CONFLUENT

Best Practices for Developing Apache Kafka® Applications on Confluent Cloud

Table of Contents

Introduction

This paper covers best practices for configuring, tuning, and monitoring Apache Kafka® applications. Although it is focused on serverless Kafka in Confluent Cloud, this paper can serve as a guide for any Kafka client application.

Confluent Cloud is a fully managed service for Apache Kafka®, a distributed streaming platform technology. Engineers can easily get started with a complete Kafka cluster and the related services they would need to build event streaming applications. Deployable in seconds, Confluent Cloud is ideal for applications that require low latency and the ability to scale.

The key benefits of Confluent Cloud include:

- Developer acceleration in building event streaming applications
- Liberation from operational burden
- Bridge from on premises to cloud with hybrid Kafka service

With Confluent Cloud, an enterprise has a single source of truth across event streaming applications. Mission-critical applications have easy access to all the data they need. Just point your client applications at Confluent Cloud, and Kafka takes care of the rest: load is automatically distributed across brokers, consumer groups automatically rebalance when a consumer is added or removed, the state stores used by applications using the Kafka Streams APIs are automatically backed up to Confluent Cloud, and failures are automatically mitigated. It's an operator's dream come true!

And this paper isn't just for Java applications. These principles are generally applicable to a Kafka client application written in any language. Learn more in the section Compatible Programming Languages.

This paper consists of three main sections that will help you develop, tune, and monitor your Kafka applications:

1. Fundamentals: required information for developing a Kafka client application to Confluent Cloud
2. Monitoring: monitor and measure performance
3. Optimizations: tune your client application for throughput, latency, durability, and availability

This white paper introduces configuration parameters that are relevant for developing Kafka applications to Confluent Cloud. The parameter names, descriptions, and default values are up to date for Confluent Platform version 5.4 (Kafka version 2.4). Consult the documentation for more information on these configuration parameters, topic overrides, and other configuration parameters.

Fundamentals for Developing Client Applications

Connecting to a Cluster

This white paper assumes you already completed the Confluent Cloud Quick Start, which walks through the required steps to obtain:

1. a Confluent Cloud user account: email address and password that logs you into the Confluent Cloud UI
2. access to a Confluent Cloud cluster: identification of the broker's endpoint via the Confluent Cloud UI or the Confluent Cloud CLI command `ccloud kafka cluster describe` (if you don't have a cluster yet, follow the quick start above to create one)
3. credentials to the cluster: a valid API key and secret for the user or service account (see the section Security for more details)

As a next step, configure your client application to connect to your Confluent Cloud cluster using the following three parameters:

1. `<BROKER ENDPOINT>`: bootstrap URL for the cluster
2. `<API KEY>`: API key for the user or service account
3. `<API SECRET>`: API secret for the user or service account

You can either define these parameters directly in the application code, or initialize a properties file and pass that file to your application. The latter is preferred in case the connection information changes, in which case you don't have to modify the code, only the properties file.

On the host with your client application, initialize a properties file with configuration to your Confluent Cloud cluster. In both examples below, you would substitute `<BROKER`

ENDPOINT>, <API KEY>, and <API SECRET> to match your cluster and credentials.

If your client is Java, create a file called $HOME/.ccloud/client.java.config that looks like this:

```
bootstrap.servers=<BROKER ENDPOINT>
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModu
le required \
    username\="<API KEY>" password\="<API SECRET>";
ssl.endpoint.identification.algorithm=https
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
```

If your client is based on one of the librdkafka bindings, create a file called $HOME/.ccloud/client.librdkafka.config that looks like this:

```
bootstrap.servers=<BROKER ENDPOINT>
sasl.username=<API KEY>
sasl.password=<API SECRET>
```

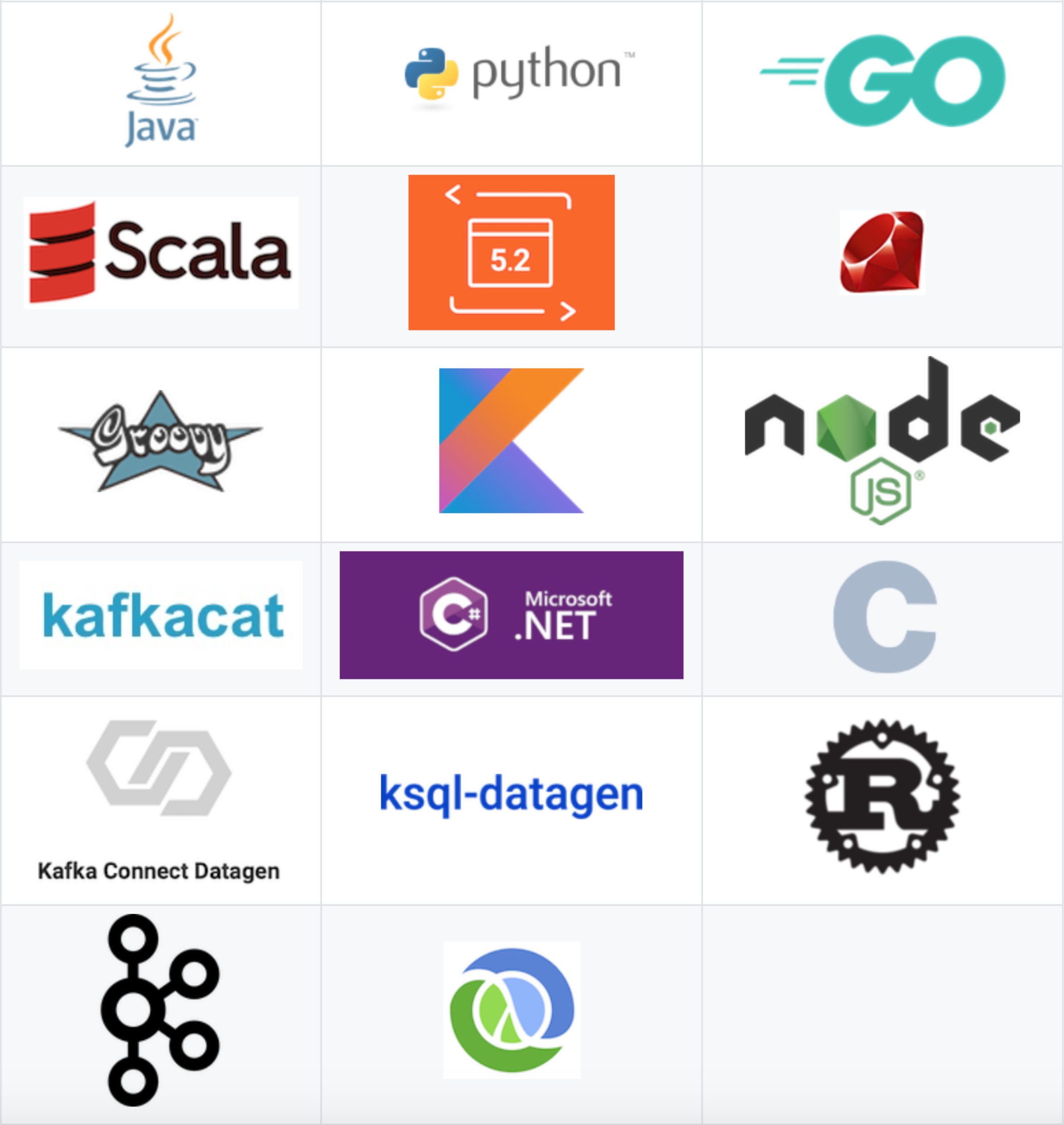
If your system/distribution does not provide root CA certificates in a standard location, you may need to also provide that path with ssl.ca.location. For example:

```
ssl.ca.location=/usr/local/etc/openssl/cert.pem
```

Compatible Programming Languages

You can write Kafka client applications to connect to Confluent Cloud in pretty much any language of your choosing. The clients just need to be configured using the Confluent Cloud cluster information and credentials.

