Apache KAFKA

A High throughput Distributed Messaging System

2CEIT702: Big Data Analytics

Introduction

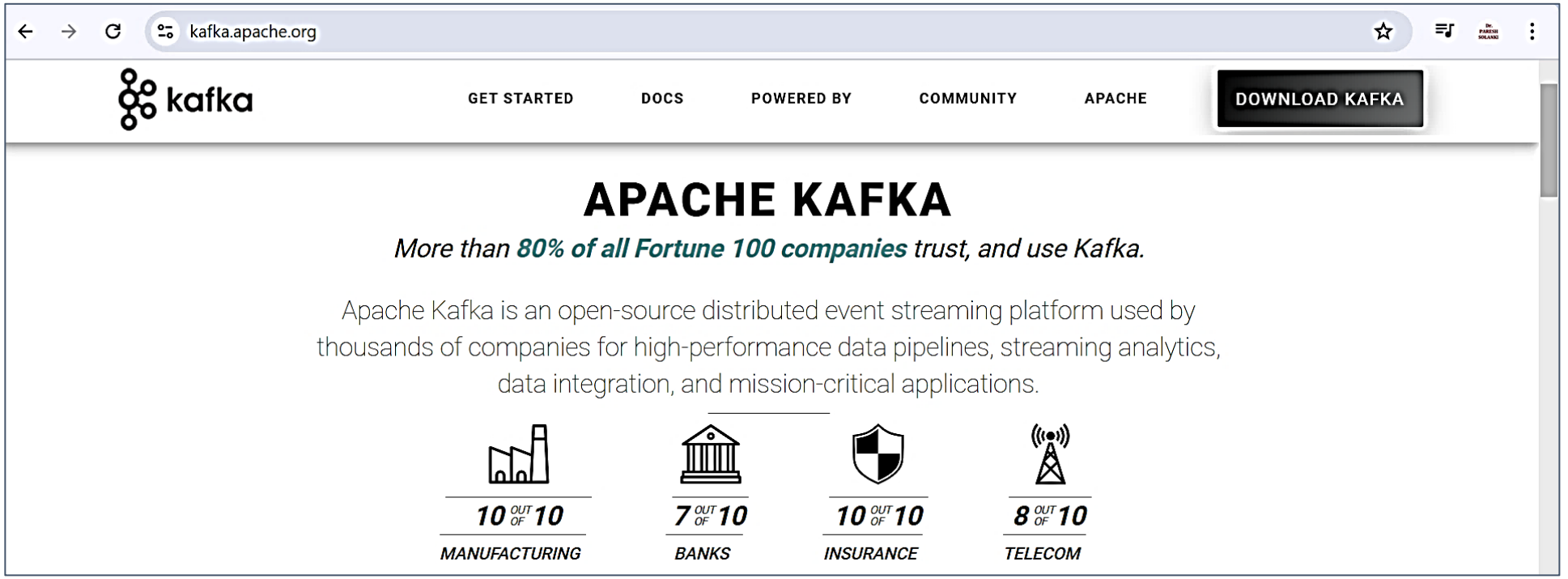
- In Big Data, we have two main challenges.
  - How to collect large volume of data?
  - How to analyze the collected data?

- How to handle large volumes of real-time data streams efficiently and reliably?

- Example: Zomato, Ola/Uber

- To overcome above challenges, you must need a messaging system.

