Optimizing Your Apache Kafka Deployment

Levers for Throughput, Latency, Durability, and Availability

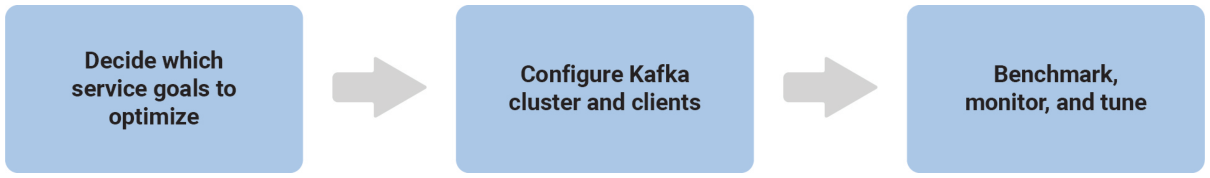
Table of Contents

Introduction

Apache Kafka® is the best open source event streaming technology that runs straight off the shelf. Just point your client applications at your Kafka cluster, and Kafka takes care of the rest: Load is automatically distributed across brokers, brokers automatically leverage zero-copy transfer to send data to consumers, 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 the cluster, and partition leadership is automatically reassigned upon broker failure. It's an operator's dream come true!

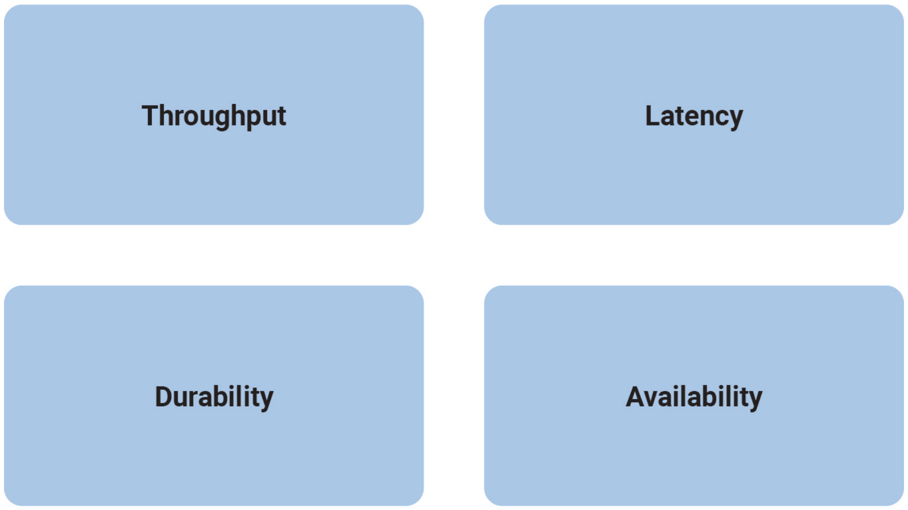
Without needing to make any changes to Kafka configuration parameters, you can set up a development Kafka environment and test basic functionality. Yet the fact that Kafka runs straight off the shelf does not mean you don't want to do some tuning before you go into production. The reason to tune is that 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 fact, the Kafka design itself provides configuration flexibility to users. To make sure your Kafka deployment is optimized for your service goals, you absolutely should investigate tuning the settings of some configuration parameters and benchmarking in your own environment. Ideally, you should do that before you go to production, or at least before you scale out to a larger cluster size.

This white paper is about how to identify those 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

Deciding Which Service Goals to Optimize

The first step is to decide which service goals you want to optimize. We'll consider four goals which often involve tradeoffs with one another: throughput, latency, durability, and availability. To figure out which goals you want to optimize, recall the use cases your cluster is 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 you have to force your team to discuss the original business use cases and what the main goals are. There are two reasons this discussion is important.

The first reason is that you can't maximize all goals at the same time. There are occasionally tradeoffs between throughput, latency, durability, and availability, which we cover in detail in this white paper. You may be familiar with the common tradeoff 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, so 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.

The second reason it is important to identify which service goal you want to optimize is that you can and should tune 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.