This list of GitHub examples represents many of the languages that are supported for client code, written in the following programming languages and tools: C, Clojure, C#, Golang, Apache Groovy, Java, Kotlin, Node.js, Python, Ruby, Rust, and Scala. These Hello World examples produce to and consume from Confluent Cloud, and for the subset of languages that support it, there are additional examples using Confluent Cloud Schema Registry and Apache Avro™.



Schema Registry and Avro

Confluent Schema Registry provides a serving layer for your metadata with a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas based on a specified subject name strategy, provides multiple compatibility settings, and allows schemas to evolve according to the configured compatibility settings and expanded Avro support. Schema Registry provides serializers that plug into Kafka clients, which handle schema storage and retrieval for Kafka messages that are sent in the Avro format. For all these reasons, we recommend that applications use Confluent Schema Registry and Avro. We've seen many customers make operational mistakes when self-managing their own Schema Registry (e.g., bad designs, inconsistent configurations, and operational faux pas)—instead, you can leverage the Confluent Cloud Schema Registry from the start. Confluent Cloud Schema Registry is a fully managed service, and all you have to do is enable it in your Confluent Cloud environment and then configure your applications to use Avro serialization/deserialization.

```
basic.auth.credentials.source=USER_INFO
schema.registry.basic.auth.user.info=<SR API KEY>: <SR API SECRET>
schema.registry.url=<SR ENDPOINT>
```

Topic Management

There are at least three important things to remember about your topics in Confluent Cloud.

First, auto topic creation is completely disabled so that you are always in control of topic creation. This means that you must first create the user topics in Confluent Cloud before an application writes to or reads from them. You can create these topics in the Confluent Cloud UI, in the Confluent Cloud CLI, or using the `AdminClient` functionality directly from within the application. There are also Kafka topics used internally by ksqlDB and Kafka Streams, and these topics will be automatically created. Although auto topic creation is disabled in Confluent Cloud, ksqlDB and Kafka Streams leverage the `AdminClient` to programmatically create their internal topics. The developer does not need to explicitly create them.

Second, the most important feature that enables durability is replication, which ensures that messages are copied to multiple brokers. If one broker were to fail, the data would still be available from at least one other broker. Therefore, Confluent Cloud enforces a replication factor of `3` to ensure data durability. Durability is important not just for user-defined topics, but also for Kafka-internal topics. For example, a Kafka Streams application creates changelog topics for state stores and repartition topics for its internal use. Their configuration setting `replication.factor` is configured to `1` by default, so in your application, you should increase it to `3`.

Finally, the application must be authorized to access Confluent Cloud. See the section Security for more information.

Security

This section touches on two elements of security: access control lists (ACLs) and end-to-end encryption. There are certainly many more aspects to security that you need to consider, including organization best practices, delineated roles and responsibilities, data architecture that abides by enterprise security concerns, legal protections for personal data, etc. Refer to Confluent Cloud Security Addendum and Data Processing Addendum for Customers for more information.

Authentication and Authorization

Confluent Cloud provides features for managing access to services. Typically, credentials and access to the services are managed by an administrator or another role within the organization, so if you are a developer, work with your administrator to get the appropriate access. The administrator can use the Confluent Cloud CLI to manage access, credentials, and ACLs.

There are two types of accounts that can connect to Confluent Cloud:

1. User account: can access the Confluent Cloud services and can log into the Confluent Cloud UI
2. Service account: can access only the Confluent Cloud services

Both user accounts and service accounts can access Confluent Cloud services, but the user account is considered a "super user" while the service account is not. As a result, if the user account has a key/secret pair, those credentials by default will work on the cluster, for example to produce/consume to a topic by virtue of being a "super user." In contrast, service accounts by default do not work on the cluster until you configure ACLs to permit specific actions.

If your client application does not provide proper credentials or if the ACL is not created, the application will fail. For example, a Java application may throw an exception such as `org.apache.kafka.common.errors.TopicAuthorizationException`. Once the application is configured with proper credentials and an administrator creates the ACL that permits the application to write to or read from a topic, it will work. See this GitHub example for using ACLs with service accounts and client applications for Confluent Cloud.

End-to-End Encryption

When an application sends data to Confluent Cloud, the data is encrypted in motion and at rest.

Encryption in motion encrypts data sent from your clients over the internet/VPC to the Kafka infrastructure. This leverages TLS 1.2 and 256 bit keys, so your clients should run code that supports this version. If you have a Java client, both Java 8 and Java 11 are supported, but you should ideally run Java 11 to leverage TLS performance improvements that were introduced in Java 9.

Encryption at rest means that data stored in Confluent Cloud's infrastructure is encrypted. Even if someone gained physical access to the storage, the data would be unreadable because it is encrypted by AES 128-bit encryption.

However, some customers may need end-to-end encryption based on the nature of the data or their own internal policies. Read the End-to-End Encryption with Confluent Cloud white paper for the recommended approach on designing end-to-end encryption into Kafka clients (or in other words, encryption both in motion and at rest). It is intended for engineering design teams who seek to implement end-to-end message payload encryption on Confluent Cloud. The goal is for the customer to maintain confidentiality of data processed and stored by Kafka and for the custodians accessing this data to be the only ones with access to the encryption keys.

Networking

When deploying client applications, it is possible to both run them on prem and connect to Confluent Cloud services as long as there is network connectivity between the two. However, to reduce latency and improve application performance by sidestepping potential WAN network instability, the best practice is to deploy the application in the same cloud provider region as your Confluent Cloud cluster. This also avoids potential cost incurred from moving data between regions.

In addition, a developer needs to be aware of extra considerations regarding connectivity:

1. Networks are virtualized
2. Resources are often elastic, so IP addresses may change

If deploying your application in a cloud provider, you may need to consider virtualized networks and how connecting a client's network to Confluent Cloud's network may require specific configurations. See Confluent Cloud Networking for further details on interconnecting these networks.

