cloudkarafka

Apache Kafka

Beginners Guide

Things we wish we knew before we started learning Apache Kafka

Acknowledgement

We want to express a big thank you to everyone who has helped us from the earliest draft to the published first edition of this e-book. A special thanks to our lovely colleagues at 84codes and to all of our remote-based tech friends. Finally, a huge thank you to all of our CloudKarafka users for your feedback and continued support.

We'd love to hear from you!

We encourage you to email us any comments that you might have about the e-book. Feedback is crucial for the next edition so feel free to tell us what you think should or shouldn't be included. If you have an application that is using CloudKarafka or a user story that you would like to share, please send us an email!

This book is for anyone who has heard about Apache Kafka and is curious to learn more but keeps getting lost in advanced documentation sites around the Apache Kafka community. We feel you, we hear you and we want to say:

Look no further! Give this a read and we look forward to meeting you in the community chats in the future!

"The expert at anything was once a beginner." - Helen Hayes

Introduction

The interest in Apache Kafka is higher than ever. Companies from a wide spectrum of industries are confronting a time where they need to rethink their infrastructure and be able to keep up with the expectations of today. Not only do they have to take customers' needs into consideration, who are demanding fast and reliable services, but the very core of a company also needs to adopt a paradigm shift of building scalable and flexible solutions that are ready to handle data on the spot. Because of this, eyes are turning towards Kafka for good reason.

Apache Kafka, which is written in Scala and Java, is a creation of former LinkedIn data engineers and was handed over to the open-source community in early 2011 as a highly scalable messaging system. Today, Apache Kafka is a part of Confluent Stream Platform and handles trillions of events every day. *Apache Kafka has established itself on the market with many trusted companies waving the Kafka banner.*

Today we're surrounded by data everywhere and the amount of data has increased a lot in a short matter of time. All of a sudden; everybody owns a smart home, a smart car and a coffee maker that senses your mood and makes you a cup of coffee if needed (I WISH!).

Data and logs that surround us need to be processed, reprocessed, analyzed and handled. Often in real-time. That's what makes the core of the web, IoT and cloud-based living of today. And that's why Apache Kafka has come to play a significant role in the message streaming landscape.

The key design principles of Kafka were formed based on this growing need for high throughput, easily scalable architectures that provide the ability to store, process and reprocess streaming data.

This book might be your first introduction to Kafka or maybe you just want to refresh your knowledge bank. Maybe your team of developers are pushing for this Kafka-thing and you need to find something that gives you a quick understanding to be able to say GO! or NO!

Either way, we hope you like this book and that you, after reading it, is feeling more secure in the Kafka landscape.

Part 1: Apache Kafka Beginner

As mentioned, Apache Kafka is on the rise since the world is approaching a new way of seeing, living and handling data. In Part 1, Apache Kafka is described from a beginner perspective. It gives a brief understanding of messaging and distributed logs and defines important concepts along the way. Part 1 is also a walkthrough of how to set up a connection to an Apache Kafka cluster and how to publish and subscribe records from this cluster.

Let's dig in...
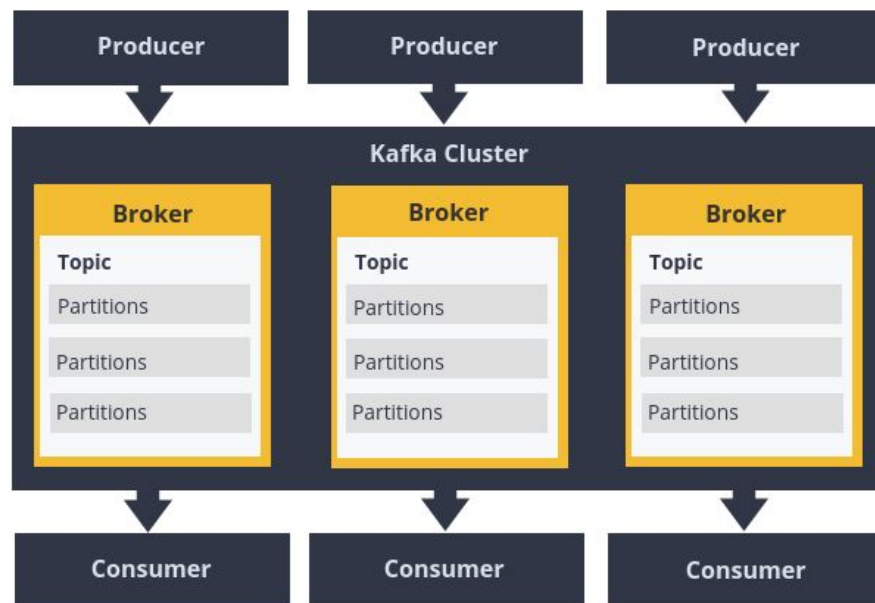
This image shows the very foundation of Apache Kafka and its components.
Producer - Cluster - Broker - Topic - Partition and Consumer

What is Apache Kafka?

Apache Kafka is a publish-subscribe (pub-sub) message system that allows messages (also called records) to be sent between processes, applications, and servers. Simply said - Kafka stores streams of records.

A record can include any kind of information. It could, for example, have information about an event that has happened on a website or could be a simple text message that triggers an event so another application may connect to the system and process or reprocess it.

Unlike most messaging systems, the message queue in Kafka (also called a log) is persistent. The data sent is stored until a specified retention period has passed by. Noticeable for Apache Kafka is that records are not deleted when consumed.

An Apache Kafka cluster consists of a chosen number of brokers/servers (also called nodes). Apache Kafka itself is storing streams of records. A record is data containing a key, value and timestamp sent from a producer. The producer publishes records on one or more topics. You can think of a topic as a category to where, applications can add, process and reprocess records (data). Consumers can then subscribe to one or more topics and process the stream of records.

Kafka is often used when building applications and systems in need of real-time streaming.

Topics and Data Streams

All Kafka records are organized into topics. Topics are the categories in the Apache Kafka broker to where records are published. Data within a record can be of various types, such as String or JSON. The records are written to a specific topic by a producer and subscribed from a specific topic by a consumer.

All Kafka records are organized into topics. Topics are the categories in the Apache Kafka broker where records are published. Data within a record can consist of various types, such as String or JSON. The records are written to a specific topic by a producer and subscribed from a specific topic by a consumer.

When the record gets consumed by the consumer, the consumer will start processing it. Consumers can consume records at a different pace, all depending on how they are configured.

Topics are configured with a retention policy, either a period of time or a size limit. The record remains in the topic until the retention period/size limit is exceeded.

Partition

Kafka topics are divided into partitions which contain records in an unchangeable sequence. A partition is also known as a commit log. Partitions allow you to parallelize a topic by splitting the data into a topic across multiple nodes.

Each record in a partition is assigned and identified by its unique offset. This offset points to the record in a partition. Incoming records are appended at the end of a partition. The consumer then maintains the offset to keep track of the next record to read. Kafka can maintain durability by replicating the messages to different brokers (nodes).

