, there may be a noticeable delay before the connector and its tasks have all transitioned to the RUNNING state. States will also diverge when tasks fail since Connect does not automatically restart failed tasks. To restart a connector/task manually, you can use the restart APIs listed above. Note that if you try to restart a task while a rebalance is taking place, Connect will return a 409 (Conflict) status code. You can retry after the rebalance completes, but it might not be necessary since rebalances effectively restart all the connectors and tasks in the cluster.

Starting with 2.5.0, Kafka Connect uses the `status.storage.topic` to also store information related to the topics that each connector is using. Connect Workers use these per-connector topic status updates to respond to requests to the REST endpoint `GET /connectors/{name}/topics` by returning the set of topic names that a connector is using. A request to the REST endpoint `PUT /connectors/{name}/topics/reset` resets the set of active topics for a connector and allows a new set to be populated, based on the connector's latest pattern of topic usage. Upon connector deletion, the set of the connector's active topics is also deleted. Topic tracking is enabled by default but can be disabled by setting `topic.tracking.enable=false`. If you want to disallow requests to reset the active topics of connectors during runtime, set the Worker property `topic.tracking.allow.reset=false`.

It's sometimes useful to temporarily stop the message processing of a connector. For example, if the remote system is undergoing maintenance, it would be preferable for source connectors to stop polling it for new data instead of filling logs with exception spam. For this use case, Connect offers a pause/resume API. While a source connector is paused, Connect will stop polling it for additional records. While a sink connector is paused, Connect will stop pushing new messages to it. The pause state is persistent, so even if you restart the cluster, the connector will not begin message processing again until the task has been resumed. Note that there may be a delay before all of a connector's tasks have transitioned to the PAUSED state since it may take time for them to finish whatever processing they were in the middle of when being paused. Additionally, failed tasks will not transition to the PAUSED state until they have been restarted.

## 9. KAFKA STREAMS

Kafka Streams is a client library for processing and analyzing data stored in Kafka. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, exactly-once processing semantics and simple yet efficient management of application state.

Kafka Streams has a **low barrier to entry**: You can quickly write and run a small-scale proof-of-concept on a single machine; and you only need to run additional instances of your application on multiple machines to scale up to high-volume production workloads. Kafka Streams transparently handles the load balancing of multiple instances of the same application by leveraging Kafka's parallelism model.

Learn More about Kafka Streams read [this](#) Section.