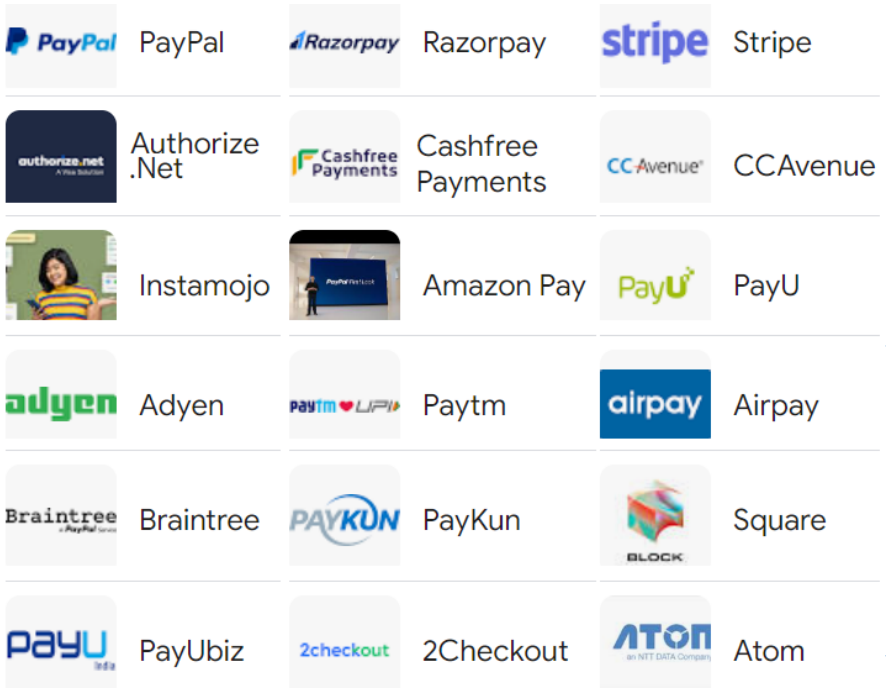
Introduction



- Kafka is a open source **distributed** **Message/Event streaming** **platform** that uses publish and subscribe mechanism to stream the records.
- Originally developed by LinkedIn and later donated to Apache foundation.
- Currently, used by many big companies.

Introduction

- **Message/Event Streaming:** There are two types of task associated with event streaming:
  - Creating real time streaming
  - Processing real time streaming