A topic can have multiple partitions. This allows multiple consumers to read from a topic in parallel. The producer decides which topic and partition the message should be placed on.

All Kafka records are organized into topics.

- **Topics are the categories to where records are published.**
- **Data within a record can be represented by various types.**
- **Records are written to a specific topic and subscribed from a specific topic.**
- **When the record reaches the consumer, the consumer will start processing it.**
- **The time of processing a record can be different for different consumers depending on consumer configuration.**

- **Topics are broken up into ordered commit logs called partitions.**
- **Messages in a partition are given a *sequential id* called an *offset*.**
- **The data in a topic is retained until a specified configuration (time/size/retention).**
- **Records can be read chronologically, but can also rewind/skip to any point in the partition.**

Replication - the power of copying and reproducing

In Kafka, replication is implemented at the partition level. The redundant unit of a topic partition is called a replica. A follower that is in-sync is called an in-sync replica. If a partition leader fails, a new in-sync replica is selected as the new leader. Each partition usually has one or more replicas meaning that partitions contain records that are replicated over a chosen number of Kafka brokers in the cluster.
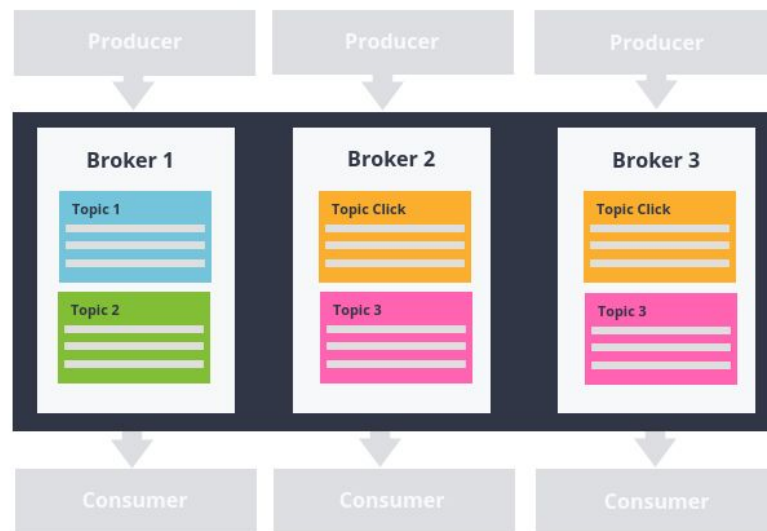
As we can see in the image above, the partitions in the "click topic" are replicated to Kafka Broker 2 and Kafka Broker 3.

It's possible for the producer to attach a key to the records and tell which partition the record should go to. These keys can be useful if you wish for a strong order, in case you are developing something that requires, for example, a unique id. When attaching a key to these records, it will ensure that records with the same key will arrive at the same partition.

Short Recap
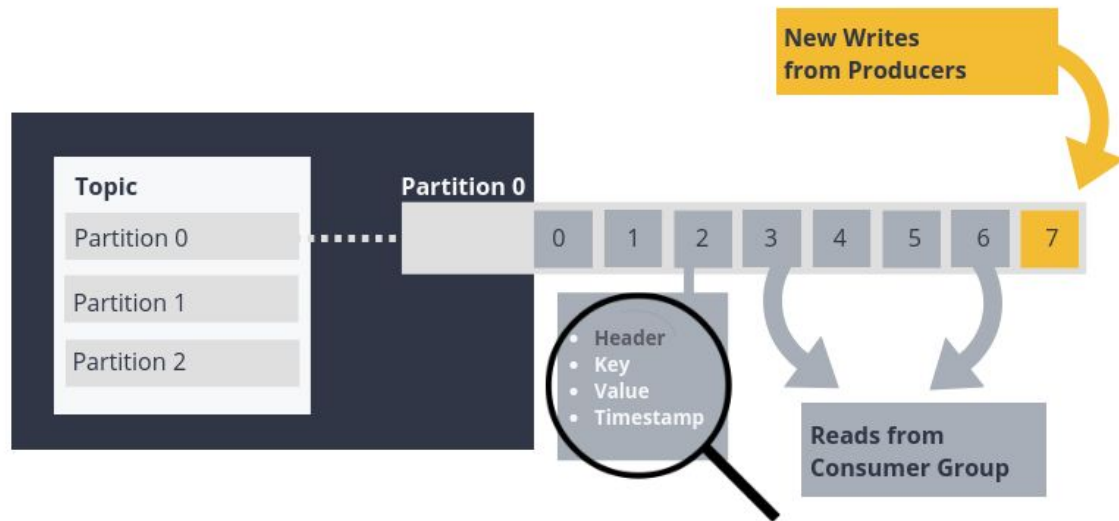
- Every Apache Kafka broker is divided into topics.
- Topics contain a decided number of partitions.
- The stream of records in the partitions is called commit log.
- The records hold information in form of a key, a value and a timestamp .
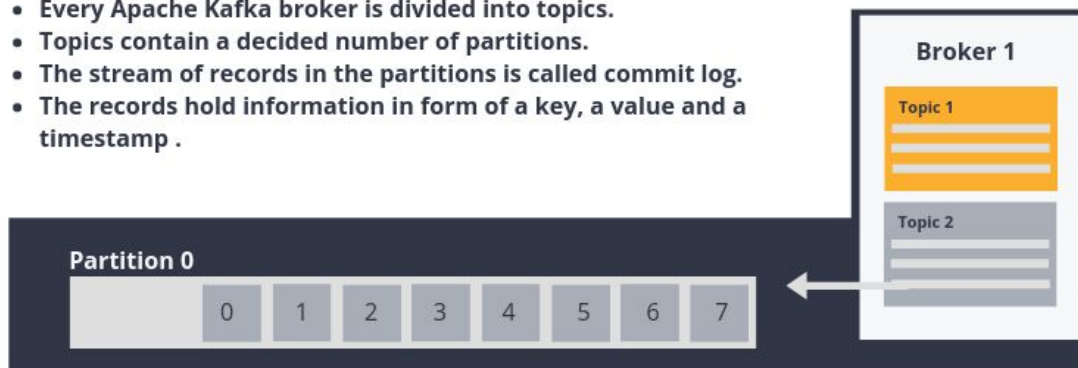
The function of "leaders" and the election of new leaders

This next part shows how records can be collaterally written to and read from, and also what makes Kafka fault-tolerant; meaning that your system continues to work at a level of satisfaction, even in the presence of failures.

