Kafka

What is Event Streaming

event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Where is event streaming used

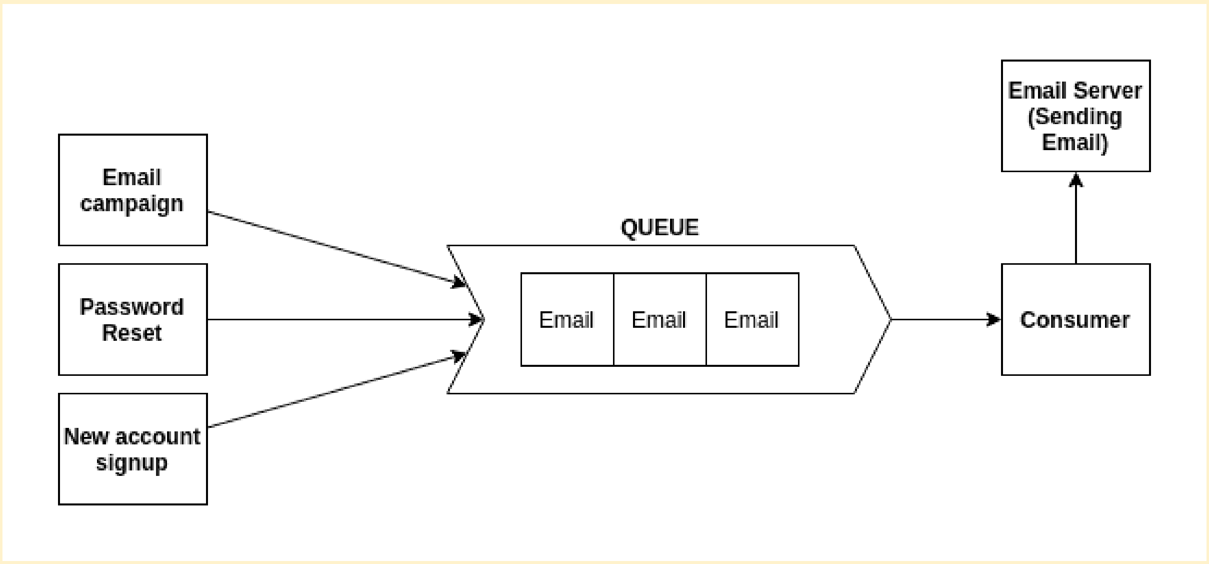
Event streaming is applied to a wide variety of use cases across a plethora of industries and organizations. Its many examples include:

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.
- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.
- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.

Apache Kafka was originated at LinkedIn and later became an open sourced Apache project in 2011, then First-class Apache project in 2012. Kafka is written in Scala and Java. Apache Kafka is publish-subscribe based fault tolerant messaging system. It is fast, scalable and distributed by design.

Why we need messaging system

A messaging system is responsible for transferring data from one application to another so the applications can focus on data without getting bogged down on data transmission and sharing. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system. There are two types of messaging patterns. The first one is a point-to-point and the other one is “publish–subscribe” (pub-sub) messaging system. Most of the messaging systems follow the pub-sub pattern.