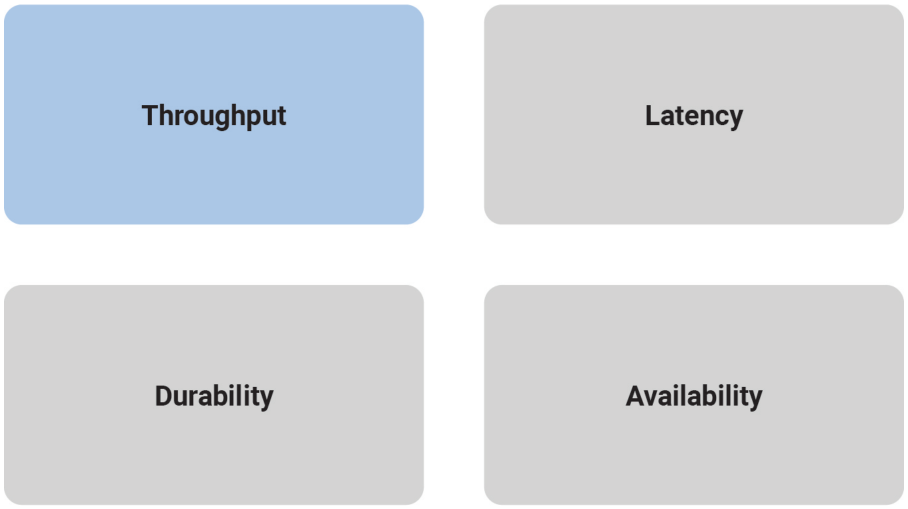
- Do you want to optimize for *high throughput*, which is the rate that data is moved from producers to brokers or brokers to consumers? Some use cases have millions of writes per second. Because of Kafka's design, writing large volumes of data into it is not a hard thing to do. It's faster than trying 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.
- Do you want to optimize for *high durability*, which guarantees that messages that have been committed 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., AWS 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.

If the service goals that you want to optimize apply to all the topics in the Kafka cluster, you can set the configuration parameters on all the brokers to be applied to all topics. On the other hand, if you want to optimize for different things on different topics, you can also set topic overrides for some of the configuration parameters. Without an explicit topic override, the broker configuration value applies.

There are hundreds of different configuration parameters, and this white paper introduces a very small subset of them that are relevant for the discussion. The parameter names, descriptions, and default values are up to date for Confluent Platform version 5.2 (Kafka version 2.2). Consult the documentation for more information on these configuration parameters, topic overrides, and other configuration parameters.

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, brokers, 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 utilize all brokers in the cluster. Although it might seem tempting just to create topics with a large number of partitions, there are tradeoffs to increasing the number of partitions. You will need to choose the partition count carefully based on consumer throughput and producer throughput. Review our guidelines for how to choose the number of partitions, and be sure to benchmark performance in your environment.

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 the brokers, 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 tradeoff 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 `lz4` and avoid `gzip` because it can be a CPU hog. Compression is applied on full batches of data, so the efficacy of batching will also impact the compression ratio—more batching results in better compression ratios. When the broker 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, the broker 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, the broker 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.

Decompression and recompression can also happen if producers are running a version prior to 0.10 because offsets need to be overwritten, or if any other message format conversion is required. But otherwise, as long as you are running updated clients and `log.message.format.version` is up to date, no message conversion is necessary.

When a producer sends a message to a Kafka cluster, the message is sent to the leader broker for the target partition. Then the producer awaits a response from the leader broker 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 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 acknowledgement. Setting `acks=1` makes it so that the leader broker will write the record to its local log and then acknowledge the request without awaiting acknowledgement from all followers. The tradeoff 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 which, in turn, enables better utilization of the bandwidth across more brokers.

Likewise, you can tune the consumers for higher throughput by adjusting how much data it gets from each fetch from the leader broker. 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 the broker, 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.

Finally, you should tune the JVM to minimize garbage collection (GC) pause time. GC is essential because it deletes unused objects and reclaims memory. However, long GC pauses are undesirable because they can negatively impact throughput and at worst case cause a broker soft failure, e.g., an expired ZooKeeper session timeout. Please read our deployment guidelines for more information on tuning JVM GC optimally.

Summary of Configurations for Optimizing Throughput

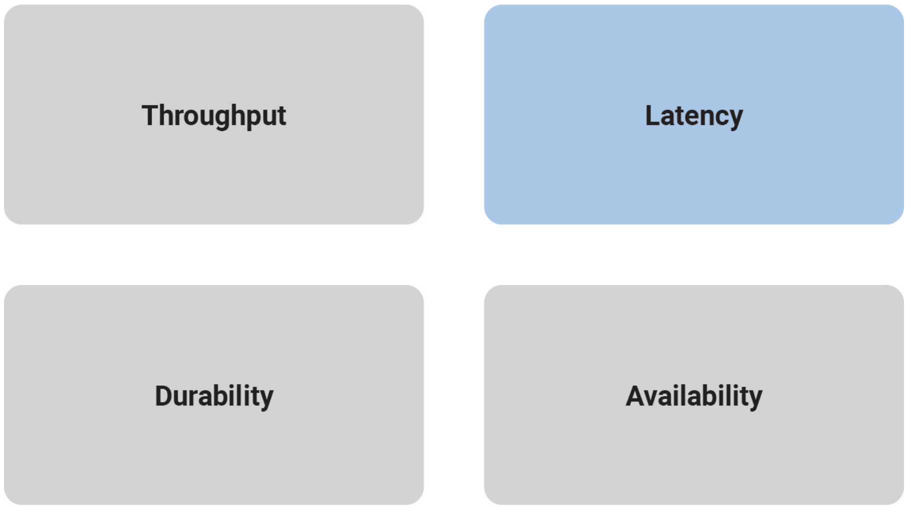
Producer:

- **`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 reinforce understanding on 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 tradeoff in that an increased number of partitions may also increase latency. A broker by default uses a single thread to replicate data from another broker, so 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.

To address the potential fetch latency, one option is try to limit the number of partitions on any given broker. You can do that either by limiting the number of partitions in the entire cluster (i.e., don't set the number of partitions per topic arbitrarily high), or by increasing the number of brokers in the cluster so that each broker has fewer partitions. For real-time applications that have very low-latency requirements but which also require many partitions, you can tune the number of fetcher threads used to replicate messages from a source broker. To increase the degree of I/O parallelism in the follower broker, adjust the configuration parameter `num.replica.fetchers`, which is the number of fetcher threads per source broker. Start your benchmark tests at its default value of 1, and then increase it if followers can't keep up with the leader.

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 the Kafka cluster. 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 to have received before considering a request complete. (Note that this acknowledgement 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 acknowledgements 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 the consumers for lower latency by adjusting how much data it gets from each fetch from the leader broker. 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 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 KSQL, there are also some performance enhancements you can do 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 KSQL 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)

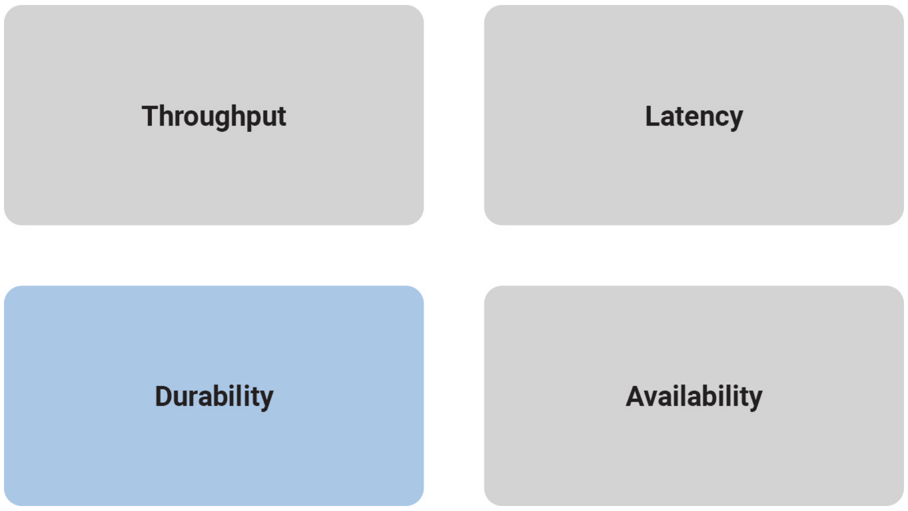
Streams:

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

Broker:

- `num.replica.fetchers`: increase if followers can't keep up with the leader (default 1)

Optimizing for Durability



Durability is all about reducing the chance for a message to get lost. The most important feature that enables durability is replication, which ensures that messages are copied to multiple brokers. If a broker has a failure, the data is available from at least one other broker. Topics with high durability requirements should have the configuration parameter `replication.factor` set to 3, which will ensure that the cluster can handle a loss of two brokers without losing the data. Also, if the Kafka cluster is enabled for auto topic creation (configuration parameter is `auto.create.topics.enable`), then you should consider changing the configuration parameter `default.replication.factor` to be 3 as well so that auto-created topics will be created with replication. Or, disable auto topic creation completely by setting `auto.create.topics.enable=false` so that you are always in control of the replication factor and partition settings for every topic.

Durability is important not just for user-defined topics, but also Kafka internal topics. Ensure all internal topics have the appropriate replication factor configured. These topics include:

- Consumer offsets topic: This topic, by default called `__consumer_offsets`, tracks the offsets of messages that have been consumed. If you are running a Kafka version with KIP-115, it will enforce `offsets.topic.replication.factor` upon the consumer offsets topic during auto topic creation.
- Kafka Streams application internal topics: These topics are created by the Kafka Streams application and used internally while executing, for example, the changelog topics for state stores and repartition topics. Its configuration setting `replication.factor` is configured to 1 by

default, so it may need to be increased.