One partition in a broker is marked as "*leader*" for the partition, and the others are marked as followers. The leader is a partition replica. Each broker can host multiple leaders and follower replicas. The leader controls the *read-and-writes* for the partition, whereas the followers replicate the data. If the partition leader fails, one of the followers become a new leader by

default. Zookeeper is used for leader election. We will leave Zookeeper for now, and get back to Zookeeper in **Part 2.**
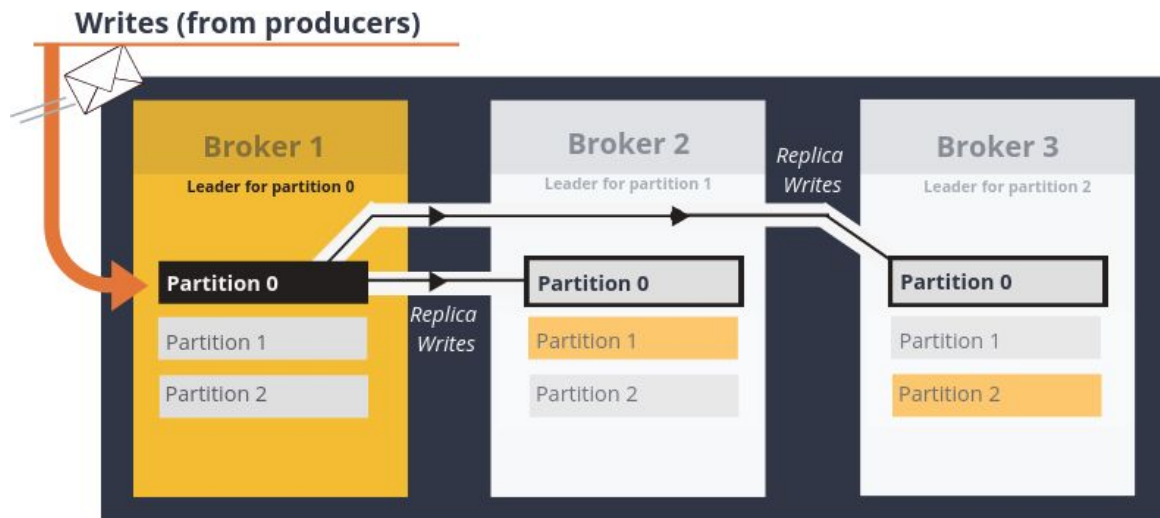
Broker 1 in the image is the leader for Partition 0
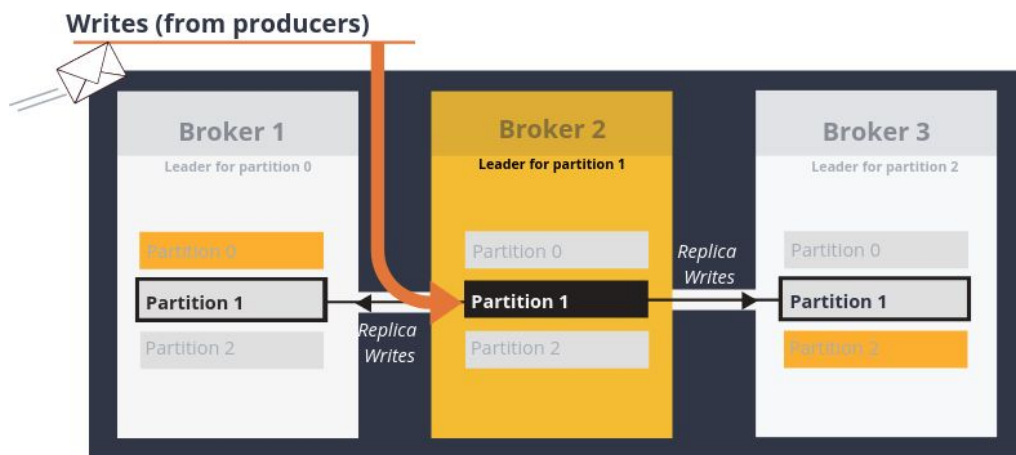
Broker 2 in the image is acting as the leader for Partition 1

The leader appends the records to its commit log and increments its *record offset*. Kafka then exposes the record to the consumer(s) after it has been received and committed.

When a record is fully committed depends on the producer ack-value configuration and in-sync replicas configuration. More about those topics can be found in **Part 2**.

Consumers and consumer groups

Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose. This allows consumers to join the broker at any point in time.

Consumers can join a group called a consumer group. A consumer group includes the set of consumer processes that are subscribing to a specific topic. Each consumer in the group is assigned a set of partitions to consume from. This allows Kafka a very high record processing throughput. Consumers will not read the same records and will subscribe to different subsets of the partitions in the topic. Kafka keeps track of all consumers, and can, therefore, guarantee that a message is only read by a single consumer in the group.
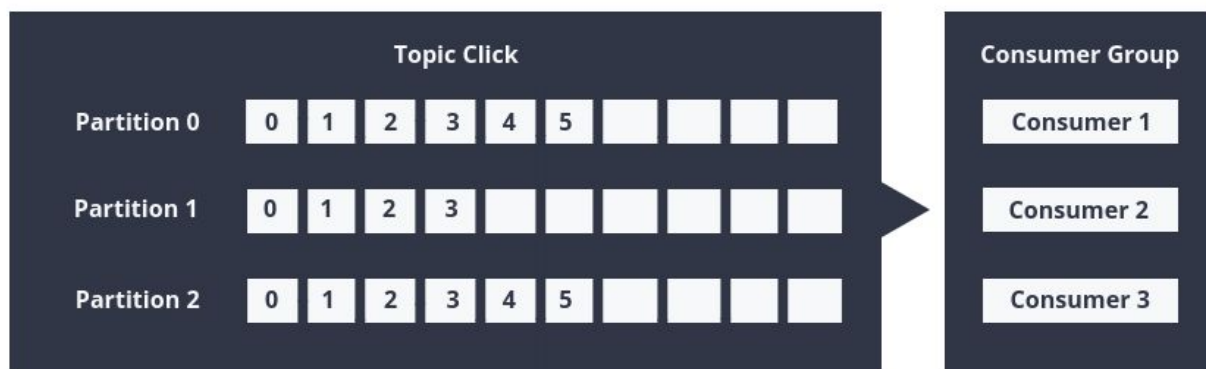
Kafka can support a large number of consumers and retain large amounts of data with very little overhead. The number of partitions impacts the maximum parallelism of consumers as you should not have more consumers than partitions.

The consumers will never overload themselves with lots of data or lose any data since all records are being queued up in Kafka. If the consumer is behind while processing records, there is the option to catch up and get back to handle data in real-time.

What makes Kafka Fault-Tolerant

- Kafka uses replicated partitions.
- A partition leader is selected to be responsible for replication.
- If a broker fails, another leader is chosen.
- Kafka only exposes a message to a consumer after it has been fully committed.
- Records are handled by consumers when the consumer is ready.
- Data loss or data overload is not possible since records remain in Kafka.

- Consumers can read messages from any offset point.
- Consumers can join clusters at any point in time.
- Consumers can be a part of a consumer group that are subscribing to a specific topic.
- Records are only read by a single consumer in the group.
- Multiple consumers can read from multiple partitions, allowing a very high record processing throughput.
- Kafka can support a large number of consumers and retain large amounts of data with very little overhead.

