What Is Apache Kafka?

Developed as a publish-subscribe messaging system to handle mass amounts of data at LinkedIn, today, Apache Kafka® is an open-source distributed event streaming platform used by over 80% of the Fortune 100.

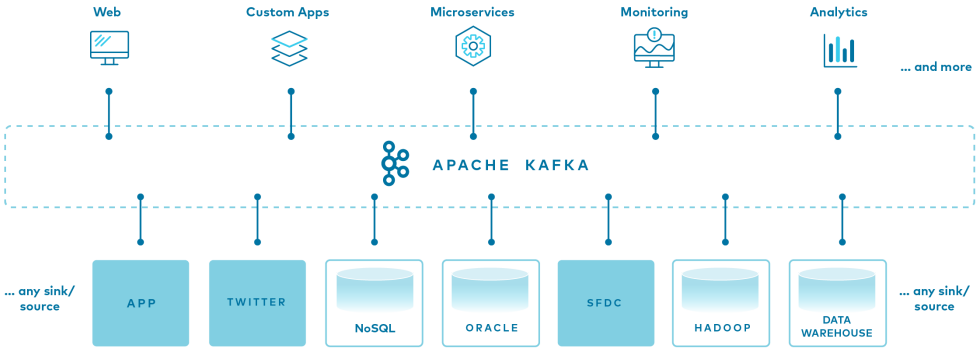
This beginner's Kafka tutorial will help you learn Kafka, its benefits, and use cases, and how to get started from the ground up. It includes a look at Kafka architecture, core concepts, and the connector ecosystem.

Introduction

➡ *Try this Apache Kafka hands-on tutorial*

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

In order to make complete sense of what Kafka does, we'll delve into what an event streaming platform is and how it works. So before delving into Kafka architecture or its core components, let's discuss what an event is. This will help explain how Kafka stores events, how to get events in and out of the system, and how to analyze event streams.



What Is an Event?

Events explained on video

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, along with a description of what happened.

In other words, an event is a combination of notification—the element of when-ness 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.

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.

This may not sound so significant now, but we'll see later on that keys are crucial for how Kafka deals with things like parallelization and data locality.

Why Kafka? Benefits and Use Cases

Kafka is used by over 100,000 organizations across the world and is backed by a thriving community of professional developers, who are constantly advancing the state of the art in stream processing together. Due to Kafka's high throughput, fault tolerance, resilience, and scalability, there are numerous use cases across almost every industry - from banking and fraud detection, to transportation and IoT. We typically see Kafka used for purposes like those below.

Data Integration

➡ Learn more: Data Pipelines free course

Kafka can connect to nearly any other data source in traditional enterprise information systems, modern databases, or in the cloud. It forms an efficient point of integration with built-in data connectors, without hiding logic or routing inside brittle, centralized infrastructure.

Metrics and Monitoring

Kafka is often used for monitoring operational data. This involves aggregating statistics from distributed applications to produce centralized feeds with real-time metrics.

Log Aggregation

A modern system is typically a distributed system, and logging data must be centralized from the various components of the system to one place. Kafka often serves as a single source of truth by centralizing data across all sources, regardless of form or volume.

Stream Processing

➡ Learn more: Kafka Streams and ksqlDB free courses

Performing real-time computations on event streams is a core competency of Kafka. From real-time data processing to dataflow programming, Kafka ingests, stores, and processes streams of data as it's being generated, at any scale.

Publish-Subscribe Messaging

As a distributed pub/sub messaging system, Kafka works well as a modernized version of the traditional message broker. Any time a process that generates events must be decoupled from the process or from processes receiving the events, Kafka is a scalable and flexible way to get the job done.

Kafka Architecture – Fundamental Concepts

Kafka Topics

Topics explained on video

Events have a tendency to proliferate—just think of the events that happened to you this morning—so we'll need a system for organizing them. Kafka's most fundamental unit of organization is the topic, which is something like a table in a relational database. As a developer using Kafka, the topic is the abstraction you probably think the most about. 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. Logs are easy to understand, because they are simple data structures with well-known semantics. First, they are append 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 un-happen. The simple semantics of a log make it feasible for Kafka to deliver high levels of sustained throughput in and out of topics, and also make it easier to reason about the replication of topics, which we'll cover more later.