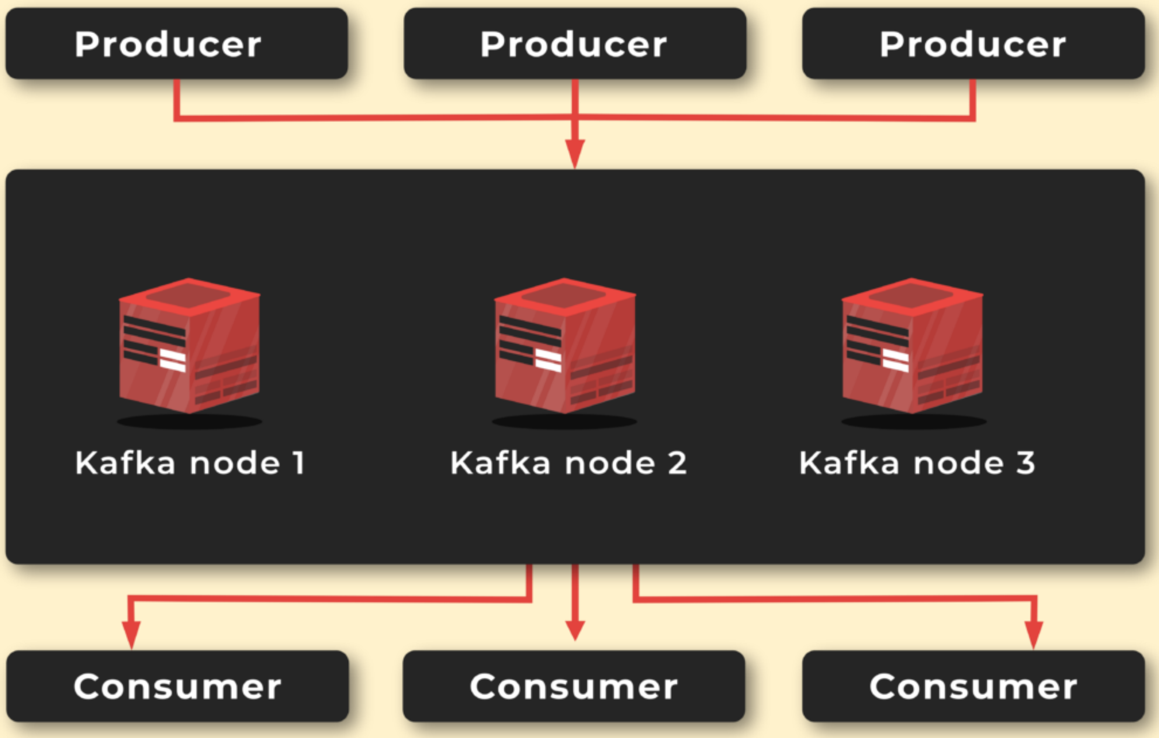


Kafka Use Cases

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications.

In our experience messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides.



Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many activity messages are generated for each user page view.

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Log Aggregation

Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off servers and puts them in a central place (a file server or HDFS perhaps) for processing. Kafka abstracts away the details of files and gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption. In comparison to log-centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication, and much lower end-to-end latency.

Stream Processing

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called Kafka Streams is available in Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include Apache Storm and Apache Samza.

Event Sourcing

Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log compaction feature in Kafka helps support this usage. In this usage Kafka is similar to Apache BookKeeper project.

Kafka Architecture

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

What is an Event

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

What are Producers and Consumers

Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

What are topics

- Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder
- Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events.
- Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded.
- Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.
- Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement

of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time.

- When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.
- To make your data fault-tolerant and highly-available, every topic can be **replicated**, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on.

What is Kafka Cluster

In a Distributed Computing System, a Cluster is a collection of computers working together to achieve a shared goal. A Kafka cluster is a system that consists of several Brokers, Topics, and Partitions for both.

The key objective is to distribute workloads equally among replicas and Partitions. Kafka Clusters Architecture mainly consists of the following 5 components:

- Topics
- Broker
- ZooKeeper
- Producers
- Consumers

Topic in detail

Apache 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.

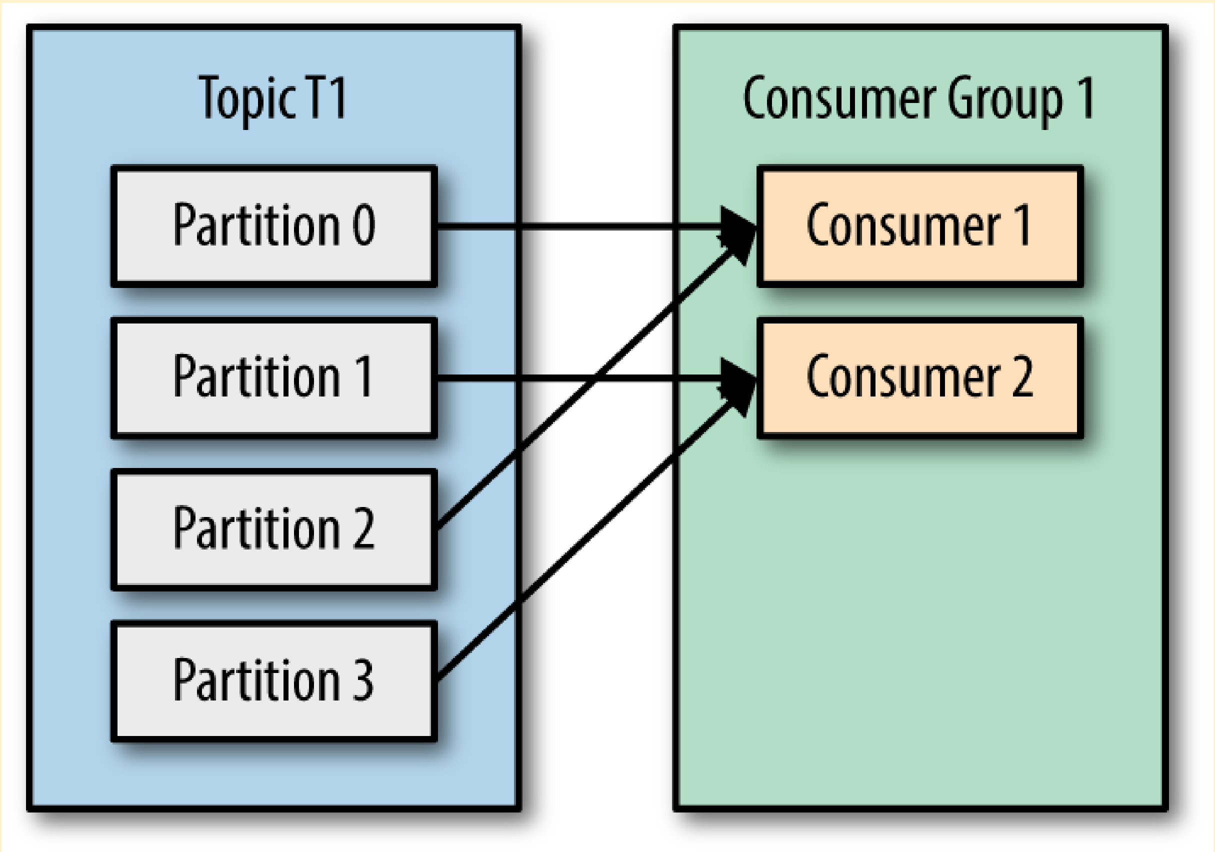
Logs are also fundamentally durable things. Traditional enterprise messaging systems have topics and queues, which store messages temporarily to buffer them between source and destination.