Apache Kafka Example

Website activity tracking

According to the creators of Apache Kafka, the original use case for Kafka was to track website activity - including page views, searches, uploads or other actions users may take. This kind of activity tracking often requires a very high volume of throughput, since messages are generated for each action and for each user.

We will now explain Apache Kafka with an example by using the concept of a basic website.

In this example, users can: click around, sign in, write blog articles,
upload images to articles and publish those articles.

When an event happens on the website, for example, when someone logs in, presses a button or when someone uploads an image to the article, a tracking event is triggered. Information about the event (a record) is placed into a specified Kafka topic. In this example, there is one topic named "click" and one named "upload".

Partitioning is based on the user id. A user with id 0 is mapped to partition 0, and a user with id 1 is mapped to partition 1, etc. The "click" topic will be split up into three partitions (three users) on two different machines.

Example:

"Button Click" Producer

"Handle Data" Consumers And Consumer Groups
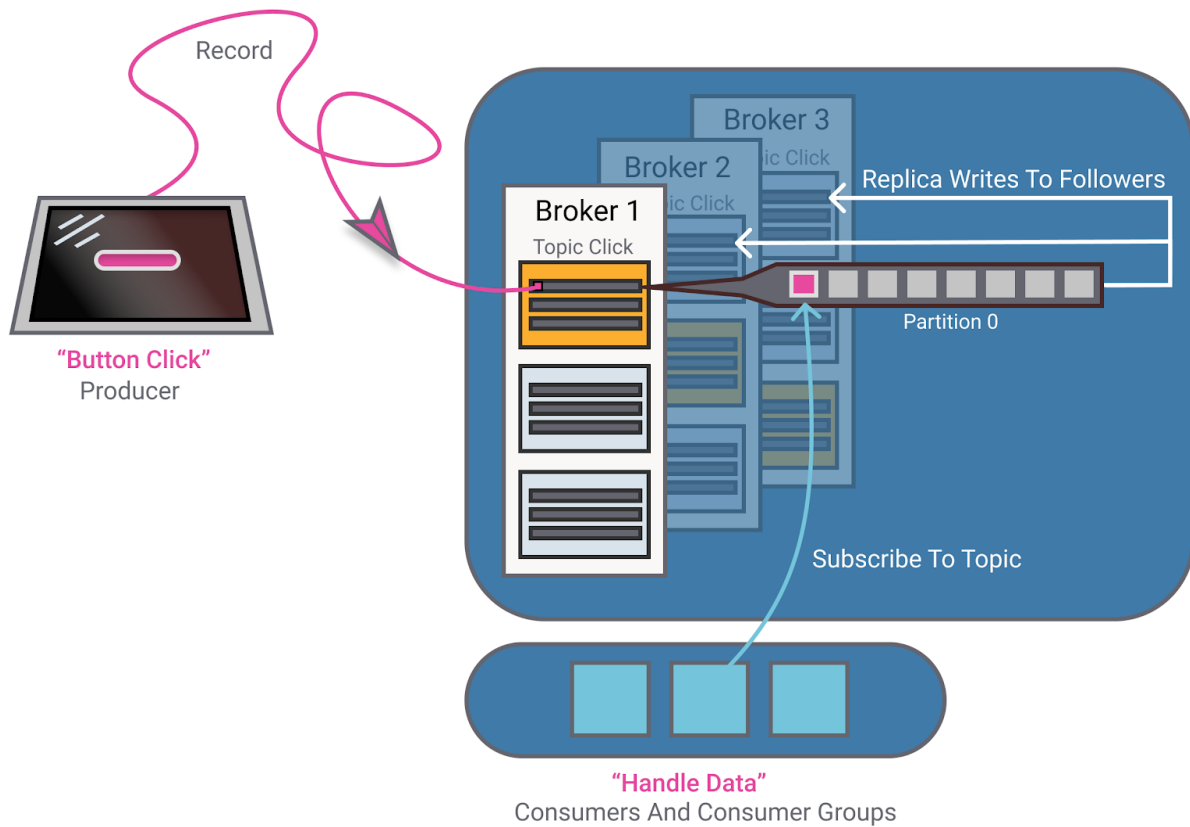
1. *A user with user-id 0 clicks on a button on the website.*
2. *The web application publishes the record to topic "click" and partition 0.*
3. *The message is appended to its commit log and the message offset is incremented.*
4. *Broker 1, which is the leader of partition 0, replicated the record to its followers, broker 2 and broker 3.*
5. *The consumer can subscribe to messages from the click-topic.*

The consumer that handles the message is now able to show monitoring usage in real-time or can replay previously consumed messages by setting the offset to a previous offset.

Example usage of Apache Kafka

Kafka is a great tool for delivering messages between producers and consumers, plus the optional topic durability allows you to store your messages permanently. Forever if you'd like! There are a lot of Kafka use cases out there. In this chapter, we're listing the most common ones.

These images show a number of producers and consumers that might write and subscribe to records to and from the Kafka Broker. This shows a variety of use cases for Apache Kafka and how it can be used as a part of an it-architecture of this kind.

Message Service

Kafka can work as a replacement for more traditional message brokers, like RabbitMQ. Millions of messages can be sent and received in real-time. Messaging decouples your processes and creates a highly scalable system. Instead of building one large application, it's beneficial to decouple different parts of your application and let communication between them be handled

asynchronously through messages (and this is what we refer to "monolith vs. microservice based architecture").

This way, different parts of your application can evolve independently, be written in different languages and/or be maintained by separated developer teams. Kafka has built-in partitioning, replication, and fault-tolerance that makes it a good solution for large-scale message processing applications.

Real-time event stream processing

Kafka can be used to aggregate and process events in real-time, such as user activity data, like clicks, navigation, and search forms from different websites of an organization. These activities can be sent to real-time monitoring systems, real-time analytics platforms and or to mass storage (like S3) for offline/batch processing.

Events can also be processed and written back to other topics in real-time using the Kafka Streams API. This is a library that can be used to create streaming applications that combine and write to multiple streams (topics) forming simple or complex processing topologies.

Log aggregation

A lot of people today are using Kafka as a log solution. Log aggregation is about finding an efficient way to gather the entries from your various log files into one single, organized place.

Data Ingestion

Today data is being used for more uses and companies are gathering data in larger quantities and at higher velocities. Multiple technologies and platforms may be leveraged to gain insights into data, provide search, auditing and any number of uses. Kafka's ability to scale makes it perfect as the front-line for data ingestion. This way the various producers of data only need to send their data to a single place while a host of backend services can consume the data as they wish. All the major analytics, search and storage systems have integrations with Kafka, making it the perfect ingestion technology.

Commit log service

A commit log is about recording the changes made to something so that it can be replayed later. Kafka can be used as a commit log since all data is stored in the topic until the configured retention has passed.