- EOS transaction state log topic: If you are using exactly once semantics (EOS), the transaction state log topic stores the latest state of a transaction, e.g., “Ongoing,” “Prepare commit,” and “Completed,” and associated metadata. Its configuration setting **`transaction.state.log.replication.factor`** is configured to 3 by default.

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 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 tradeoff is tolerating a higher latency because the leader broker waits for acknowledgements 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 acknowledgement 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 the cluster that cause a producer retry, the producer may send duplicate messages to the broker
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 brokers 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 necessitate 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 the cluster, 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.

The Kafka cluster 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, and 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`, broker `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.

Kafka can extend the guarantees provided for broker failure to cover rack failures as well. The rack awareness feature spreads replicas of the same partition across different racks. This limits the risk of data loss in case all the brokers on a rack fail at once. This feature can also be applied to cloud-based solutions like Amazon's EC2 by assigning brokers to different availability zones. You can specify which rack a broker belongs to by setting the configuration parameter `broker.rack`, and then Kafka will automatically ensure that replicas span as many racks as they can.

If there are broker failures, the Kafka cluster can automatically detect the failures and elect new partition leaders. New partition leaders are chosen from the existing replicas that are running. The brokers in the ISR list have the latest messages and if one of them becomes the new leader, it can continue where the previous leader broker left off, copying messages to the replicas that still need to catch up. The configuration parameter `unclean.leader.election.enable` indicates whether brokers in the ISR list which are not caught up to the leader (i.e., "unclean") are eligible to become leaders themselves. For higher durability, this should be disabled by setting `unclean.leader.election.enable=false` to ensure that new leaders should be elected from just the ISR list. This prevents the chance of losing messages that were committed but not replicated. The tradeoff is tolerating more downtime until the other replicas come in sync.

Because we recommend that you use Kafka's replication for durability and allow the operating system to control flushing data from the page cache to disk, you generally should not need to change the flush settings. However, for critical topics with extremely low throughput rates, there may be a longer period of time before the OS flushes to disk. For topics like those, you may consider tuning either `log.flush.interval.ms` or `log.flush.interval.messages` to be small. For example, if you want to persist to disk synchronously after every message, you can set `log.flush.interval.messages=1`.

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. You can configure how commits happen with the configuration parameter `enable.auto.commit`. By default, offsets are configured to be automatically committed during the consumer's `poll()` call at a periodic interval. But if the consumer is part of a transactional chain and you need strong message delivery guarantees, you want the offsets to be committed only after the consumer finishes completely processing the messages. Thus, for durability, 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 streaming applications by setting the configuration parameter `processing.guarantee`.

Summary of Configurations for Optimizing Durability

Producer:

- `replication.factor=3` (topic override available)
- `acks=all` (default 1)
- `enable.idempotence=true` (default false), to handle message duplication and ordering
- `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)

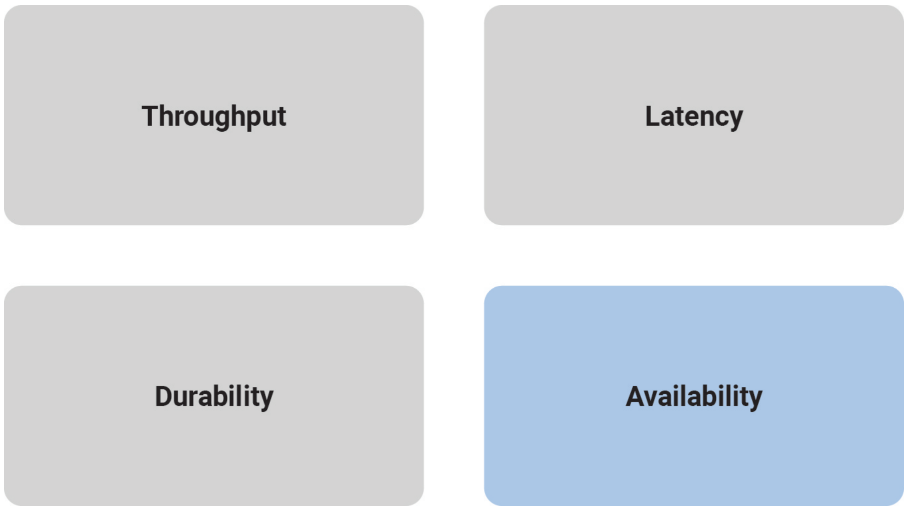
Streams:

- `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

Broker:

- `default.replication.factor=3` (default 1)
- `auto.create.topics.enable=false` (default true)
- `min.insync.replicas=2` (default 1); topic override available
- `unclean.leader.election.enable=false` (default false); topic override available
- `broker.rack`: rack of the broker (default null)
- `log.flush.interval.messages`, `log.flush.interval.ms`: for topics with very low throughput, set message interval or time interval low as needed (default allows the OS to control flushing); topic override available

Optimizing for Availability



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

Higher partition counts may increase parallelism, but having more partitions can also increase recovery time in the event of a broker failure. All produce and consume requests will pause until the leader election completes, and these leader elections happen per partition. So take this recovery time into consideration when choosing partition counts.