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.

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 does not leave us without options here: It gives us the ability to partition topics.

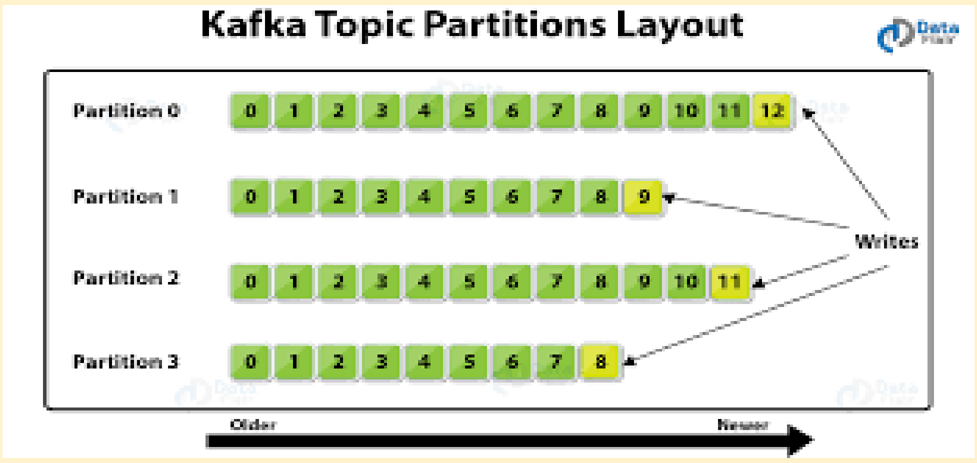
Partitioning

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.



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.



What is a Cluster

In a Distributed Computing System, a Cluster is a collection of computers working together to achieve a shared goal. A Kafka cluster is a system that consists of several Brokers, Topics, and Partitions for both.

What is a Broker

The Kafka Server is known as Broker, which is in charge of the **Topic's Message Storage**. Each of the Kafka Clusters comprises more than one Kafka Broker to maintain load balance. However, since they are stateless, ZooKeeper is used to preserve the Kafka Clusters state.

It's usually a good idea to consider Topic replication when constructing a Kafka system. As a result, if a Broker goes down, its Topics' duplicates from another Broker can fix the situation.

A Topic with a Kafka Replication Factor of 2 will have one additional copy in a separate Broker. Further, the replication factor cannot exceed the entire number of Brokers accessible.