Multiple applications used by millions of users

Creating real time Stream data

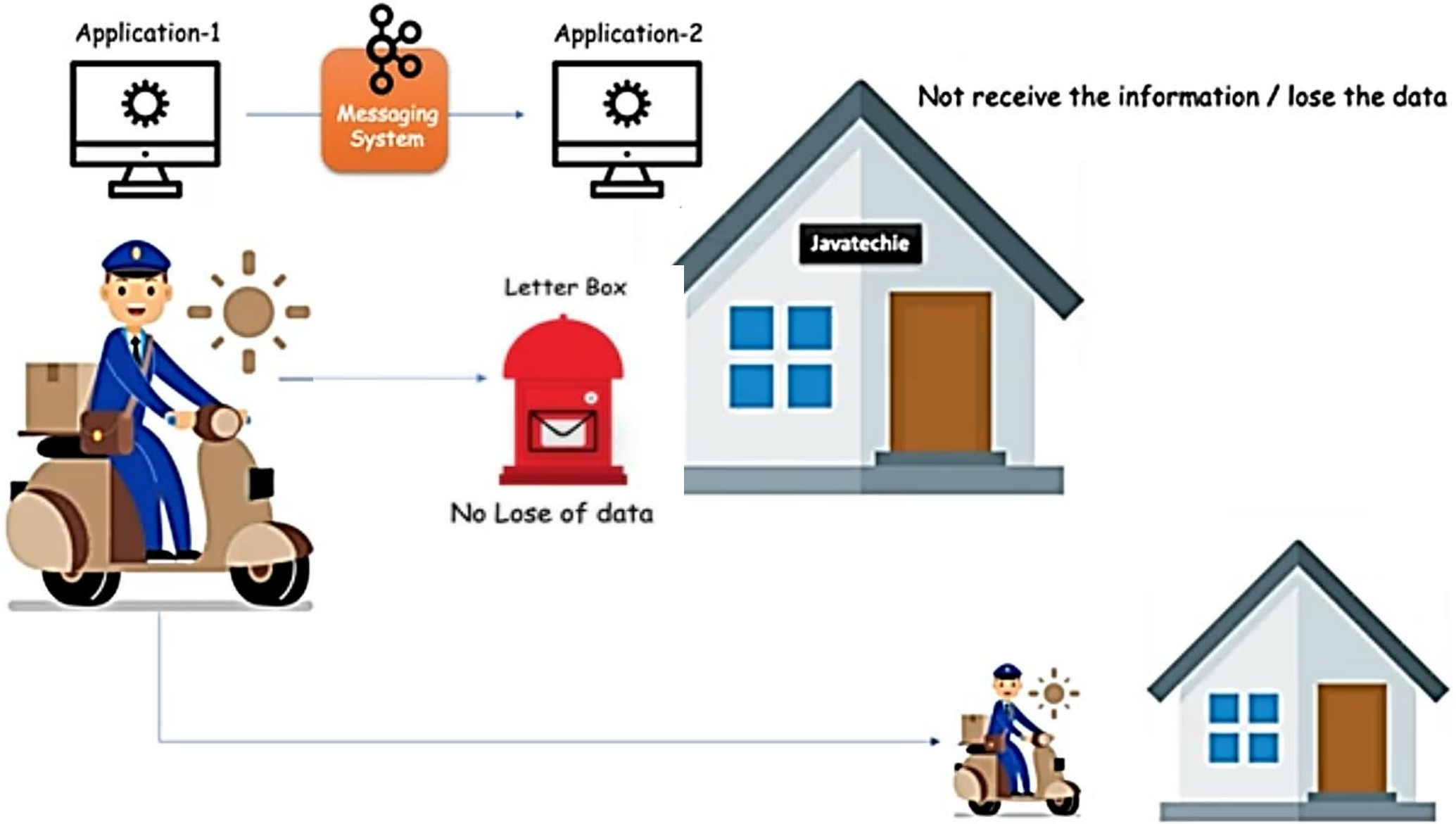
Kafka

Kafka (Distributed)