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.

Broker failures will result in partition leader elections. While this happens automatically, you do have control over which brokers are eligible to become leaders. To optimize for durability, new leaders should be elected from just the ISR list to prevent the chance of losing messages that were committed but not replicated. By contrast, to optimize for high availability, new leaders can be allowed to be elected even if they were removed from the ISR list. To make this happen, set the configuration parameter `unclean.leader.election.enable=true`. This makes leader election happen more quickly, which increases overall availability.

When a broker is started up, the broker scans its log data files in preparation for getting in sync with other brokers. This process is called log recovery. The number of threads per data directory to be used for log recovery at startup and log flushing at shutdown is defined by the configuration parameter `num.recovery.threads.per.data.dir`. Brokers with thousands of log segments will have a large number of index files, which can cause the log loading process on broker startup to be slow. If you are using RAID (high performance with fault tolerance), then increasing `num.recovery.threads.per.data.dir` to the number of disks may reduce the log loading time.

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`. The lower the session timeout is set, the faster a failed consumer will be detected, which will decrease time to recovery in the case of a failure. Set this as low as possible to detect hard failures but not so low that soft failures occur. 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`.

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 re-initialized 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`, the changelog is replayed from the earliest offset.

- If a local state store does exist, i.e., `num.standby.replicas` is greater than 0, 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`: as low as feasible (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

Broker:

- `unclean.leader.election.enable=true` (default false); topic override available
- `min.insync.replicas=1` (default 1); topic override available
- `num.recovery.threads.per.data.dir`: number of directories in `log.dirs` (default 1)

Benchmarking, Monitoring, and Tuning

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, hardware profile of each broker (CPU, memory, bandwidth, etc.), other features you have enabled, the data profile, etc. If you are tuning Kafka beyond the defaults, we generally recommend running benchmark tests. Regardless of your service goals, you should understand what the performance profile of the cluster 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, cluster size, and the number of producer and consumer processes.

Start benchmarking by testing your environment with the default Kafka configuration parameters, and please familiarize yourself with what the defaults are! Determine the baseline input performance profile for a given producer. First, set up the test 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 generated mock data unrealistically is padded with all zeros. This may result in compression performance results better than what would be seen in production. Ensure that the padded bytes reflect production data.

Run a single producer client on a single server. The Kafka cluster should be large enough so that it is not a bottleneck. 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.

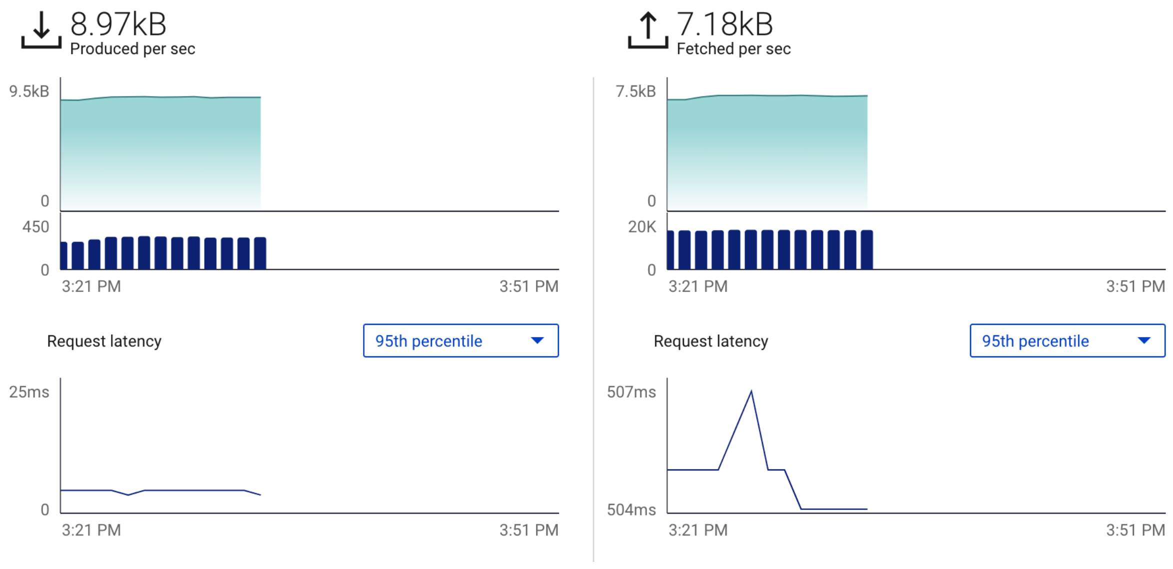
Then you can run a benchmark test for different permutations of configuration parameters that reflect your service goals. Focus on the configuration parameters discussed in this white paper, and avoid the temptation to discover and change other parameters from their default values without understanding exactly how they impact the entire system. Focusing on the parameters already discussed, 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.