Event sourcing

Event sourcing is an architectural style where domain events are treated as first-class citizens and the primary source of truth of a system. In today's polyglot persistence world, event sourcing allows multiple data stores such as RDBMS, search engines, caches, etc., to stay in-sync as they all are fed by the same stream of domain events. The current state for any entity can be reconstructed by replaying its events.

Kafka makes a great platform for event sourcing because it stores all records as a time-ordered sequence and provides the necessary ordering guarantees that an event store needs.

Get started with Apache Kafka

NOTE: To be able to follow this guide you need to **SET UP** a Kafka cluster at CloudKarafka, or **DOWNLOAD** and install Apache Kafka and Zookeeper on your own servers.

Hosted free Apache Kafka instance at CloudKarafka

CloudKarafka is a hosted Apache Kafka solution that automates every part of the setup, meaning that all you need to do is sign up for an account and create your broker. You can select which datacenter you want to have your broker in, and the number of nodes in your setup.

A big benefit of using CloudKarafka is that you don't need to set up and install Kafka, care about cluster handling or ZooKeeper. CloudKarafka can be used for free with the plan *Developer Duck*. Go to the plan page and sign up for any plan and create an instance.
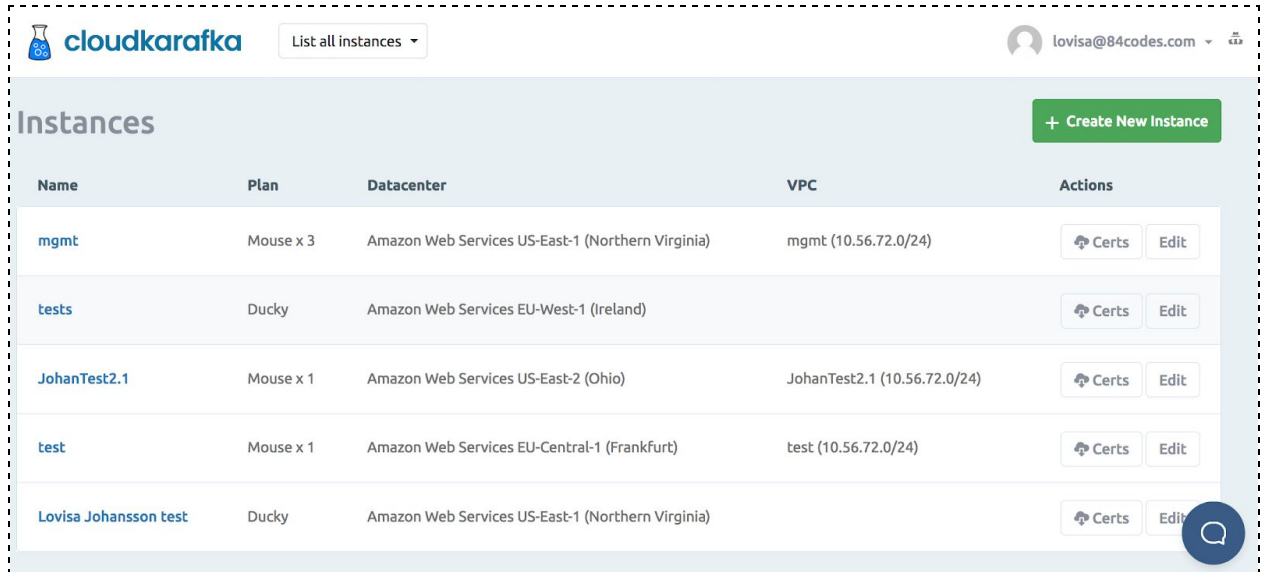
When your instance is created, click on the *details*. Before you start coding you need to ensure that you can set up a secure connection. You can download certificates, use SASL/SCRAM or set up VPC peering to your AWS VPC.

This tutorial shows how to get started with the free plan, *Developer Duck,* since everyone should be able to complete this guide. If you are going to set up a dedicated instance, we recommend you to have a look here.

You need to connect via SASL/SCRAM or Certificates to shared (and free) plans. VPC is another option that is only available for dedicated plans.

Secure connection via certificates

Get started by downloading the certificates (connection environment variables) for the instance. You can find the cert download button from the instances overview page. It is named: *Certs* as seen in the image below. Press the "Create New Instance" button and save the given *.env* file into your project. The file contains environmental variables that you need to use in your project.

Secure connection via SASL/SCRAM

You can also authenticate using [SASL/SCRAM.](#) When using SASL/SCRAM you only need to locate the username and password on the *Details* page and add them in your code.
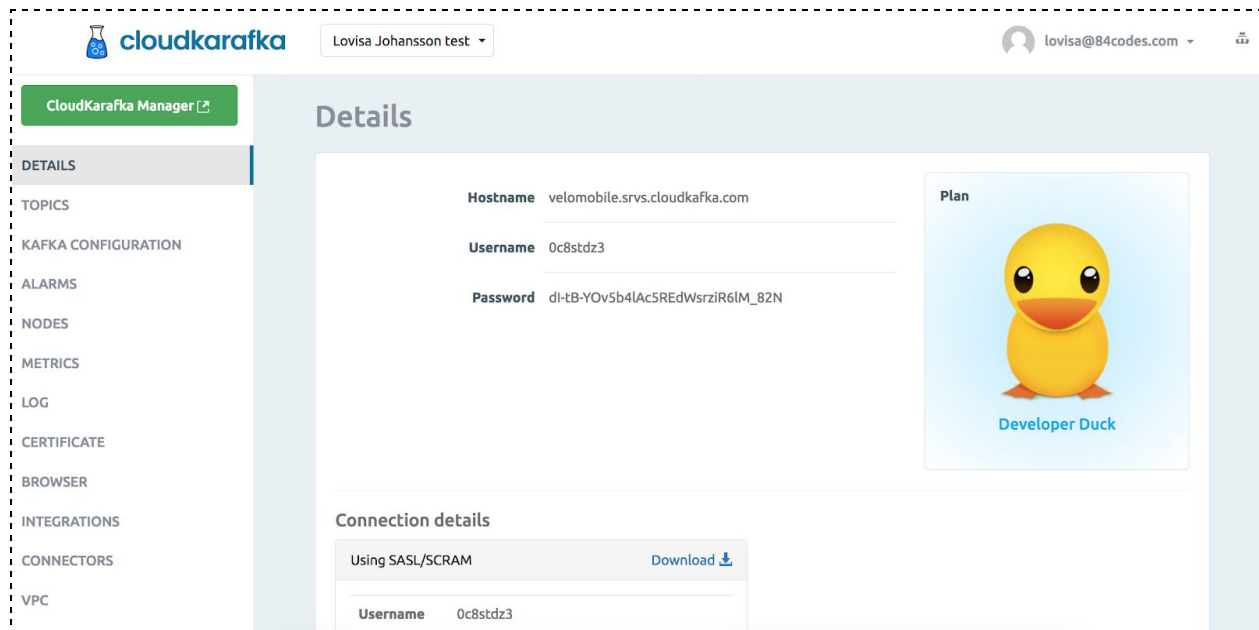
Secure connection via VPC

VPC information can be found by opening up the *VPC Peering* tab in the CloudKarafka control panel. You will find peering information on the details page for your instance. CloudKarafka will request to set up a VPC connection as soon as you have saved your VPC peering details. After that, you will need to accept the VPC peering connection request from us. The request can be accepted from the Amazon VPC console at https://console.aws.amazon.com/vpc/. Please note that the subnet given must be the same as your VPC subnet.