You also need to anticipate that broker IP addresses may change. The client's JVM may cache the mapping between hostname and IP address, but if a broker in your Confluent Cloud cluster changes IP address, your client may keep resolving the hostname to the old IP address. To improve how the JVM resolves hostnames to IP addresses, configure two parameters:

1. `networkaddress.cache.ttl`: indicates the caching policy for successful name lookups. By default, this is `-1` which means cache forever. Instead, set this to 30 seconds.
2. `networkaddress.cache.negative.ttl`: indicates the caching policy for unsuccessful name lookups from the name service. By default, this is 10 seconds. Instead, set this to `0` which indicates never cache.

They can be set either as JVM arguments or in the application code, as shown below:

```
java.security.Security.setProperty("networkaddress.cache.ttl" , "30");
java.security.Security.setProperty("networkaddress.cache.negative.ttl"
, "0");
```

Finally, configure how the client handles failures if it is configured to multiple broker IP addresses. The domain name service (DNS) maps hostnames to IP addresses, and when a Kafka client does a DNS lookup on a broker's hostname, DNS may return multiple IP addresses. If the first IP address fails to connect, then by default the client will fail the connection. Instead, it is desirable that the client attempts to connect to all IP addresses before failing the connection. Configure this with `client.dns.lookup=use_all_dns_ips`. The prefix for the configuration parameter will vary by the type of client application and needs to be set independently for producers and consumers. Additionally, in a Kafka Streams application you may need to configure:

```
streamsConfigs.put(StreamsConfig.producerPrefix("client.dns.lookup"),
"use_all_dns_ips");
streamsConfigs.put(StreamsConfig.consumerPrefix("client.dns.lookup"),
"use_all_dns_ips");
```

Monitoring

Before going into production, make sure a robust monitoring system is in place for all of the producers, consumers, topics, and any other Kafka or Confluent components you are using. Performance monitoring provides a quantitative account of how each component is doing. This monitoring is important once you're in production, because production environments are dynamic: data profiles may change, you may add new clients, and you may enable new features. Ongoing monitoring is as much about identifying and responding to potential failures as it is about ensuring that the services goals are consistently met even as the production environment changes.

Metrics API

The Confluent Cloud Metrics API provides programmatic access to actionable operational metrics for your Confluent Cloud deployment. You can get server-side metrics for the Confluent managed services, but you cannot get client-side metrics via the Metrics API (see the sections Producers and Consumers for client-side metrics). These metrics are enabled by default, and any authorized user can get self-serve access to them. They are aggregated at the topic level and cluster level, which is very useful for monitoring overall usage and performance, particularly since these relate to customer billing. We recommend using the Metrics API to query metrics at these granularities, but other resolutions are available if needed:

- Bytes produced per minute grouped by topic
- Bytes consumed per minute grouped by topic
- Max retained bytes per hour over two hours for a given topic
- Max retained bytes per hour over two hours for a given cluster

As an example of metrics at the topic level, here is the number of bytes produced per minute for a topic called `test-topic`:

```
{
    "data": [
        {
            "timestamp": "2019-12-19T16:01:00Z",
            "metric.label.topic": "test-topic",
            "value": 203.0
        },
        {
            "timestamp": "2019-12-19T16:02:00Z",
            "metric.label.topic": "test-topic",
            "value": 157.0
        },
        {
            "timestamp": "2019-12-19T16:03:00Z",
            "metric.label.topic": "test-topic",
            "value": 371.0
        },
        {
            "timestamp": "2019-12-19T16:04:00Z",
            "metric.label.topic": "test-topic",
            "value": 288.0
        }
    ]
}
```

As an example of metrics at the cluster level, here is the max retained bytes per hour over two hours for a Confluent Cloud cluster:

```
{
    "data": [
        {
            "timestamp": "2019-12-19T16:00:00Z",
            "value": 507350.0
        },
        {
            "timestamp": "2019-12-19T17:00:00Z",
            "value": 507350.0
        }
    ]
}
```

You can pull these metrics easily over the public internet using HTTPS, capturing them at regular intervals to get a time series, operational view of cluster performance. You can integrate these into any cloud provider monitoring tools like Azure Monitor, Google Cloud's Stackdriver, or Amazon CloudWatch, or into existing monitoring systems like Prometheus, Datadog, etc., and plot them in a time series graph to see usage over time. When writing your own application to use the Metrics API, consult the full API specification to leverage advanced features.

Client JMX Metrics

Kafka applications expose some internal JMX (Java Management Extensions) metrics, and many users run JMX exporters to feed these metrics into their monitoring systems. You can get these JMX metrics for your client applications and the services that you manage, though not for the Confluent-managed services, which are not directly exposed to users. To get these JMX client metrics, start the Kafka client applications with the `JMX_PORT` environment variable configured. There are many Kafka-internal metrics that are exposed through JMX to provide insight into the performance of your applications.

Producers

Throttling

Depending on your Confluent Cloud service plan, you are limited to certain throughput rates for produce (write). If your client applications exceed these produce rates, the quotas on the brokers will detect it and the client application requests will be throttled by the brokers. It's important to ensure your producers are well behaved. If they are being throttled, consider two options. The first option is to make modifications to the application to optimize its throughput, if possible (read the section Optimizing for Throughput for more details). The second option is to upgrade to a cluster configuration with higher limits. In Confluent Cloud, you can choose from Standard and Dedicated clusters, and Dedicated clusters are customizable for higher limits. The Metrics API can give you some indication of throughput from the server side, but it doesn't provide throughput metrics on the client side. To get throttling metrics per producer, monitor the following client JMX metrics:

Metric	Description
`kafka.producer:type=producer-metrics,client-id=([-.w]+),name=produce-throttle-time-avg`	The average time in ms that a request was throttled by a broker
`kafka.producer:type=producer-metrics,client-id=([-.w]+),name=produce-throttle-time-max`	The maximum time in ms that a request was throttled by a broker

User Processes

To further tune the performance of your producer, monitor the producer time spent in user processes if the producer has non-blocking code to send messages. Using the `io-ratio` and `io-wait-ratio` metrics described below, user processing time is the fraction of time not spent in either of these. If time in these are low, then the user processing time may be high which keeps the single producer I/O thread busy. For example, you can check if the producer is using any callbacks, which are invoked when messages have been acknowledged and run in the I/O thread:

Metric	Description
`kafka.producer:type=producer-metrics,client-id=([-.w]+),name=io-ratio`	Fraction of time that the I/O thread spent doing I/O
`kafka.producer:type=producer-metrics,client-id=([-.w]+),name=io-wait-ratio`	Fraction of time that the I/O thread spent waiting

Consumers

Throttling

Depending on your Confluent Cloud service plan, you are limited to certain throughput rates for consume (read). If your client applications exceed these consume rates, the quotas on the brokers will detect it and the brokers will throttle the client application requests. It's important to ensure your consumers are well behaved, and if they are being throttled, consider two options. The first option is to make modifications to the application to optimize its throughput, if possible (read the section Optimizing for Throughput for more details). The second option is to upgrade to a cluster configuration with higher limits. In Confluent Cloud, you can choose from Standard and Dedicated clusters, and Dedicated clusters are customizable for higher limits. The Metrics API can give you some indication of throughput from the server side, but it doesn't provide throughput metrics on the client side. To get throttling metrics per consumer, monitor the following client JMX metrics:

Metric	Description
`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),name=fetch-throttle-time-avg`	The average time in ms that a broker spent throttling a fetch request
`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),name=fetch-throttle-time-max`	The maximum time in ms that a broker spent throttling a fetch request

Consumer Lag

Additionally, it is important to monitor your application's `consumer lag`, which is the number of records for any partition that the consumer is behind in the log. This metric is particularly important for real-time consumer applications where the consumer should be processing the newest messages with as low latency as possible. Monitoring consumer lag can indicate whether the consumer is able to fetch records fast enough from the brokers. Also consider how the offsets are committed. For example, exactly once semantics (EOS) provides stronger guarantees while potentially increasing consumer lag. You can monitor consumer lag from the Confluent Cloud UI, as described in the documentation. Alternatively, if you are capturing JMX metrics, you can monitor `records-lag-max`:

Metric	Description
`kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-.w]+),records-lag-max`	The maximum lag in terms of number of records for any partition in this window. An increasing value over time is your best indication that the consumer group is not keeping up with the producers.

Optimizations and Tuning

You can do unit testing, integration testing, and Avro and schema compatibility testing of your application in a CI/CD pipeline, all locally, or in a test environment in Confluent Cloud. The blog post [Testing Your Streaming Application]() describes how to leverage utilities to simulate parts of the Kafka services, e.g., `MockProducer`, `MockConsumer`, `TopologyTestDriver`, `MockProcessorContext`, `EmbeddedKafkaCluster`, and `MockSchemaRegistryClient`, among others. Once your application is up and running to Confluent Cloud, verify all the functional pieces of the architecture work and check the dataflows end to end.

After you have done the functional validation, you may proceed to making optimizations to tune performance. The following sections describe guidelines for benchmarking and how to optimize your applications depending on your service goals.

Benchmarking

Benchmark testing is important because there is no one-size-fits-all recommendation for the configuration parameters discussed above. Proper configuration always depends on the use case, other features you have enabled, the data profile, etc. If you are tuning Kafka clients beyond the defaults, we generally recommend running benchmark tests. Regardless of your service goals, you should understand what the performance profile of your application is—it is especially important when you want to optimize for throughput or latency. Your benchmark tests can also feed into the calculations for determining the correct number of partitions and the number of producer and consumer processes.

Start by measuring your bandwidth using the Kafka utilities `kafka-producer-perf-test` and `kafka-consumer-perf-test`. This provides a baseline performance to your Confluent Cloud instance, taking application logic out of the equation.

Then benchmark your client application, starting with the default Kafka configuration parameters, and familiarize yourself with the default values. Determine the baseline input performance profile for a given producer by removing dependencies on anything upstream from the producer. Rather than receiving data from upstream sources, modify your producer to generate its own mock data at very high output rates, such that the data generation is not a bottleneck.

If you are testing with compression, be aware of how the mock data is generated. Sometimes mock data is unrealistic, containing repeated substrings or being padded with zeros, which may result in a better compression performance than what would be seen in production. Ensure that the mock data reflects the type of data used in production in order to get results that more accurately reflect performance in production. Or, instead of using mock data, consider using copies of production data or `cleansed` production data in your benchmarking.

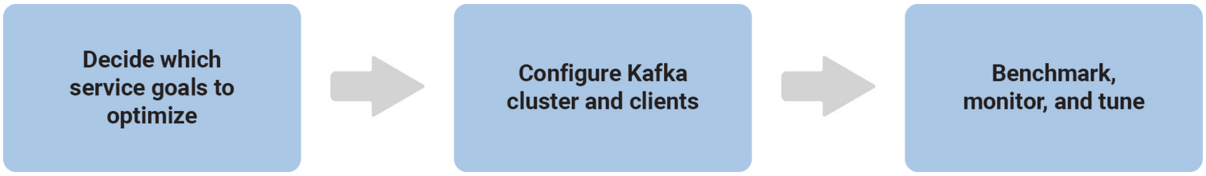
Run a single producer client on a single server. Measure the resulting throughput using the available JMX metrics for the Kafka producer. Repeat the producer benchmarking test, increasing the number of producer processes on the server in each iteration to determine the number of producer processes per server to achieve the highest throughput. You can determine the baseline output performance profile for a given consumer in a similar way. Run a single consumer client on a single server. Repeat this test, increasing the number of consumer processes on the server in each iteration to determine the number of consumer processes per server to achieve the highest throughput.

Then you can run a benchmark test for different permutations of configuration parameters that reflect your service goals. The following sections describe how different configuration parameters impact your application performance and how you can tune them accordingly. Focus on those configuration parameters, and avoid the temptation to discover and change other parameters from their default values without understanding exactly how they impact the entire system. Tune the settings on each iteration, run a test, observe the results, tune again, and so on, until you identify settings that work for your throughput and latency requirements. Refer to this blog post when considering partition count in your benchmark tests.

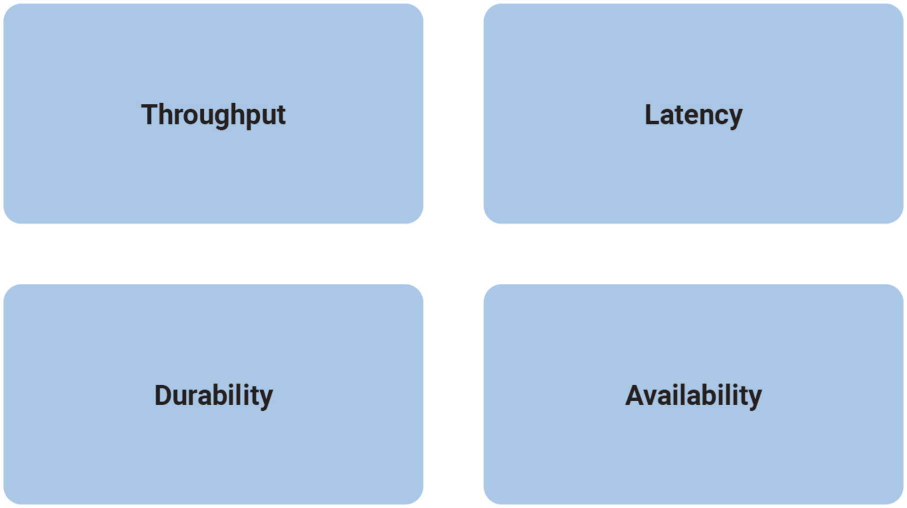
Service Goals

The fact that you can get your Kafka client application up and running to Confluent Cloud within seconds does not mean you don't want to do some tuning before you go into production. The reason to do some tuning is that different use cases will have different sets of requirements that will drive different service goals. To optimize for those service goals, there are Kafka configuration parameters that you should change in your application. In fact, Kafka's design inherently provides configuration flexibility to users. To make sure your Kafka deployment is optimized for your service goals, you absolutely should tune the settings of some of your Kafka client configuration parameters and benchmark in your own environment. We strongly recommend benchmarking your application before you go to production.