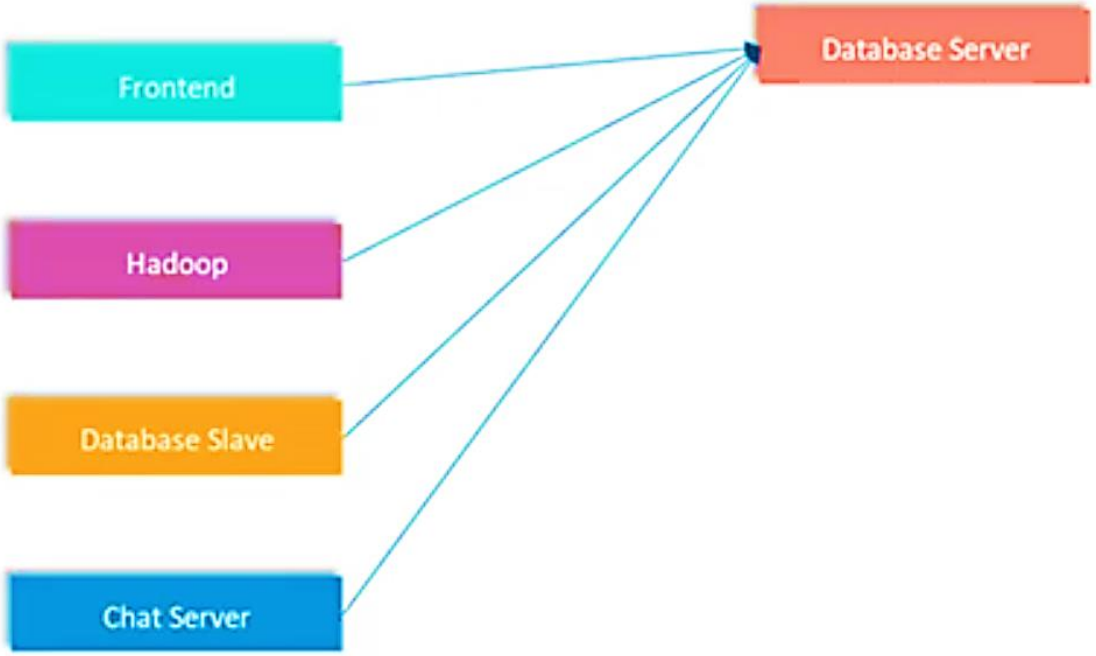
Processing real time Stream data

Check: Only 10 transactions are allowed per Day

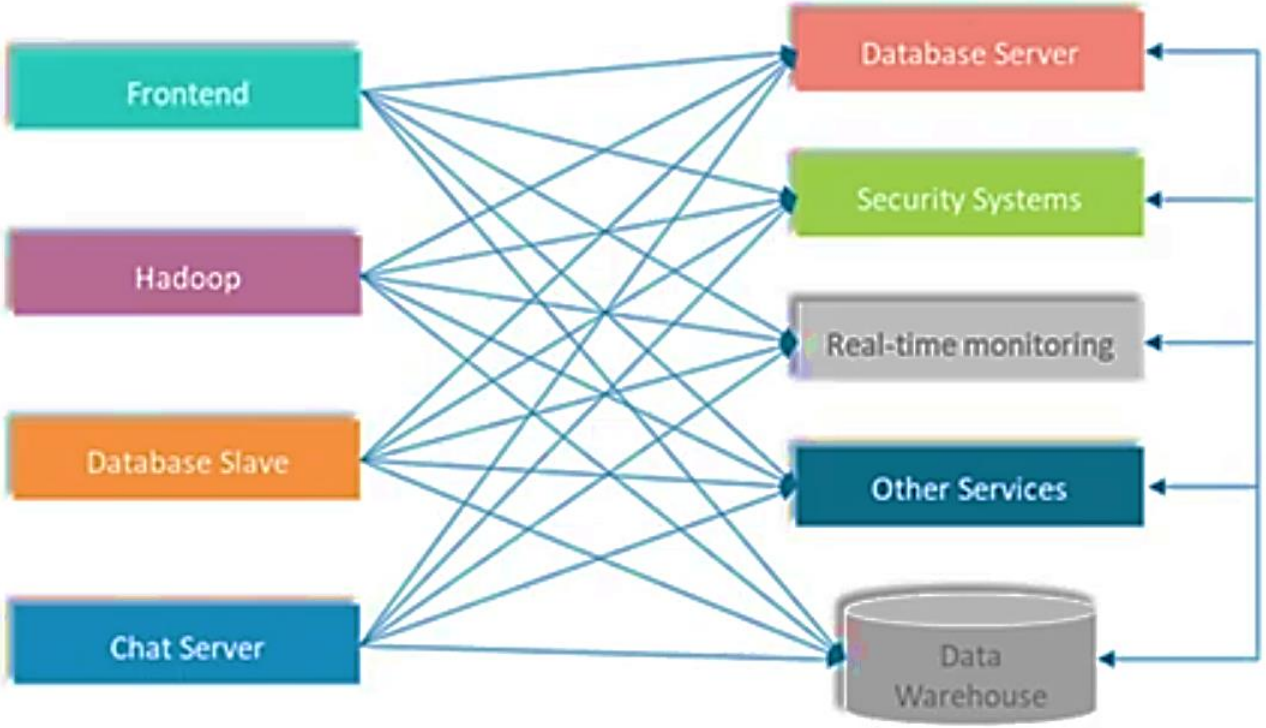
Why do we need kafka? (Simple Example)



Why do we need kafka?



No Problem at all.