Before you go into production, make sure you put a robust monitoring system in place for all of the brokers, producer, consumers, topics, and any other Kafka or Confluent components you are using. Health monitoring provides assurance that components are functioning well—that brokers are up and running, each one has enough disk space, that every message is received end to end, and so on. Performance monitoring provides a quantitative account of how each component is doing. This monitoring is important once you're in production, especially since production environments are dynamic: Data profiles may change, you may add new clients, you may scale out your cluster, and you may enable new Kafka 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.

Confluent offers the enterprise-class Confluent Control Center for monitoring the health and performance of your Kafka cluster. Alternatively, you may start the Kafka processes with the `JMX_PORT` environment variable configured in order to expose the internal JMX metrics to JMX tools. There are many Kafka internal metrics that are exposed through JMX to provide insight into the performance of the cluster.

When you operationalize the monitoring of the Kafka cluster, you may consider setting alerts based on your service goals. These alerts may even vary topic to topic. For example, one topic that is servicing an application with low-latency requirements may have different alert levels compared to another topic that is servicing an application with high-throughput requirements.

First, gauge the load in the cluster. The system health dashboard view in Control Center provides a summary of the produce and fetch requests across the cluster.



The following metrics are a good starting point to gauge the load on a broker.

Metric	Description
`kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec`	The bytes in per second the broker is receiving
`kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec`	The bytes out per second the broker is sending
`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec`	Number of incoming messages per second
`kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|FetchConsumer|FetchFollower}`	Number of requests per second, for produce, consumer fetch, and replica follower fetch
`kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|FetchConsumer|FetchFollower}`	Total time a request takes to be completed, for produce, consumer fetch, and replica follower fetch

If you want to identify performance bottlenecks, you will need to monitor metrics that reveal where processing time is being spent. You may need to pair multiple JMX metrics to see where the bottleneck lies. For example, if the metric `TotalTimeMs` shows a long time to process requests, you can follow up with the metrics below to understand where that request time is being spent.

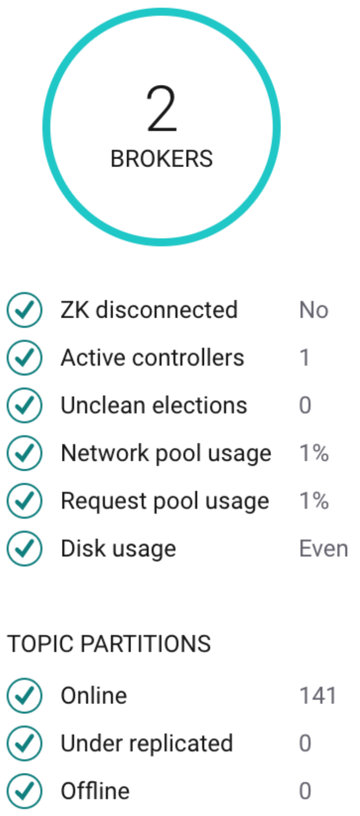
Metric	Description
`kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}`	The time the request is waiting in the request queue, for produce, consumer fetch, and replica follower fetch. A high value can imply there aren't enough IO threads or the CPU is a bottleneck.
`kafka.network:type=RequestMetrics,name=LocalTimeMs,request={Produce|FetchConsumer|FetchFollower}`	The time the request is being processed by the leader locally, for produce, consumer fetch, and replica follower fetch. Often a high value implies a slow disk.
`kafka.network:type=RequestMetrics,name=RemoteTimeMs,request={Produce|FetchConsumer|FetchFollower}`	The time the request is waiting on a remote client, for produce, consumer fetch, and replica follower fetch. A high value can imply a slow network connection. For fetch requests, if the remote time is high, it can be that there is not enough data to give in a fetch response.
`kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request={Produce|FetchConsumer|FetchFollower}`	The time the request is waiting in the response queue, for produce, consumer fetch, and replica follower fetch. A high value can imply there aren't enough network threads.
`kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request={Produce|FetchConsumer|FetchFollower}`	The time the request is being sent back to the client, for produce, consumer fetch, and replica follower fetch. A high value can imply there aren't enough network threads or the CPU or network is a bottleneck.