This more advanced section is about how to identify your service goals, configure your Kafka deployment to optimize for them, and ensure that you are achieving them through monitoring.


Decide which service goals to optimize
Configure Kafka cluster and clients
Benchmark, monitor, and tune

The first step is to decide which service goals you want to optimize. We'll consider four goals that often involve trade-offs with one another: throughput, latency, durability, and availability. To figure out which goals you want to optimize, recall the use cases your Kafka applications are going to serve. Think about the applications, the business requirements—the things that absolutely cannot fail for that use case to be satisfied. Think about how Kafka as an event streaming technology fits into the pipeline of your business.



Sometimes the question of which service goal to optimize is hard to answer, but it is still extremely critical to discuss the original business use cases and what the main goals are with your team.

The main reason is that you can't maximize all goals at the same time. There are occasionally trade-offs between throughput, latency, durability, and availability, which this white paper will cover in detail. You may be familiar with the common trade-off in performance between throughput and latency and perhaps between durability and availability as well. As you consider the whole system, you will find that you cannot think about any of them in isolation, which is why this paper looks at all four service goals together. This does not mean that optimizing one of these goals results in completely losing out on the others. It just means that they are all interconnected, and thus you can't maximize all of them at the same time.

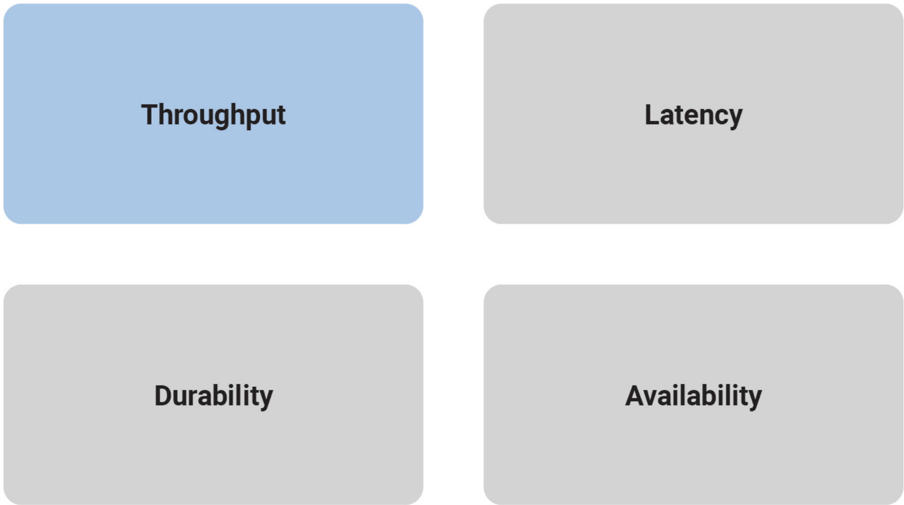
Another reason it is important to identify which service goal you want to optimize is so you can tune your Kafka configuration parameters to achieve it. You need to understand what your users expect from the system to ensure you are optimizing Kafka to meet their needs.

- Do you want to optimize for *high throughput*, which is the rate that data is moved from producers to brokers or brokers to consumers? Some use cases have millions of writes per second. Because of Kafka's design, writing large volumes of data into it is not a hard thing to do. It's faster than trying to push volumes of data through a traditional database or key-value store, and it can be done with modest hardware.

- Do you want to optimize for *low latency*, which is the elapsed time moving messages end to end (from producers to brokers to consumers)? One example of a low-latency use case is a chat application, where the recipient of a message needs to get the message with as little latency as possible. Other examples include interactive websites where users follow posts from friends in their network, or real-time stream processing for the Internet of Things (IoT).
- Do you want to optimize for *high durability*, which guarantees that committed messages will not be lost? One example use case for high durability is an event streaming microservices pipeline using Kafka as the event store. Another is for integration between an event streaming source and some permanent storage (e.g., Amazon S3) for mission-critical business content.
- Do you want to optimize for *high availability*, which minimizes downtime in case of unexpected failures? Kafka is a distributed system, and it is designed to tolerate failures. In use cases demanding high availability, it's important to configure Kafka such that it will recover from failures as quickly as possible.

One caution before we jump into how to optimize Kafka for different service goals: the values for some of the configuration parameters discussed in this paper depend on other factors, such as average message size, number of partitions, etc. These can greatly vary from environment to environment. For some configuration parameters, we provide a range of reasonable values, but recall that benchmarking is always crucial to validate the settings for your specific deployment.

Optimizing for Throughput



To optimize for throughput, the producers and consumers need to move as much data as they can within a given amount of time. For high throughput, you are trying to maximize the rate at which this data moves. This data rate should be as fast as possible. A topic partition is the unit of parallelism in Kafka. Messages to different partitions can be sent in parallel by producers, written in parallel by different brokers, and read in parallel by different consumers. In general, a higher number of topic partitions results in higher throughput, and to maximize throughput, you want enough partitions to distribute them across the brokers in your Confluent Cloud cluster.

Although it might seem tempting just to create topics with a very large number of partitions, there are trade-offs to increasing the number of partitions. Review our guidelines for how to choose the number of partitions. Be sure to choose the partition count carefully based on producer throughput and consumer throughput, and benchmark performance in your environment. Also take into consideration the design of your data patterns and key assignments so that messages are distributed as evenly as possible across topic partitions. This will prevent overloading certain topic partitions relative to others.

Next, let's discuss the batching strategy of Kafka producers. Producers can batch messages going to the same partition, which means they collect multiple messages to send together in a single request. The most important step you can take to optimize throughput is to tune the producer batching to increase the batch size and the time spent waiting for the batch to fill up with messages. Larger batch sizes result in fewer requests to Confluent Cloud, which reduces load on producers as well as the broker CPU overhead to process each request. With the Java client, you can configure the `batch.size` parameter to increase the maximum size in bytes of each message batch. To give more time for batches to fill, you can configure the `linger.ms` parameter to have the producer wait longer before sending. The delay allows the producer to wait for the batch to reach the configured `batch.size`. The trade-off is tolerating higher latency, since messages are not sent as soon as they are ready to send.

You can also easily enable compression, which means a lot of bits can be sent as fewer bits. Enable compression by configuring the `compression.type` parameter, which can be set to one of the following standard compression codecs: `lz4`, `snappy`, `zstd`, and `gzip`. For performance, we generally recommend using `lz4`. We strongly recommend not using `gzip` because it's much more compute intensive relative to the other codecs, so your application may not perform as well. Compression is applied on full batches of data, so better batching results in better compression ratios. When Confluent Cloud receives a compressed batch of messages from a producer, it always decompresses the data in order to validate it. Afterwards, it considers the compression codec of the destination topic.

