1. What Is Apache Kafka?

Apache Kafka is an event streaming platform used to collect, process, store, and integrate data at scale. It has numerous use cases including distributed logging, stream processing, data integration, and pub/sub messaging.

What Are Events?

An event is any type of action, incident, or change that's identified or recorded by software or applications. For example,

- a payment,
- a website click, or
- a temperature reading, etc.

In other words, an event is a combination of notification—the element of whenness that can be used to trigger some other activity—and state. That state is usually fairly small, say less than a megabyte or so, and is normally represented in some structured format, say in JSON or an object serialized with Apache Avro™ or Protocol Buffers. (A data serialization and exchange framework)

Kafka and Events – Key/Value Pairs

Kafka is based on the abstraction of a distributed commit log. By splitting a log into partitions, Kafka is able to scale-out systems.

As such, Kafka models events as key/value pairs. Internally, keys and values are just sequences of bytes, but externally in your programming language of choice, they are often structured objects represented in your language's type system. Kafka famously calls the translation between language types and internal bytes serialization and deserialization. The serialized format is usually JSON, JSON Schema, Avro, or Protobuf.

Values are typically the serialized representation of an application domain object or some form of raw message input, like the output of a sensor.

Keys can also be complex domain objects but are often primitive types like strings or integers. The key part of a Kafka event is not necessarily a unique identifier for the event, like the primary key of a row in a relational database would be. It is more likely the identifier of some entity in the system, like a user, order, or a particular connected device.

2. Kafka Topics

Apache Kafka's most fundamental unit of organization is the topic, which is something like a table in a relational database. You create different topics to hold different kinds of events and different topics to hold filtered and transformed versions of the same kind of event.

A topic is a log of events (The logs that underlie Kafka topics are files stored on disk). Logs are easy to understand, because they are simple data structures with well-known semantics.

- First, they are appended only: When you write a new message into a log, it always goes on the end.
- Second, they can only be read by seeking an arbitrary offset in the log, then by scanning sequential log entries.
- Third, events in the log are immutable—once something has happened, it is exceedingly difficult to make it unhappen. The simple semantics of a log make it feasible for Kafka to deliver high levels of sustained throughput in and out of topics.

Since Kafka topics are logs. Every topic can be configured to expire data after it has reached a certain age (or the topic overall has reached a certain size), from as short as seconds to as long as years or even to retain messages indefinitely. When you write an event to a topic, it is as durable as it would be if you had written it to any database you ever trusted.

3. Kafka Partitioning

If a topic were constrained to live entirely on one machine, that would place a pretty radical limit on the ability of Apache Kafka to scale.

It could manage many topics across many machines—Kafka is a distributed system, after all—but no one topic could ever get too big or aspire to accommodate too many reads and writes. Fortunately, Kafka gives us the ability to partition topics.

Partitioning takes the single topic log and breaks it into multiple logs, each of which can live on a separate node in the Kafka cluster. This way, the work of storing messages, writing new messages, and processing existing messages can be split among many nodes in the cluster.

How Partitioning Works

Having broken a topic up into partitions, we need a way of deciding which messages to write to which partitions. Typically,

- if a message has no key, subsequent messages will be distributed round-robin among all the topic's partitions. In this case, all partitions get an even share of the data, but we don't preserve any kind of ordering of the input messages.
- If the message does have a key, then the destination partition will be computed from a hash of the key. This allows Kafka to guarantee that messages having the same key always land in the same partition, and therefore are always in order.

For example, if you are producing events that are all associated with the same customer, using the customer ID as the key guarantees that all of the events from a given customer will always arrive in order. It is often worth it in order to preserve the ordering of keys.

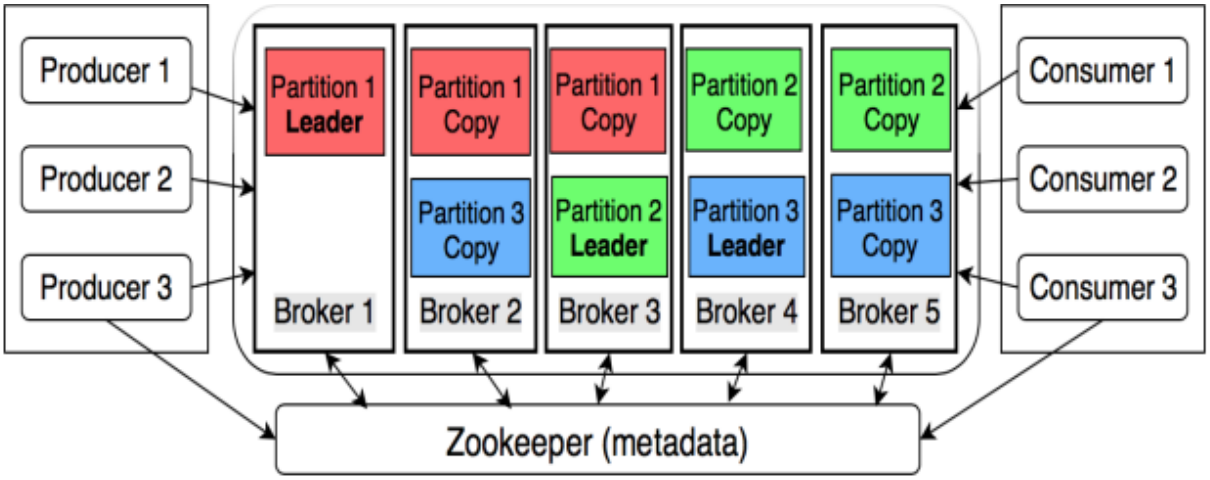
4. Kafka Brokers

Apache Kafka is composed of a network of machines called brokers. In a contemporary deployment, these may not be separate physical servers but containers running on pods running on virtualized servers running on actual processors in a physical datacenter somewhere. However they are deployed, they are independent machines each running the Kafka broker process. Each broker hosts some set of partitions and handles incoming requests to write new events to those partitions or read events from them. Brokers also handle replication of partitions between each other.

5. Replication

It would not do if we stored each partition on only one broker. Whether brokers are bare metal servers or managed containers, they and their underlying storage are susceptible to failure, so we need to copy partition data to several other brokers to keep it safe. Those copies are called follower replicas, whereas the main partition is called the leader replica. When you produce data to the leader—in general, reading and writing are done to the leader—the leader and the followers work together to replicate those new writes to the followers.

This happens automatically, and while you can tune some settings in the producer to produce varying levels of durability guarantees, this is not usually a process you have to think about as a developer building systems on Kafka. All you really need to know as a developer is that your data is safe, and that if one node in the cluster dies, another will take over its role.



Visualization example

- https://softwaremill.com/kafka-visualisation/