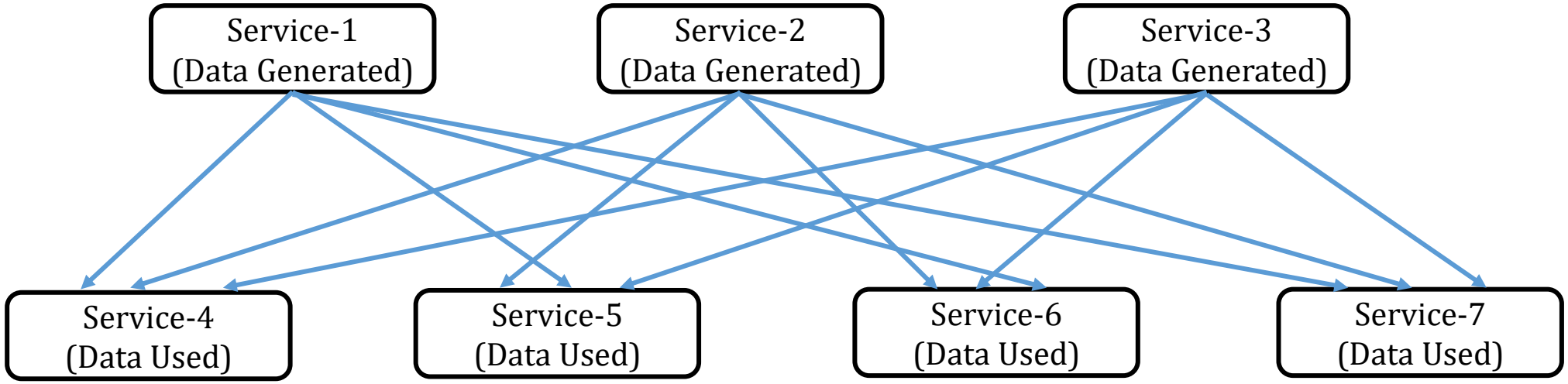


Problems raised when multiple services used.

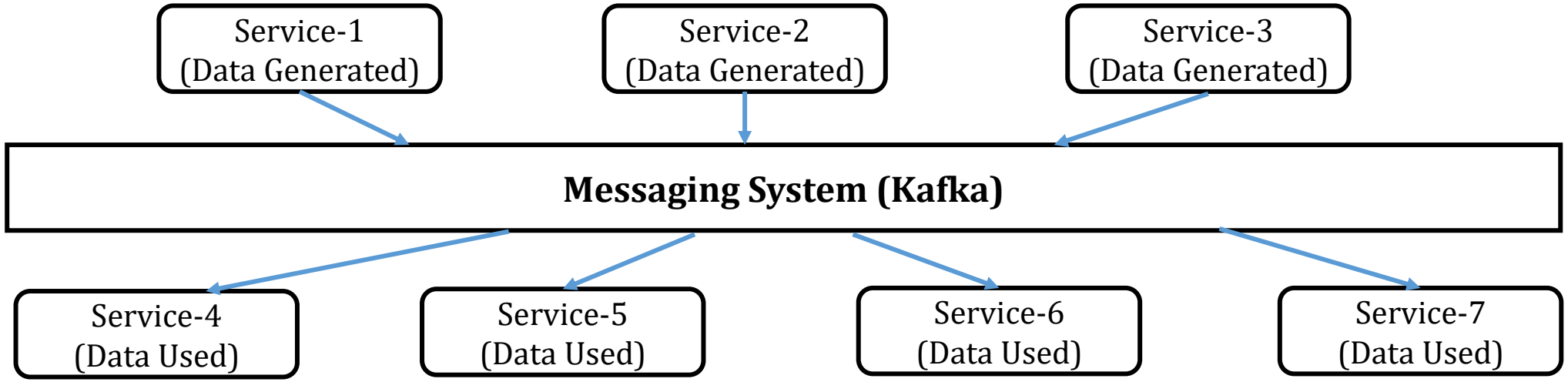
Problems:

Data format, Number of Connections, Connection types

Why do we need kafka?



Tightly Coupled System, Multiple connections in between, Each connections facing many difficulties (like, handle data types, connection types, Schema, and so on.)



What is a Messaging System?

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it.

- Distributed messaging is based on the concept of reliable message queuing.