Logs are also fundamentally durable things. Traditional enterprise messaging systems have topics and queues, which store messages temporarily to buffer them between source and destination.

Since Kafka topics are logs, there is nothing inherently temporary about the data in them. 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. The logs that underlie Kafka topics are files stored on disk. 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.

The simplicity of the log and the immutability of the contents in it are key to Kafka's success as a critical component in modern data infrastructure—but they are only the beginning.

Kafka Partitioning

Partitioning explained on video

If a topic were constrained to live entirely on one machine, that would place a pretty radical limit on the ability of 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 does not leave us without options here: It 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 Kafka 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. This creates the possibility that a very active key will create a larger and more active partition, but this risk is small in practice and is manageable when it presents itself. It is often worth it in order to preserve the ordering of keys.

Kafka Brokers

Brokers explained on video

So far we have talked about events, topics, and partitions, but as of yet, we have not been too explicit about the actual computers in the picture. From a physical infrastructure standpoint, 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.

Replication

Replication explained on video

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 replica, 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.

Client Applications

Now let's get outside of the Kafka cluster itself to the applications that use Kafka: the producers and consumers. These are client applications that contain your code, putting messages into topics and reading messages from topics. Every component of the Kafka platform that is not a Kafka broker is, at bottom, either a producer or a consumer or both. Producing and consuming are how you interface with a cluster.

Kafka Producers

Producers explained on video

The API surface of the producer library is fairly lightweight: In Java, there is a class called KafkaProducer that you use to connect to the cluster. You give this class a map of configuration parameters, including the address of some brokers in the cluster, any appropriate security configuration, and other settings that determine the network behavior of the producer. There is another class called ProducerRecord that you use to hold the key-value pair you want to send to the cluster.

To a first-order approximation, this is all the API surface area there is to producing messages. Under the covers, the library is managing connection pools, network buffering, waiting for brokers to acknowledge messages, retransmitting messages when necessary, and a host of other details no application programmer need concern herself with.

➡ Get started writing a Kafka application with step-by-step guide and code samples

Kafka Consumers

Using the consumer API is similar in principle to the producer. You use a class called KafkaConsumer to connect to the cluster (passing a configuration map to specify the address of the cluster, security, and other parameters). Then you use that connection to subscribe to one or more topics. When messages are available on those topics, they come back in a collection called ConsumerRecords, which contains individual instances of messages in the form of ConsumerRecord objects. A ConsumerRecord object represents the key/value pair of a single Kafka message.

KafkaConsumer manages connection pooling and the network protocol just like KafkaProducer does, but there is a much bigger story on the read side than just the network plumbing. First of all, Kafka is different from legacy message queues in that reading a message does not destroy it; it is still there to be read by any other consumer that might be interested in it. In fact, it's perfectly normal in Kafka for many consumers to read from one topic. This one small fact has a positively disproportionate impact on the kinds of software architectures that emerge around Kafka, which is a topic covered very well elsewhere.

Also, consumers need to be able to handle the scenario in which the rate of message consumption from a topic combined with the computational cost of processing a single message are together too high for a single instance of the application to keep up. That is, consumers need to scale. In Kafka, scaling consumer groups is more or less automatic.

➡ Get started writing a Kafka application with step-by-step guide and code samples

Kafka Components and Ecosystem

If all you had were brokers managing partitioned, replicated topics with an ever-growing collection of producers and consumers writing and reading events, you would actually have a pretty useful system. However, the experience of the Kafka community is that certain patterns will emerge that will encourage you and your fellow developers to build the same bits of functionality over and over again around core Kafka.

You will end up building common layers of application functionality to repeat certain undifferentiated tasks. This is code that does important work but is not tied in any way to the business you're actually in. It doesn't contribute value directly to your customers. It's infrastructure, and it should be provided by the community or by an infrastructure vendor.

It can be tempting to write this code yourself, but you should not. Kafka Connect, the Confluent Schema Registry, Kafka Streams, and ksqlDB are examples of this kind of infrastructure code. We'll take a look at each of them in turn.

Kafka Connect

In the world of information storage and retrieval, some systems are not Kafka. Sometimes you would like the data in those other systems to get into Kafka topics, and sometimes you would like data in Kafka topics to get into those systems. As Apache Kafka's integration API, this is exactly what Kafka Connect does.

What Does Kafka Connect Do?