The role of Zookeeper

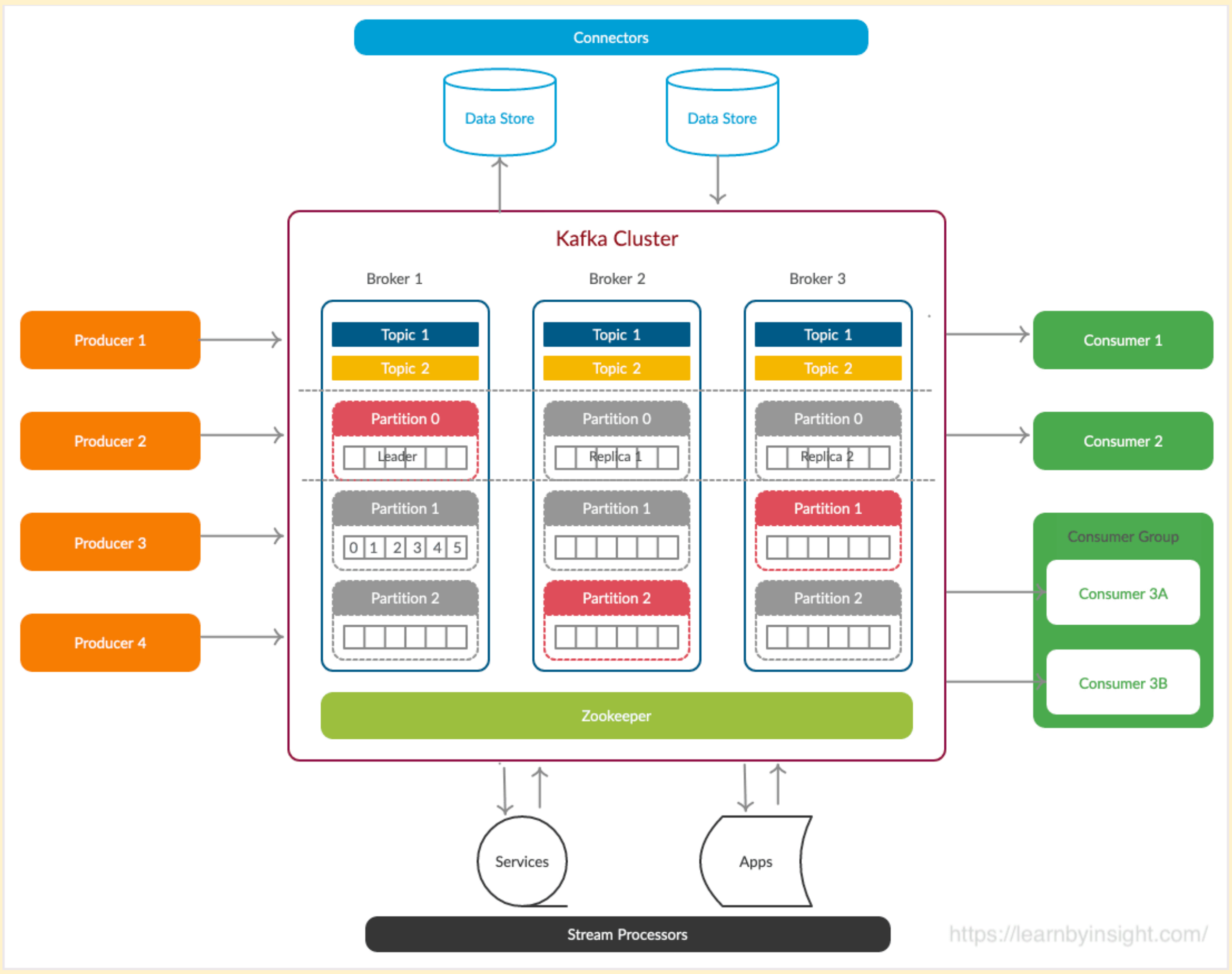
The Consumer Clients' details and Information about the Kafka Clusters are stored in a ZooKeeper. It acts like a **Master Management Node** where it is in charge of managing and maintaining the Brokers, Topics, and Partitions of the Kafka Clusters.

The Zookeeper keeps track of the Brokers of the Kafka Clusters. It determines which Brokers have crashed and which Brokers have just been added to the Kafka Clusters, as well as their lifetime.

Then, it notifies the Producer or Consumers of Kafka queues about the state of Kafka Clusters. This facilitates the coordination of work with active Brokers for both Producers and Consumers.

Zookeeper also keeps track of which Broker is the subject Partition's Leader and gives that information to the Producer or Consumer so they may read and write messages

Complete Kafka Architecture



//create a topic

$ bin/kafka-topics.sh--create--topic my-topic--bootstrap-server localhost:9092--partitions 3--replication-factor 1

//view details of topic

$ bin/kafka-topics.sh--bootstrap-server=localhost:9092--describe--topic my-topic

//list topics

```
$ bin/kafka-topics.sh--list--zookeeper localhost:2181
```

start a producer

```
$ bin/kafka-console-producer.sh--topic my-topic--bootstrap-server
localhost:9092
```

start a consumer

```
$ bin/kafka-console-consumer.sh--topic my-topic--from-beginning--bootstrap-
server localhost:9092
```

change retention period(default 168 hours or 7 days)

```
$ bin/kafka-topics.sh--zookeeper=localhost:2181--alter--topic my-topic--config
retention.ms=300000
```

```
/usr/bin/kafka-configs--bootstrap-server localhost:9092--entity-type topics--
entity-name mf-event-avro-enriched--alter--add-config
retention.ms=172800000
```