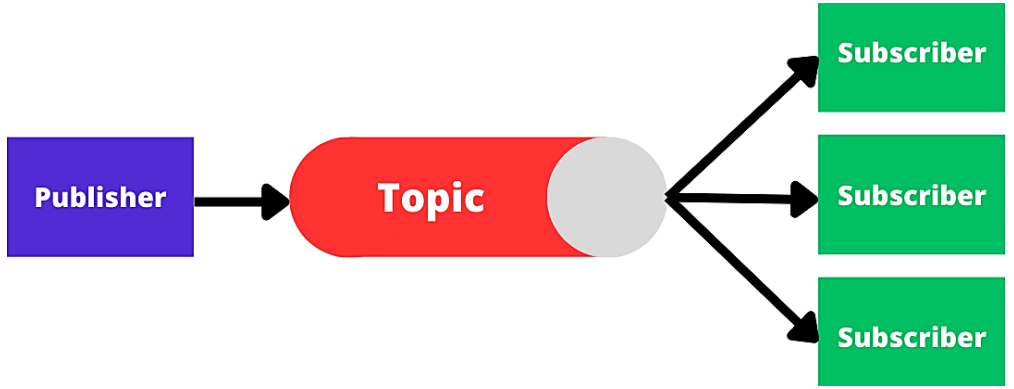
- Messages are queued asynchronously between client applications and messaging system.

- Two types of messaging systems are available:
  - Point to Point messaging system
  - Publish-Subscribe (pub-sub) messaging system.

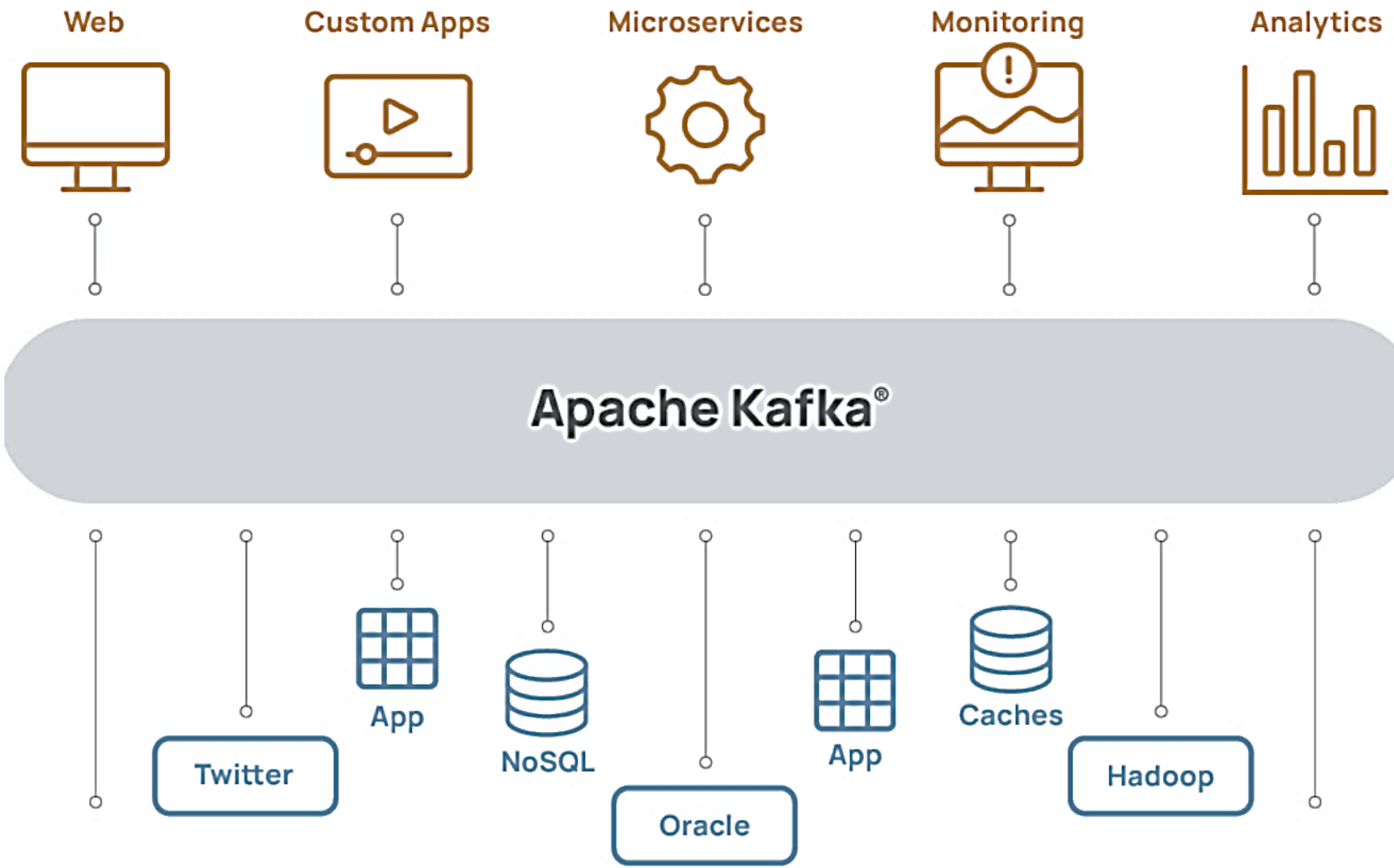
- Most of the messaging systems follow **public-subscribe.**