- If the compression codec of the destination topic are left at the default setting of `producer`, or if the codecs of the batch and destination topic are the same, Confluent Cloud takes the compressed batch from the client and writes it directly to the topic's log file without taking cycles to recompress the data
- Otherwise, Confluent Cloud needs to recompress the data to match the codec of the destination topic, and this can result in a performance impact; therefore, keep the compression codecs the same if possible

When a producer sends a message to Confluent Cloud, the message is sent to the leader broker for the target partition. Then the producer awaits a response from the leader broker (assuming `acks` is not set to `0`, in which case the producer will not wait for any acknowledgment from the broker at all) to know that its message has been committed before proceeding to send the next messages. There are automatic checks in place to make sure consumers cannot read messages that haven't been committed yet. When leader brokers send those responses, it may impact the producer throughput: the sooner a producer receives a response, the sooner the producer can send the next message, which generally results in higher throughput. So producers can set the configuration parameter `acks` to specify the number of acknowledgments the leader broker must have received before responding to the producer with an acknowledgment. Setting `acks=1` makes the leader broker write the record to its local log and then acknowledge the request without awaiting acknowledgment from all followers. The trade-off is you have to tolerate lower durability, because the producer does not have to wait until the message is replicated to other brokers.

Kafka producers automatically allocate memory for the Java client to store unsent messages. If that memory limit is reached, then the producer will block on additional sends until memory frees up or until `max.block.ms` time passes. You can adjust how much memory is allocated with the configuration parameter `buffer.memory`. If you don't have a lot of partitions, you may not need to adjust this at all. However, if you have a lot of partitions, you can tune `buffer.memory`—while also taking into account the message size, linger time, and partition count—to maintain pipelines across more partitions. This in turn enables better utilization of the bandwidth across more brokers.

Likewise, you can tune consumers for higher throughput by adjusting how much data they get from each fetch from the leader broker in Confluent Cloud. You can increase how much data the consumers get from the leader for each fetch request by increasing the configuration parameter `fetch.min.bytes`. This parameter sets the minimum number of bytes expected for a fetch response from a consumer. Increasing this will also reduce the number of fetch requests made to Confluent Cloud, reducing the broker CPU overhead to process each fetch, thereby also improving throughput. Similar to the consequence of increasing batching on the producer, there may be a resulting tradeoff to higher latency when increasing this parameter on the consumer. This is because the broker won't send the consumer new messages until the fetch request has enough messages to fulfill the size of the fetch request, i.e., `fetch.min.bytes`, or until the expiration of the wait time, i.e., configuration parameter `fetch.max.wait.ms`.

Assuming the application allows it, use consumer groups with multiple consumers to parallelize consumption. Parallelizing consumption may improve throughput because multiple consumers can balance the load, processing multiple partitions simultaneously. The upper limit on this parallelization is the number of partitions in the topic.

Summary of Configurations for Optimizing Throughput

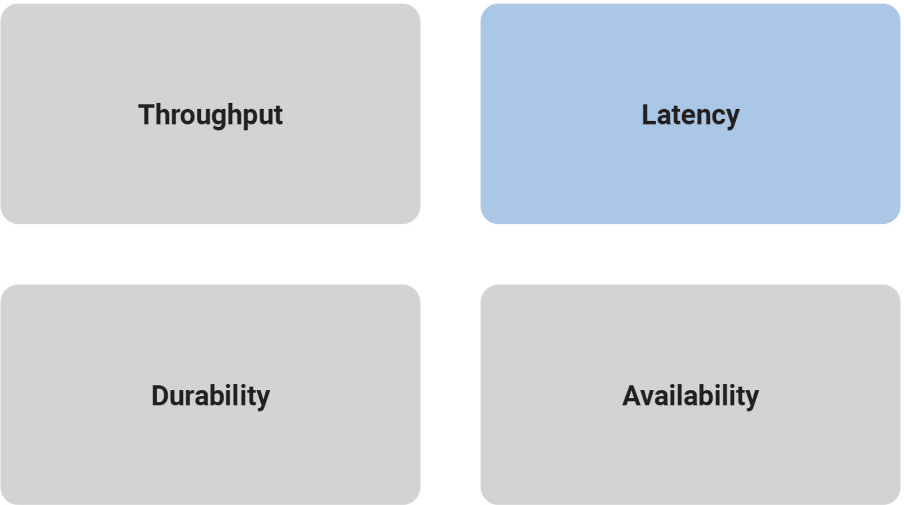
Producer:

- `batch.size`: increase to 100000–200000 (default 16384)
- `linger.ms`: increase to 10–100 (default 0)
- `compression.type=lz4` (default `none`, i.e., no compression)
- `acks=1` (default 1)
- `buffer.memory`: increase if there are a lot of partitions (default 33554432)

Consumer:

- `fetch.min.bytes`: increase to ~100000 (default 1)

Optimizing for Latency



Many of the Kafka configuration parameters discussed in the section on throughput have default settings that optimize for latency. Thus, those configuration parameters generally don't need to be adjusted, but we will review the key parameters to understand how they work.

Confluent has guidelines on how to choose the number of partitions. Because a partition is a unit of parallelism in Kafka, an increased number of partitions may increase throughput. However, there is a trade-off in that an increased number of partitions may also increase latency. It may take longer to replicate a lot of partitions shared between each pair of brokers and consequently take longer for messages to be considered committed. No message can be consumed until it is committed, so this can ultimately increase end-to-end latency.

Producers automatically batch messages, which means they collect messages to send together. The less time that is given waiting for those batches to fill, then generally there is less latency producing data to Confluent Cloud. By default, the producer is tuned for low latency and the configuration parameter `linger.ms` is set to 0, which means the producer will send as soon as it has data to send. In this case, it is not true that batching is disabled—messages are always sent in batches—but sometimes a batch may have only one message (unless messages are passed to the producer faster than it can send them).

Consider whether you need to enable compression. Enabling compression typically requires more CPU cycles to do the compression, but it reduces network bandwidth utilization. So disabling compression typically spares the CPU cycles but increases network bandwidth utilization. Depending on the compression performance, you may consider leaving compression disabled with `compression.type=none` to spare the CPU cycles, although a good compression codec may potentially reduce latency as well.

You can tune the number of acknowledgments the producer requires the leader broker in the Confluent Cloud cluster to have received before considering a request complete. (Note that this acknowledgment to the producer differs from when a message is considered committed—more on that in the next section.) The sooner the leader broker responds, the sooner the producer can continue sending the next batch of messages, thereby generally reducing producer latency. Set the number of required acknowledgments with the producer `acks` configuration parameter. By default, `acks=1`, which means the leader broker will respond sooner to the producer before all replicas have received the message. Depending on your application requirements, you can even set `acks=0` so that the producer will not wait for a response for a producer request from the broker, but then messages can potentially be lost without the producer even knowing.

Similar to the batching concept on the producers, you can tune consumers for lower latency by adjusting how much data it gets from each fetch from the leader broker in Confluent Cloud. By default, the consumer configuration parameter `fetch.min.bytes` is set to `1`, which 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, i.e., the configuration parameter `fetch.max.wait.ms`. Looking at these two configuration parameters together lets you reason through the size of the fetch request, i.e., `fetch.min.bytes`, or the age of a fetch request, i.e., `fetch.max.wait.ms`.