Create a topic

You can create a topic by opening the *Topic* view. You are free to decide partitions, replicas, retention byte and retention time in milliseconds.

In this example, two topics are created, *0c8stdz3-click* and *0c8stdz3-update*. Those are symbolizing the topics from the example in the previous chapters, where we have partitioning based on the user id. A user with id 0 is mapped to partition 0, and a user with id 1 is mapped to partition 1, etc. The *0c8stdz3-click* topic is split up into three partitions (three users) on two different machines.

CloudKarafka MGMT

CloudKafka MGMT interface is enabled by default on all clusters, including shared clusters (developer duck). From here, topics, consumers, retention period, users and permissions can be handled - created, deleted and listed in the browser and you can monitor message rates, as well as send or receive messages manually.

Publish and subscribe

To be able to communicate with Apache Kafka, you need a library or framework that understands Apache Kafka. In other words, you need to download the client-library/framework for the programming language that you intend to use for your applications.

A client-library is an "applications programming interface" (API) for use in writing client applications. The producer and consumer APIs have several methods that can be used, in this case, to communicate with Apache Kafka. The methods should be used when you, for example, connect to the Kafka broker (using the given parameters, a hostname for instance) or when you publish a message to a topic. Both consumers and producers can be written in any language that has a Kafka client written for it.

1. **Set up an Apache Kafka instance at CloudKarafka.**

2. **Download the client library you attend to use.**

3. **Set up a secure connection.** A TCP connection will be set up between the application and Apache Kafka.

4. **In publisher: Publish a message to a partition on a topic.**

5. **In consumer: Consume a message from a partition in a topic.**

The consumer can subscribe from the latest offset, or it can replay previously subscribed records by setting the offset to an earlier one.

Apache Kafka and Ruby

This tutorial contains step-by-step instructions that show how to set up a secure connection, how to publish to a topic, and how to subscribe from a topic in Apache Kafka with Ruby.

Once you have your Apache Kafka instance, you need to download the API for Ruby. You can find the complete ruby code example on GitHub:
https://github.com/CloudKarafka/ruby-kafka-example

The sample project contains everything you need to get started with producing and consuming records with Kafka.

Producer code
```ruby
require 'bundler/setup'
require 'rdkafka'

config = {
        :"bootstrap.servers" => ENV['CLOUDKARAFKA_BROKERS'],
        :"group.id"        => "cloudkarafka-example",
        :"sasl.username"       => ENV['CLOUDKARAFKA_USERNAME'],
        :"sasl.password"       => ENV['CLOUDKARAFKA_PASSWORD'],
        :"security.protocol" => "SASL_SSL",
        :"sasl.mechanisms"   => "SCRAM-SHA-256"
}
topic = "#{ENV['CLOUDKARAFKA_TOPIC_PREFIX']}test"

rdkafka = Rdkafka::Config.new(config)
producer = rdkafka.producer

100.times do |i|
  puts "Producing message #{i}"
  producer.produce(
        topic:   topic,
        payload: "Payload #{i}",
        key:     "Key #{i}"
  ).wait
end
```

Consumer code
```ruby
require 'bundler/setup'
require 'rdkafka'

config = {
      :"bootstrap.servers" => ENV['CLOUDKARAFKA_BROKERS'],
      :"group.id"        => "cloudkarafka-example",
      :"sasl.username"     => ENV['CLOUDKARAFKA_USERNAME'],
      :"sasl.password"     => ENV['CLOUDKARAFKA_PASSWORD'],
      :"security.protocol" => "SASL_SSL",
      :"sasl.mechanisms"   => "SCRAM-SHA-256"
}
topic = "#{ENV['CLOUDKARAFKA_TOPIC_PREFIX']}test"

rdkafka = Rdkafka::Config.new(config)
consumer = rdkafka.consumer
consumer.subscribe(topic)

begin
  consumer.each do |message|
```

```ruby
    puts "Message received: #{message}"
  end
rescue Rdkafka::RdkafkaError => e
  retry if e.is_partition_eof?
  raise
end
```

Part 2 - Performance optimization for Apache Kafka

Part 2 will guide you in how to best tune your Kafka Cluster to meet your high-performance needs. You will find important tips, broker configurations, common errors and most importantly - we will give you our best recommendations for optimization of your Apache Kafka Cluster.

Performance optimization for Apache Kafka includes optimization tips for Kafka, divided up between Producers, Brokers, and Consumers.

Performance optimization for Apache Kafka - Producers

The producer in Kafka is responsible for writing the data to the Kafka Brokers and can be seen as the trigger in the Apache Kafka workflow. The producer can be optimized in various ways to meet the needs of your Apache Kafka setup. By refining your producer setup, you can avoid common errors and ensure your configuration meets your expectations.

Ack-value

An acknowledgment (**ACK**) is a signal passed between communicating processes to signify acknowledgment, i.e., receipt of the message sent. The ack-value is a producer configuration parameter in Apache Kafka and can be set to the following values:

acks=0

The producer never waits for an ack from the broker when the ack value is set to 0. No guarantee can be made that the broker has received the message. The producer doesn't try to send the record again since the producer never knows that the record was lost. This setting provides lower latency and higher throughput at the cost of much higher risk of message loss.

acks=1

When setting the ack value to 1, the producer gets an ack after the leader has received the record. The leader will write the record to its log but will respond without awaiting a full acknowledgment from all followers. The message will be lost only if the leader fails immediately after acknowledging the record, but before the followers have replicated it. This setting is the middle ground for latency, throughput, and durability. It is slower but more durable than acks=0.

acks=all

Setting the ack value to all means that the producer gets an ack when all in-sync replicas have received the record. The leader will wait for the full set of in-sync replicas to acknowledge the record. This means that it takes a longer time to send a message with ack value all, but it gives the strongest message durability.

Read more about ack-values in Kafka [here](#).

How to set the Apache Kafka ack-value

For the highest throughput set the value to 0. For no data loss, set the ack-value to all (or -1). For high, but not maximum durability and for high but not maximum throughput - set the ack-value to 1. Ack-value 1 can be seen as an intermediate between both of the above.

What does In-Sync really mean?

Kafka considers that a record is committed when all replicas in the In-Sync Replica set (ISR) have confirmed that they have written the record to disk. The acks=all setting requests that an ack is sent once all in-sync replicas (ISR) have the record. But what is the ISR and what is it for?