Metric	Description
`kafka.network:type=RequestChannel,name=ResponseQueueSize`	The size of the request queue. This should ideally be close to zero. Spikes are normal but a constantly saturated request queue will not be able to process incoming or outgoing requests.
`kafka.network:type=RequestChannel,name=ResponseQueueSize`	The aggregate size of the response queue. If this grows over time, it can be a sign of contention on either the send side or the request handling side.
`kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent`	The average fraction of time the network threads are idle. A lower value indicates more resources are used. Useful when paired with the above request-related JMX metrics
`kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent`	The average fraction of time the I/O threads are idle. Useful when paired with the above request-related JMX metrics.
`kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs`	The frequency of Kafka log page cache flushes, and how long the flushes take. Can be used to diagnose a slow disk or misconfigured page cache. Flushes that take a long time indicate poor storage performance.

For your producer benchmark tests, you should make sure the producer code itself is not introducing delays. Monitor the producer time spent in user processes. Using the `io-ratio` and `io-wait-ratio` metrics described below, user processing time is the fraction of time not in either of these, so if time in these are low, then the user processing time may be high and it keeps the single producer I/O thread busy. For example, 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 the I/O thread spent doing I/O
`kafka.producer:type=producer-metrics,client-id=([-.w]+),name=io-wait-ratio`	Fraction of time the I/O thread spent waiting

For general cluster health—that is, brokers are up and running and partitions are being replicated—monitor overall cluster state in Control Center.



2
BROKERS

✓	ZK disconnected	No
✓	Active controllers	1
✓	Unclean elections	0
✓	Network pool usage	1%
✓	Request pool usage	1%
✓	Disk usage	Even

TOPIC PARTITIONS

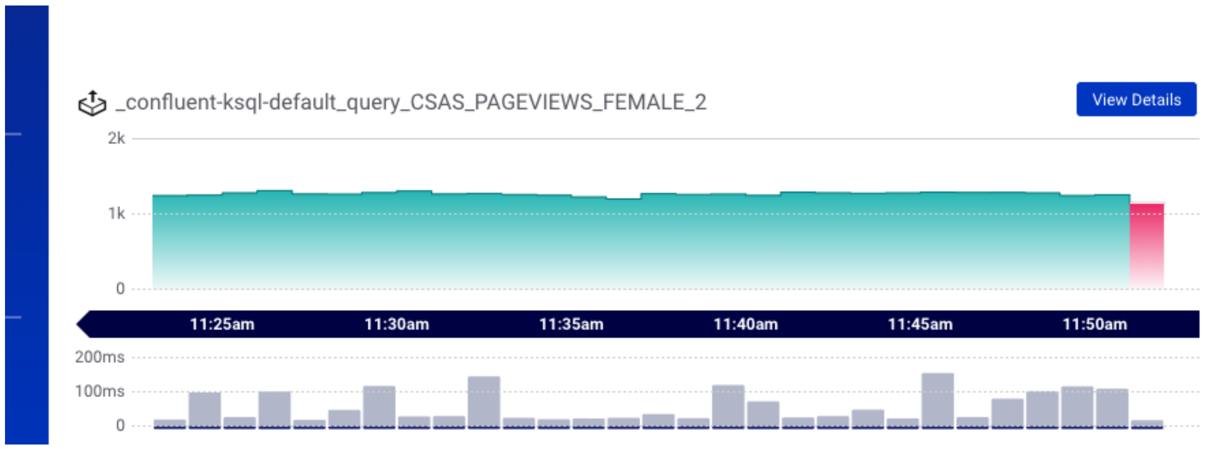
✓	Online	141
✓	Under replicated	0
✓	Offline	0

If you are capturing JMX metrics, monitor `IsrShrinksPerSec` and `UnderReplicatedPartitions`.