➡ Learn more: Data Pipelines free course

On the one hand, Kafka Connect is an ecosystem of pluggable connectors, and on the other, a client application. As a client application, Connect is a server process that runs on hardware independent of the Kafka brokers themselves. It is scalable and fault tolerant, meaning you can run not just one single Connect worker but a cluster of Connect workers that share the load of moving data in and out of Kafka from and to external systems. Kafka Connect also abstracts the business of code away from the user and instead requires only JSON configuration to run. For example, here's how you'd stream data from Kafka to Elasticsearch:

Copy

```
{
  "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
  "topics"         : "my_topic",
  "connection.url" : "http://elasticsearch:9200",
  "type.name"      : "_doc",
  "key.ignore"     : "true",
  "schema.ignore"  : "true"
}
```

Benefits of Kafka Connect

➡ Learn more: Kafka Connect 101 free course

One of the primary advantages of Kafka Connect is its large ecosystem of connectors. Writing the code that moves data to a cloud blob store, or writes to Elasticsearch, or inserts records into a relational database is code that is unlikely to vary from one business to the next. Likewise, reading from a relational database, Salesforce, or a legacy HDFS filesystem is the same operation no matter what sort of application does it. You can definitely write this code, but spending your time doing that doesn't add any kind of unique value to your customers or make your business more uniquely competitive.

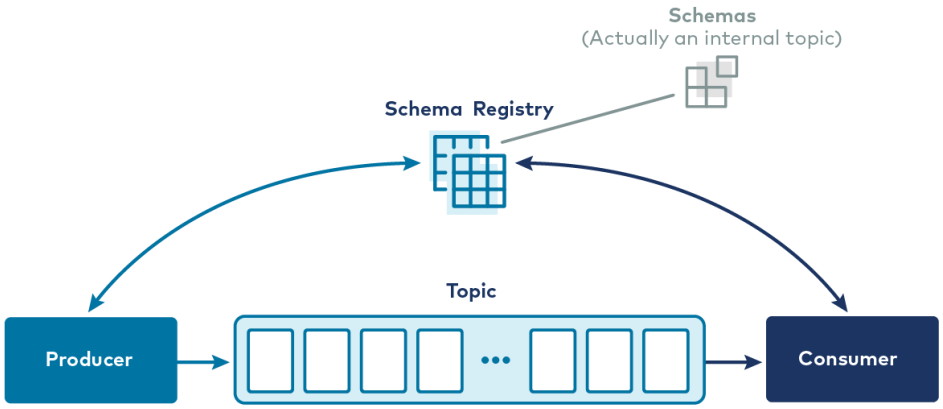
All of these are examples of Kafka connectors available in the Confluent Hub, a curated collection of connectors of all sorts and most importantly, all licenses and levels of support. Some are commercially licensed and some can be used for free. Connect Hub lets you search for source and sink connectors of all kinds and clearly shows the license of each connector. Of course, connectors need not come from the Hub and can be found on GitHub or elsewhere in the marketplace. And if after all that you still can't find a connector that does what you need, you can write your own using a fairly simple API.

Now, it might seem straightforward to build this kind of functionality on your own: If an external source system is easy to read from, it would be easy enough to read from it and produce to a destination topic. If an external sink system is easy to write to, it would again be easy enough to consume from a topic and write to that system. But any number of complexities arise,

including how to handle failover, horizontally scale, manage commonplace transformation operations on inbound or outbound data, distribute common connector code, configure and operate this through a standard interface, and more.

Connect seems deceptively simple on its surface, but it is in fact a complex distributed system and plugin ecosystem in its own right. And if that plugin ecosystem happens not to have what you need, the open-source Connect framework makes it simple to build your own connector and inherit all the scalability and fault tolerance properties Connect offers.

Schema Registry



Once applications are busily producing messages to Kafka and consuming messages from it, two things will happen. First, new consumers of existing topics will emerge. These are brand new applications—perhaps written by the team that wrote the original producer of the messages, perhaps by another team—and will need to understand the format of the messages in the topic. Second, the format of those messages will evolve as the business evolves. Order objects gain a new status field, usernames split into first and last name from full name, and so on. The schema of our domain objects is a constantly moving target, and we must have a way of agreeing on the schema of messages in any given topic.

Confluent Schema Registry exists to solve this problem.