What is the ISR?

The ISR is simply all the replicas of a partition that are "in-sync" with the leader. The definition of "in-sync" depends on the topic configuration, but by default, it means that a replica is or has been fully caught up with the leader in the last 10 seconds. The setting for this time period is: *replica.lag.time.max.ms* and has a server default which can be overridden on a per topic basis.

At a minimum the, ISR will consist of the leader replica and any additional follower replicas that are also considered in-sync. Followers replicate data from the leader to themselves by sending Fetch Requests periodically, by default every 500ms.

If a follower fails, then it will cease sending fetch requests and after the default, 10 seconds will be removed from the ISR. Likewise, if a follower slows down, perhaps a network related issue or constrained server resources, then as soon as it has been lagging behind the leader for more than 10 seconds it is removed from the ISR.

What is ISR for?

The ISR acts as a tradeoff between safety and latency.

As a producer, if we really didn't want to lose a message, we'd make sure that the message has been replicated to all replicas before receiving an acknowledgment. But this is problematic as the loss or slowdown of a single replica could cause a partition to become unavailable or add

extremely high latencies. So the goal to be able to tolerate one or more replicas being lost or being very slow.

When a producer uses the "all" value for the acks setting. It is saying: only give me an acknowledgment once all in-sync replicas have the message. If a replica has failed or is being really slow, it will not be part of the ISR and will not cause unavailability or high latency, and we still, normally, get redundancy of our message.

So the ISR exists to balance safety with availability and latency. But it does have one surprising Achilles heel. If all followers are going slow, then the ISR might only consist of the leader. So an acks=all message might get acknowledged when only a single replica (the leader) has it. This leaves the message vulnerable to being lost. This is where the min-insync.replicas broker/topic configuration helps. If it is set it to 2 for example, then if the ISR does shrink to one replica, then the incoming messages are rejected. It acts as a safety measure for when we care deeply about avoiding message loss.

Batch messages in Apache Kafka

Records can be sent together in a specific way as groups, called a batch. The batch can then be sent when the specified criteria for the batch is met; when the number of records for the batch has reached a certain number or after a given amount of time. Sending batches of messages is recommended since it will increase the throughput.

Always keep a good balance between building up batches and the sending rate. A small batch might give low throughput and lots of overhead. However, a small batch is still better than not using batches at all. A batch that's too large might take a long time to collect, keeping consumers idling. This depends on the use case too. If you have a real-time application make sure you don't have large batches.

Compression of large records

The producer can compress records and the consumer can decompress them. We recommend that you compress large records to reduce the disk footprint and also the footprint on the wire. It's not a good idea to send large files through Kafka. Put large files on shared storage instead of sending it through Kafka. Read more about compression in Apache Kafka here.

Apache Kafka client libraries

The Protocol for Apache Kafka changes a lot. It's therefore hard for clients to keep up to date with all of these changes. Always make sure to use Apache Kafka clients that are up to date. The Java client is always the feature-complete client. All demos on the CloudKarafka documentation pages are wrappers around **librdkafka**.

Performance optimization for Apache Kafka - Brokers

By refining the broker setup, you can avoid common errors and ensure your configuration meets your expectations.

Topics and Partitions

This section describes how to set up topics and partitions.

More partitions - higher throughput

One partition is only able to handle one consumer. The number of consumers should, therefore, be equal to the number of partitions. Multiple partitions allow for multiple consumers to read from a topic in parallel. This creates a more scalable system. With more partitions, it's possible to handle a larger throughput since all consumers can work in parallel.

Do not set up too many partitions

Partitions are the key to Kafka scalability, but that does not mean that you should have too many partitions. Several customers have too many partitions, which in turn consumes all resources from the server. Each partition in a topic uses a lot of RAM (file descriptors). The load on the CPU will also get higher with more partitions since Kafka needs to keep track of all of the partitions. More than 50 partitions for a topic are rarely recommended for Best Practice.

All CloudKarafka brokers have a very large number of file descriptors.

The balance between cores and consumers

Each partition in Kafka is single-threaded. Too many partitions will not reach its full potential if you have a low number of cores on the server. Therefore, it is important to try to keep a good balance between cores and consumers. It is not ideal to have consumers idling, due to fewer cores than consumers.

Kafka Broker

The more brokers in a cluster, the higher the performance delivered since the load is spread between all of your nodes. Replication is one of Kafka's strengths and a key component while deciding how many brokers to include in a cluster in the minimum in-sync replicas setting.

Minimum in-sync replicas

The minimum number of in-sync replicas specify how many replicas that are needed to be available for the producer to successfully send records to a partition. The number of replicas in your topic is specified by you when creating the topic. The number of replicas specified can be changed in the future.

A high number of minimum in-sync replicas gives a higher persistence, but on the other hand, might reduce availability because the minimum number of replicas given must be available before a publish. If you have a 3 node cluster and the minimum in-sync replicas are set to 3, and one node goes down, the other two nodes will not able to receive any data. Only care about the minimum number of in-sync replicas when it comes to the availability of your cluster and reliability guarantees.

The minimum number of in-sync replicas has nothing to do with the throughput. Setting the minimum number of in-sync replicas to larger than 1 may ensure less or no data loss, but throughput varies depending on the ack value configuration.

Default minimum in-sync replicas are set to 1 by default in CloudKarafka. This means that the minimum number of in-sync replicas that must be available for the producer to successfully send records to a partition must be 1.

Partition load between brokers

A common error is that load is not distributed equally between brokers. It's important to keep an eye on the partition distribution and do re-assignments to new brokers if needed, to ensure no broker is overloaded while another is idling.

The CloudKarafka MGMT interface will show a warning if or when a partition distribution is needed. The partition distribution is also simplified in the MGMT interface, you can simply press a button to distribute the data. The MGMT interface will check for existing partitions and spread the data between them. If you are not using the CloudKarafka MGMT, we recommend you to use the command-line tool to spread your data.

Do not hardcode partitions

Keys are used to determine the partition within a log to which a record is appended to. A common error is that the same key is used for many records, making every record end up on the same partition.

Make sure that you never hardcode the record key value.

Number of partitions

A higher number of partitions is preferable for high throughput in Kafka. Although, a high number of partitions will put more load on the machines and might affect the latency of messages. Consider the desired result and don't exaggerate.

Most customers of CloudKarafka have 3 nodes in the setup and the number of replicas set to 3. You can have as many replicas as you have nodes in your system.

Default created topic

When sending a record to a non-existent topic, the topic is created by default *auto.create.topics.enable* and is set to true by default in Apache Kafka.

This configuration can be changed so topics are not created if they do not exist. This configuration can be helpful in minimizing mistakes caused by misspelling or miscommunication between developers.