Types of Messaging system

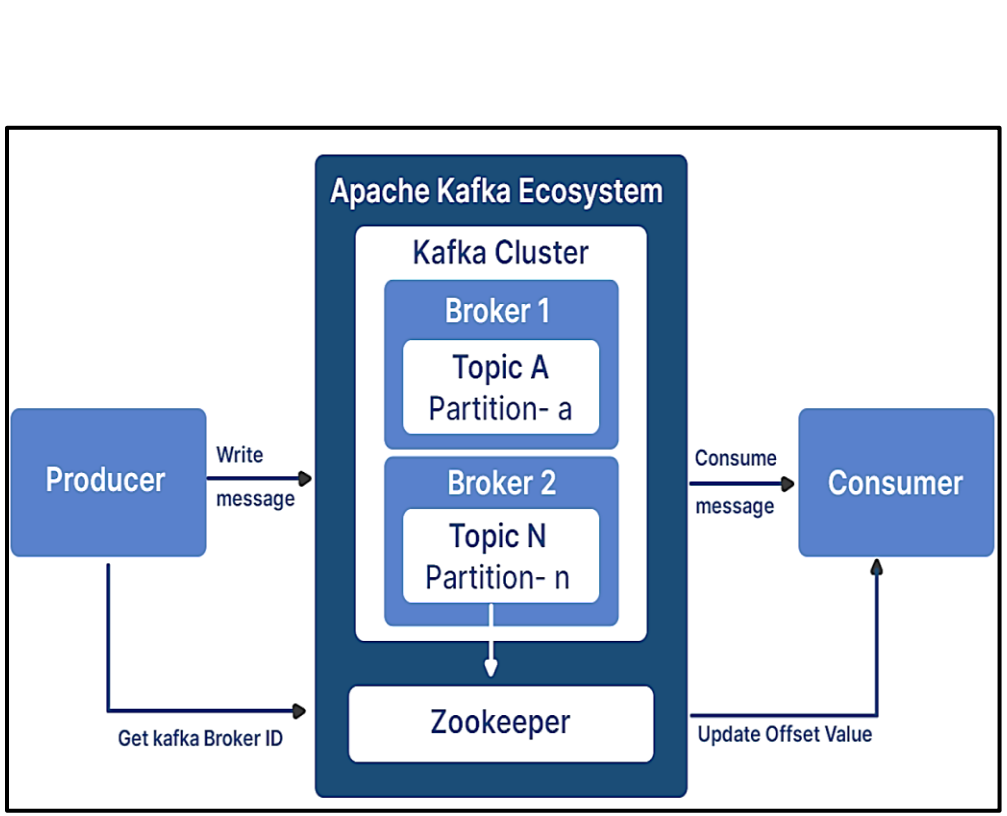
Point to Point Messaging system	Public-Subscribe Messaging system
Massages are persisted in queue.	Massages are persisted in Topics
Particular message can be consumed by one receiver/consumer only. Once the message is consumed, it is removed from the queue. It is useful in scenarios where only one consumer needs to process a message.	Particular message can be consumed by any number of receivers/consumers. It is beneficial in scenarios where there are multiple consumers for a particular message.
There is no time dependency bid for the receiver to receive the message.	There is time dependency bid for the receiver to receive the message.
When the receiver receives the message, It will send an acknowledgement back to the sender.	When the subscriber receives the message, It does not send an acknowledgement back to the publisher.