What Is Schema Registry?

Schema Registry is a standalone server process that runs on a machine external to the Kafka brokers. Its job is to maintain a database of all of the schemas that have been written into topics in the cluster for which it is responsible. That "database" is persisted in an internal Kafka topic and cached in the Schema Registry for low-latency access. Schema Registry can be run in a redundant, high-availability configuration, so it remains up if one instance fails.

Schema Registry is also an API that allows producers and consumers to predict whether the message they are about to produce or consume is compatible with previous versions. When a producer is configured to use the Schema Registry, it calls an API at the Schema Registry REST endpoint and presents the schema of the new message. If it is the same as the last message produced, then the produce may succeed. If it is different from the last message but matches the compatibility rules defined for the topic, the produce may still succeed. But if it is different in a way that violates the compatibility rules, the produce will fail in a way that the application code can detect.

Likewise on the consume side, if a consumer reads a message that has an incompatible schema from the version the consumer code expects, Schema Registry will tell it not to consume the message. Schema Registry doesn't fully automate the problem of schema evolution—that is a challenge in any system regardless of the tooling—but it does make a difficult problem much easier by keeping runtime failures from happening when possible.

Looking at what we've covered so far, we've got a system for storing events durably, the ability to write and read those events, a data integration framework, and even a tool for managing evolving schemas. What remains is the purely computational side of stream processing.

Kafka Streams

➡ Learn more: Kafka Streams free course

In a growing Kafka-based application, consumers tend to grow in complexity. What might have started as a simple stateless transformation (e.g., masking out personally identifying information or changing the format of a message to conform with internal schema requirements) soon evolves into complex aggregation, enrichment, and more. If you recall the consumer code we looked at up above, there isn't a lot of support in that API for operations like those: You're going to have to build a lot

of framework code to handle time windows, late-arriving messages, lookup tables, aggregation by key, and more. And once you've got that, recall that operations like aggregation and enrichment are typically stateful.

That "state" is going to be memory in your program's heap, which means it's a fault tolerance liability. If your stream processing application goes down, its state goes with it, unless you've devised a scheme to persist that state somewhere. That sort of thing is fiendishly complex to write and debug at scale and really does nothing to directly make your users' lives better. This is why Apache Kafka provides a stream processing API. This is why we have Kafka Streams.

What Is Kafka Streams?

Kafka Streams is a Java API that gives you easy access to all of the computational primitives of stream processing: filtering, grouping, aggregating, joining, and more, keeping you from having to write framework code on top of the consumer API to do all those things. It also provides support for the potentially large amounts of state that result from stream processing computations. If you're grouping events in a high-throughput topic by a field with many unique values then computing a rollup over that group every hour, you might need to use a lot of memory.

Indeed, for high-volume topics and complex stream processing topologies, it's not at all difficult to imagine that you'd need to deploy a cluster of machines sharing the stream processing workload like a regular consumer group would. The Streams API solves both problems by handling all of the distributed state problems for you: It persists state to local disk and to internal topics in the Kafka cluster, and it automatically reassigns state between nodes in a stream processing cluster when adding or removing stream processing nodes to the cluster.

In a typical microservice, stream processing is a thing the application does in addition to other functions. For example, a shipment notification service might combine shipment events with events in a product information changelog containing customer records to produce shipment notification objects, which other services might turn into emails and text messages. But that shipment notification service might also be obligated to expose a REST API for synchronous key lookups by the mobile app or web front end when rendering views that show the status of a given shipment.

The service is reacting to events—and in this case, joining three streams together, and perhaps doing other windowed computations on the joined result—but it is also servicing HTTP requests against its REST endpoint, perhaps using the Spring Framework or Micronaut or some other Java API in common use. Because Kafka Streams is a Java library and not a set of dedicated infrastructure components that do stream processing and only stream processing, it's trivial to stand up services that use other frameworks to accomplish other ends (like REST endpoints) and sophisticated, scalable, fault-tolerant stream processing.

Learn About Kafka with More Free Courses and Tutorials

- Learn stream processing in Kafka with the Kafka Streams course
- Get started with Kafka Connectors in the Kafka Connect course
- Check out Michael Noll's four-part series on Streams and Tables in Apache Kafka
- Listen to the podcast about Knative 101: Kubernetes and Serverless Explained with Jacques Chester