If you have a Kafka event streaming application or are using ksqlDB, there are also some performance enhancements you can make within the application. For scenarios where you need to perform table lookups at very large scale and with a low processing latency, you can use local stream processing. A popular pattern is to use Kafka Connect to make remote databases available local to Kafka. Then you can leverage the Kafka Streams API or ksqlDB to perform very fast and efficient local joins of such tables and streams, rather than requiring the application to make a query to a remote database over the network for each record. You can track the latest state of each table in a local state store, thus greatly reducing the processing latency as well as reducing the load of the remote databases when doing such streaming joins.

Kafka Streams applications are founded on processor topologies, a graph of stream processor nodes that can act on partitioned data for parallel processing. Depending on the application, there may be conservative but unnecessary data shuffling based on repartition topics, which would not result in any correctness issues but can introduce performance penalties. To avoid performance penalties, you may enable topology optimizations for your event streaming applications by setting the configuration parameter `topology.optimization`. Enabling topology optimizations may reduce the amount of reshuffled streams that are stored and piped via repartition topics.

Summary of Configurations for Optimizing Latency

Producer:

- `linger.ms=0` (default 0)
- `compression.type=none` (default `none`, i.e., no compression)
- `acks=1` (default 1)

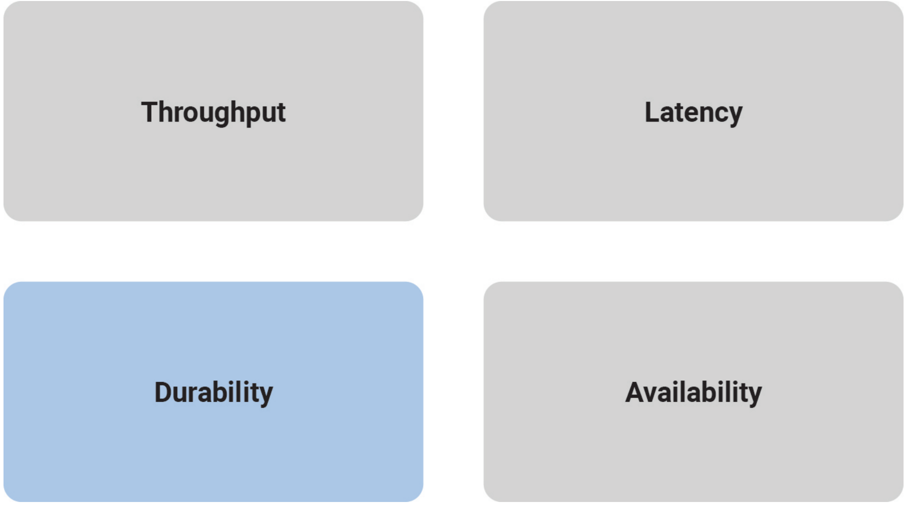
Consumer:

- `fetch.min.bytes=1` (default 1)

Streams:

- `StreamsConfig.TOPOLOGY_OPTIMIZATION`: `StreamsConfig.OPTIMIZE` (default `StreamsConfig.NO_OPTIMIZATION`)
- Streams applications have embedded producers and consumers, so also check those configuration recommendations

Optimizing for Durability



Durability is all about reducing the chance for a message to get lost. Confluent Cloud enforces a replication factor of `3` to ensure data durability.

Producers can control the durability of messages written to Kafka through the `acks` configuration parameter. This parameter was discussed in the context of throughput and latency optimization, but it is primarily used in the context of durability. To optimize for high durability, we recommend setting it to `acks=all` (equivalent to `acks=-1`), which means the leader will wait for the full set of in-sync replicas (ISRs) to acknowledge the message and to consider it committed. This provides the strongest available guarantees that the record will not be lost as long as at least one in-sync replica remains alive. The trade-off is tolerating a higher latency because the leader broker waits for acknowledgments from replicas before responding to the producer.

Producers can also increase durability by trying to resend messages if any sends fail to ensure that data is not lost. The producer automatically tries to resend messages up to the number of times specified by the configuration parameter `retries` (default `MAX_INT`) and up to the time duration specified by the configuration parameter `delivery.timeout.ms` (default 120000), the latter of which was introduced in KIP-91. You can tune `delivery.timeout.ms` to the desired upper bound for the total time between sending a message and receiving an acknowledgment from the broker, which should reflect business requirements of how long a message is valid for.

There are two things to take into consideration with these automatic producer retries:

duplication and message ordering.

1. *Duplication*: if there are transient failures in Confluent Cloud that cause a producer retry, the producer may send duplicate messages to Confluent Cloud
2. *Ordering*: multiple send attempts may be "in flight" at the same time, and a retry of a previously failed message send may occur after a newer message send succeeded

To address both of these, we generally recommend that you configure the producer for idempotency, i.e., `enable.idempotence=true`, for which the brokers in Confluent Cloud track messages using incrementing sequence numbers, similar to TCP. Idempotent producers can handle duplicate messages and preserve message order even with request pipelining—there is no message duplication because the broker ignores duplicate sequence numbers, and message ordering is preserved because when there are failures, the producer temporarily constrains to a single message in flight until sequencing is restored. In case the idempotence guarantees can't be satisfied, the producer will raise a fatal error and reject any further sends, so when configuring the producer for idempotency, the application developer needs to catch the fatal error and handle it appropriately.

However, if you do not configure the producer for idempotency but the business requirements call for it, you need to address the potential for message duplication and ordering issues in other ways. To handle possible message duplication if there are transient failures in Confluent Cloud, be sure to build your consumer application logic to process duplicate messages. To preserve message order while also allowing resending failed messages, set the configuration parameter `max.in.flight.requests.per.connection=1` to ensure that only one request can be sent to the broker at a time. To preserve message order while allowing request pipelining, set the configuration parameter `retries=0` if the application is able to tolerate some message loss.

Instead of letting the producer automatically retry sending failed messages, you may prefer to manually code the actions for exceptions returned to the producer client, e.g., the `onCompletion()` method in the `Callback` interface in the Java client. If you want manual retry handling, disable automatic retries by setting `retries=0`. Note that producer idempotency tracks message sequence numbers, which makes sense only when automatic retries are enabled. Otherwise, if you set `retries=0` and the application manually tries to resend a failed message, then it just generates a new sequence number so the duplication detection won't work. Disabling automatic retries can result in message gaps due to individual send failures, but the broker will preserve the order of writes it receives.

Confluent Cloud provides durability by replicating data across multiple brokers. Each partition will have a list of assigned replicas (i.e., brokers) that should have copies the data. The list of replicas that are caught up to the leader are called in-sync replicas (ISRs). For each partition, leader brokers will automatically replicate messages to other brokers that are in their ISR list. When a producer sets `acks=all` (or `acks=-1`), then the configuration parameter `min.insync.replicas` specifies the minimum threshold for the replica count in the ISR list. If this minimum count cannot be met, then the producer will raise an exception. 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 `replication.factor=3`, topic configuration override `min.insync.replicas=2`, and producer `acks=all`, thereby ensuring that the producer raises an exception if a majority of replicas do not receive a write.