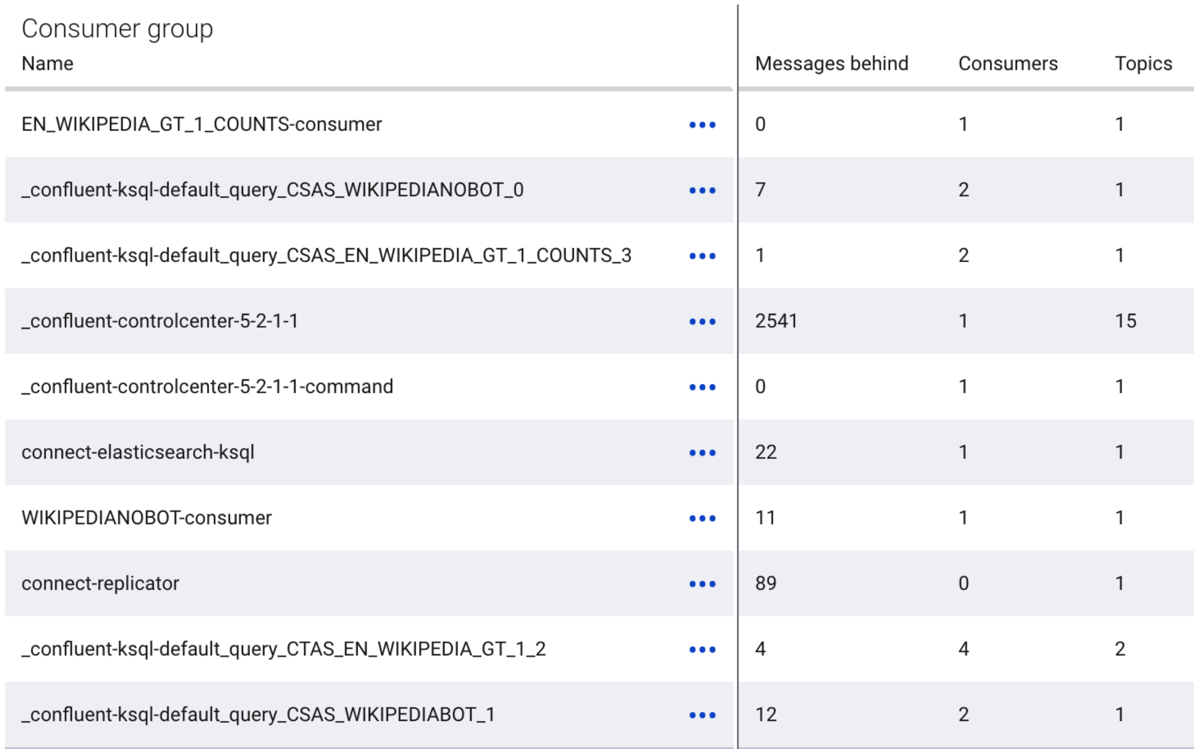
If these values are thrashing and you are not deliberately shutting down brokers gracefully, then it is likely brokers are soft failing or there is a connectivity issue between them.

Metric	Description
`kafka.server:type=ReplicaManager,name=IsrShrinksPerSec`	The rate at which the ISR is shrinking. The ISR will shrink if a broker is shutdown, either gracefully or not.
`kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`	Should always be 0. If it is greater than 0, likely a broker is not keeping up with replication.

Additionally, if your service goal is low latency, monitor the streams end to end for throughput and latency, per consumer group, consumer, topic, or partition.



To see how far behind the consumer client applications are in reading the log files, monitor the consumer lag, per consumer group and per consumer.



Consumer group

Name	Messages behind	Consumers	Topics
EN_WIKIPEDIA_GT_1_COUNTS-consumer	0	1	1
_confluent-ksql-default_query_CSAS_WIKIPEDIANOBOT_0	7	2	1
_confluent-ksql-default_query_CSAS_EN_WIKIPEDIA_GT_1_COUNTS_3	1	2	1
_confluent-controlcenter-5-2-1-1	2541	1	15
_confluent-controlcenter-5-2-1-1-command	0	1	1
connect-elasticsearch-ksql	22	1	1
WIKIPEDIANOBOT-consumer	11	1	1
connect-replicator	89	0	1
_confluent-ksql-default_query_CTAS_EN_WIKIPEDIA_GT_1_2	4	4	2
_confluent-ksql-default_query_CSAS_WIKIPEDIABOT_1	12	2	1

If you are capturing JMX metrics, monitor `records-lag-max`. This metric compares the offset most recently seen by the consumer to the most recent offset in the log. This metric is important for real-time consumer applications where the consumer should be processing the newest messages with as low latency as possible, so the lag should be small.

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.

Additionally, you should monitor the underlying operating system since the health of the operating system is the foundation for the health of Kafka. These are some of the components you should monitor:

- Memory utilization

- Disk utilization
- CPU utilization
- Open file handles
- Disk I/O
- Network I/O

If you would like any assistance with optimizing your Kafka deployment, benchmark testing, or monitoring, you can leverage the expertise of the professional services team at Confluent.

Additional Resources

- When it Absolutely, Positively, Has to be There: Reliability Guarantees in Kafka
- Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)
- How to Choose the Number of Topics/Partitions in a Kafka Cluster
- Confluent Professional Services
- Confluent Training
- Confluent Platform Download