Default retention period

A record sent to a Kafka cluster is appended to the end of one of the logs. The record remains in the topic for a configurable period of time, until a configurable size is reached or until the specified retention for the topic exceeds. The message stays in the log, even if the record has been consumed.

The default retention period can be changed in the CloudKarafka MGMT interface.

Record order in Apache Kafka

One partition will guarantee an unchangeable sequence of your log. Two or more partitions will break the order, as the order is not guaranteed between partitions. Records sent within Apache Kafka can be strictly ordered, even though your setup contains more than one partition. You will achieve a strict order of records by setting up a consistent message key that sorts records in the order specified, for example, user-ID. This guarantees that all records from a specific user always end up in the same partition.

Please note that if the purpose of using Apache Kafka requires that all records must be ordered within one topic, then you have to use only one partition.
More on this topic can be found in this [blog post.](#)

Number of Zookeepers

Apache Zookeeper is a stand-alone, centralized service, acting across nodes to relieve Kafka from administrative duties. The *controller* in an Apache Kafka cluster is one of the brokers that has the additional duty of electing new partition leaders when the existing leader fails. There can be only one controller at a time and controller elections are coordinated by ZooKeeper.

Zookeeper requires a majority of servers to be functioning. If you, for example, have 5 servers in your cluster, you would need 3 servers to be up and running for Zookeeper to be working.
I.e., you can afford to lose one Zookeeper in a 3 node cluster, and you can afford to lose 2 Zookeeper in a 5 node cluster.

Apache Kafka server type

What you need when setting up a Kafka cluster is lots of memory. The data sent to the broker is always written to disk, but it also stays in the memory for as long as there is space to keep it in there. More memory will give a higher throughput since Kafka Consumers try to read data from the memory.

Kafka does not require high CPU, as long as there are not too many partitions running. A larger plan in CloudKarafka equals a larger disk and a longer retention period for log messages since you will not run out of disk space.

Performance optimization for Apache Kafka - Consumers

By refining the consumer setup, you can avoid common errors and ensure your configuration meets your expectations.

Consumers can read log messages from the broker, starting from a specific offset. Consumers are allowed to read from any offset point they choose. This allows consumers to join the cluster at any point in time.

A consumer can join a group, called a consumer group. A consumer group includes the set of consumer processes that are subscribing to a specific topic. Consumers in the group then divide the topic partitions fairly amongst themselves by establishing that each partition is only consumed by a single consumer from the group. For instance, each consumer in the group is assigned a set of partitions to subscribe from. Kafka guarantees that a message is only read by a single consumer in the group.

Consumer Connections

Make sure all consumers in a consumer group have a good connection. Partitions are redistributed between consumers every time a consumer connects or drop out of the consumer group. This means that consumers in the group are not able to consume records during this time. If one consumer in a group has a bad connection, the whole group is affected and will be unavailable during every reconnect. A redistribution of partitions takes around 2-3 seconds or longer. To make sure your setup is running smoothly, we strongly recommend securing the connection between your consumers.

Number of consumers

Ideally, the number of partitions should be equal to the number of consumers, however, this also depends on your use case.

- **Number of consumers > number of partitions**
  If the number of consumers is greater, some consumers will be idling, i.e., this means that you might be wasting client resources.

- **Number of partitions < number of consumers**
  Some consumers will read from multiple partitions if the number of partitions is greater than the number of consumers.

This is true in the scenario where there's a high-bandwidth event flow in the topic. But remember, not all use cases are like that. There are use cases where consumers would be mostly idling, like while waiting for user-triggered events. There could also be cases where having a single partition is completely fine, because of low event flow.

As mentioned before, the availability will be affected if one consumer has a bad connection. The more consumers you have, the larger the risk is that one might drop and halt all other consumers.

Apache Kafka and server concepts

Log

Write-ahead log, commit log, transaction log; Each partition in Apache Kafka is a log - a time-ordered, append-only sequence of data, from where data is removed only when a given retention period has been exceeded. Records are appended to the end of the log and can be read in order. The log can also be rewound and records can be skipped over for consumers to read from any point in the partition.
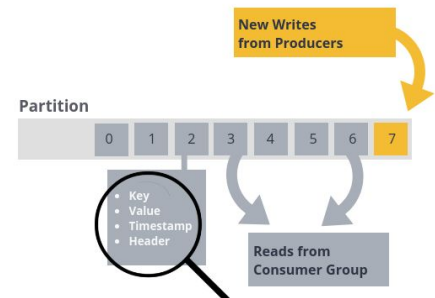
Record or Message

Data sent to and from the broker is called a record, a *key-value pair.* The record contains the topic name and partition number. The Kafka broker keeps records inside topic partitions.
A record (also called a message) represents information such as lines in a log file, a row of stock market data, an error message from a system or an event that is supposed to be handled.

Broker

The brokers in a Kafka cluster handle the process of receiving, storing and forwarding the records to the interested consumers.

A Kafka cluster consists of one or more servers called Kafka brokers. It's the processes or servers in Kafka that process the messages. The Kafka broker is mediating the conversation between different computer systems, like a queue in a message system.

Topics

Records are grouped into categories called topics. A Topic is a category/feed name to which records are stored and published. Example: LogMessage or StockMessage.

If you wish to send a record you send it to a specific topic and if you want to read a record you read it from a specific topic.

Retention period

Records published to the cluster will stay in the cluster until a configurable retention period has passed. Kafka retains all records for a set amount of time or until a configurable size is reached. The consumption time is not impacted by the size of the log.

Producer, Producer API

The processes that publish records into a topic are called producers and are using the producer API.

Consumer, Consumer API

The processes that consume records from a topic are called consumers and are using the consumer API.

Partition

Topics are divided into one or more partitions, which can be replicated between nodes. Partitions are the unit of parallelism in Kafka. Partitions allow records in a topic to be distributed to multiple brokers. A topic can have any number of partitions.

Offset

Kafka topics are divided into a number of partitions, which contain records in an unchangeable sequence. Each record in a partition is assigned and identified by its unique offset.

Consumer group

A consumer group includes the set of consumers that are subscribing to a specific topic. Kafka consumers are usually a part of a consumer group. Each consumer in the group is assigned a set of partitions, from which they are able to consume messages. Each consumer in the group will receive records from different subsets of the partitions in the topic.

ZooKeeper

Zookeeper is a stand-alone, centralized service, acting across nodes to relieve Kafka from administrative duties. Zookeeper is responsible for controller elections, the configuration of topics, handling access control lists and cluster memberships.

Instance ("As in a CloudKarafka instance")

When a CloudKarafka plan is created, you get what we call CloudKarafka instance or an instance of Apache Kafka. It could be a dedicated instance, an Apache Kafka broker, or a shared instance, which gives you five dedicated topics on a shared plan.

Replication, replicas

Replication is the process of copying records from the leader replica to follower replicas. Followers periodically (every 500ms by default) send fetch requests to the leader who then responds with a batch of records. Replication in Apache Kafka happens at the partition level.