You also need to consider what happens to messages if there is an unexpected consumer failure to ensure that no messages are lost as they are being processed. Consumer offsets track which messages have already been consumed, so how and when consumers commit message offsets are crucial for durability. You want to avoid a situation where a consumer commits the offset of a message, starts processing that message, and then unexpectedly fails. This is because the subsequent consumer that starts reading from the same partition will not reprocess messages with offsets that have already been committed.

By default, offsets are configured to be automatically committed during the consumer's `poll()` call at a periodic interval, and this is typically good enough for most use cases. But if the consumer is part of a transactional chain and you need strong message delivery guarantees, you may want the offsets to be committed only after the consumer finishes completely processing the messages. You can configure whether these consumer commits happen automatically or manually with the configuration parameter `enable.auto.commit`. For extra durability, you may disable the automatic commit by setting `enable.auto.commit=false` and explicitly call one of the commit methods in the consumer code (e.g., `commitSync()` or `commitAsync()`).

For even stronger guarantees, you may configure your applications for EOS transactions, which enable atomic writes to multiple Kafka topics and partitions. Since some messages in the log may be in various states of a transaction, consumers can set the configuration parameter `isolation.level` to define the types of messages they should receive. By setting `isolation.level=read_committed`, consumers will receive only non-transactional messages or committed transactional messages, and they will not receive messages from open or aborted transactions. To use transactional semantics in a `consume-process-produce` pattern and ensure each message is processed exactly once, a client application should set `enable.auto.commit=false` and should not commit offsets manually, instead using the `sendOffsetsToTransaction()` method in the `KafkaProducer` interface. You may also enable exactly once for your event streaming applications by setting the configuration parameter `processing.guarantee`.

Summary of Configurations for Optimizing Durability

Producer:

- `replication.factor=3`
- `acks=all` (default 1)
- `enable.idempotence=true` (default false), to prevent duplicate messages and out-of-order messages
- `max.in.flight.requests.per.connection=1` (default 5), to prevent out of order messages when not using an idempotent producer

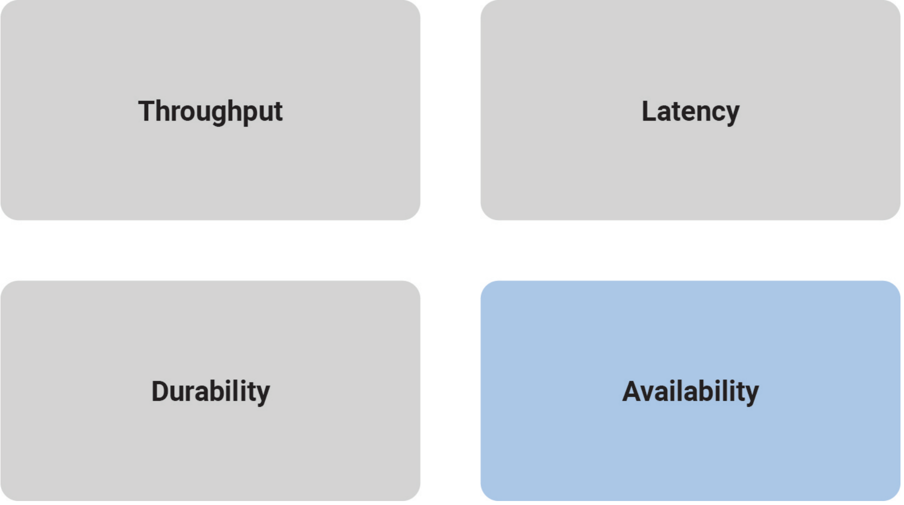
Consumer:

- `enable.auto.commit=false` (default true)
- `isolation.level=read_committed` (when using EOS transactions)

Streams:

- `StreamsConfig.REPLICATION_FACTOR_CONFIG`: 3 (default 1)
- `StreamsConfig.PROCESSING_GUARANTEE_CONFIG`: `StreamsConfig.EXACTLY_ONCE` (default `StreamsConfig.AT_LEAST_ONCE`)
- Streams applications have embedded producers and consumers, so also check those configuration recommendations

Optimizing for Availability



To optimize for high availability, you should tune your Kafka application to recover as quickly as possible from failure scenarios.

When a producer sets `acks=all` (or `acks=-1`), the configuration parameter `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. In the case of a shrinking ISR, the higher this minimum value is, the more likely there is to be a failure on producer send, which decreases availability for the partition. On the other hand, by setting this value low (e.g., `min.insync.replicas=1`), the system will tolerate more replica failures. As long as the minimum number of replicas is met, the producer requests will continue to succeed, which increases availability for the partition.

On the consumer side, consumers can share processing load by being a part of a consumer group. If a consumer unexpectedly fails, Kafka can detect the failure and rebalance the partitions amongst the remaining consumers in the consumer group. The consumer failures can be hard failures (e.g., `SIGKILL`) or soft failures (e.g., expired session timeouts), and they can be detected either when consumers fail to send heartbeats or when they fail to send `poll()` calls. The consumer liveness is maintained with a heartbeat, now in a background thread since KIP-62, and the timeout used to detect failed heartbeats is dictated by the configuration parameter `session.timeout.ms`. Increase the session timeout to take into account potential network delays and to avoid soft failures. Soft failures occur most commonly in two scenarios: when a batch of messages returned by `poll()` takes too long to process or when a JVM GC pause takes too long. If you have a `poll()` loop that spends too much time processing messages, you can address this either by increasing the upper bound on the amount of time that a consumer can be idle before fetching more records with `max.poll.interval.ms` or by reducing the maximum size of batches returned with the configuration parameter `max.poll.records`. Although higher session timeouts increase the time to detect and recover from a consumer failure, relatively speaking, incidents of failed clients are less likely than network issues.

Finally, when rebalancing workloads by moving tasks between event streaming application instances, you can reduce the time it takes to restore task processing state before the application instance resumes processing. In Kafka Streams, state restoration is usually done by replaying the corresponding changelog topic to reconstruct the state store. The application can replicate local state stores to minimize changelog-based restoration time, by setting the configuration parameter `num.standby.replicas`. Thus, when a stream task is initialized or reinitialized on the application instance, its state store is restored to the most recent snapshot accordingly:

- If a local state store does not exist, i.e., `num.standby.replicas=0`, then the changelog is replayed from the earliest offset.
- If a local state store does exist, i.e., `num.standby.replicas` is greater than 0, then the changelog is replayed from the previously checkpointed offset. This method takes less time because it is applying a smaller portion of the changelog.

Summary of Configurations for Optimizing Availability

Consumer:

- `session.timeout.ms`: increase (default 10000)

Streams:

- `StreamsConfig.NUM_STANDBY_REPLICAS_CONFIG`: 1 or more (default 0)
- Streams applications have embedded producers and consumers, so also check those configuration recommendations

Next Steps

Develop, monitor, and tune your Kafka application to Confluent Cloud. You may also run GitHub examples that demonstrate how you can build event streaming applications with fully managed services in Confluent Cloud.

Additional Resources

- Documentation
- Confluent Professional Services
- Confluent Training