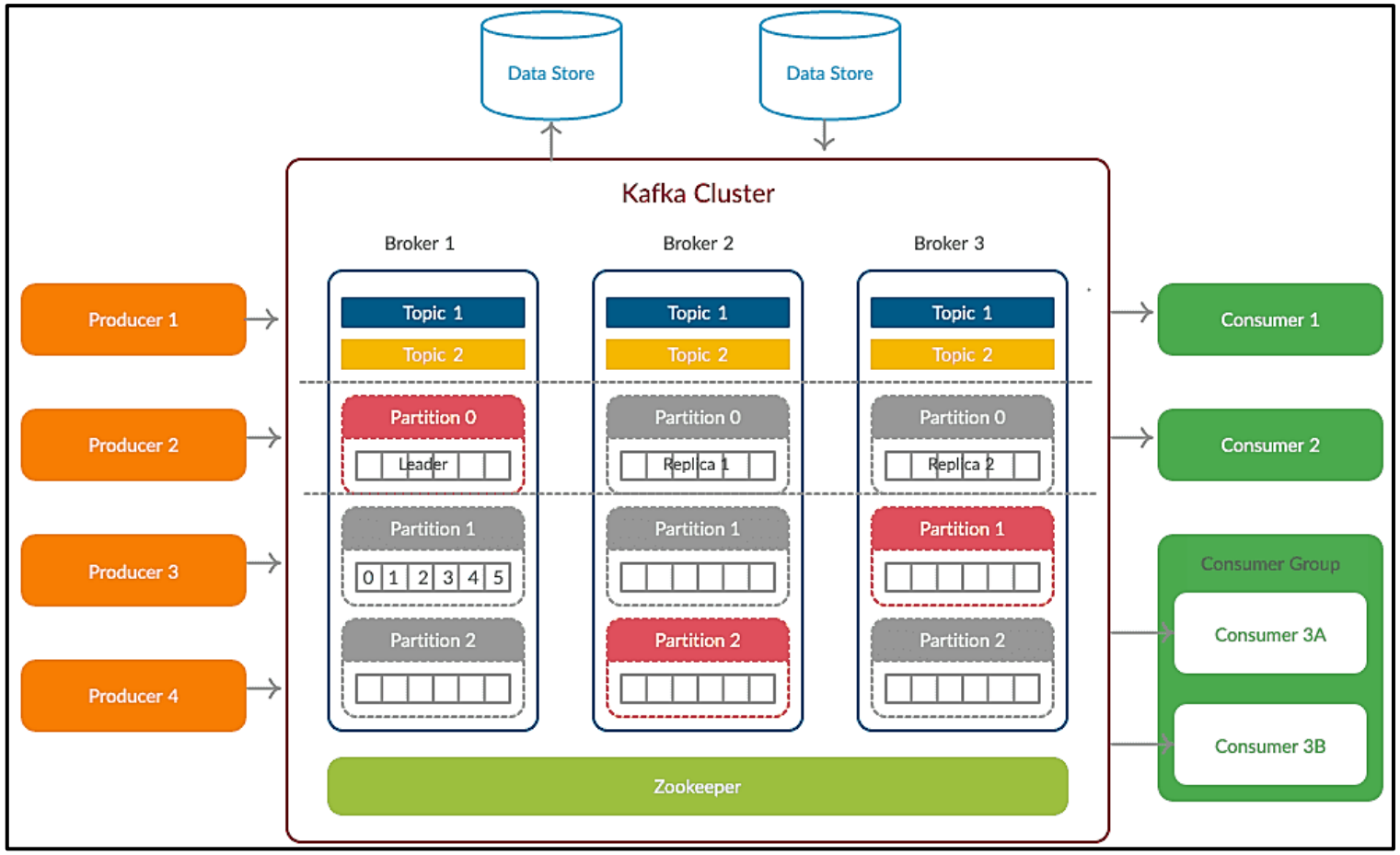


Kafka Ecosystem



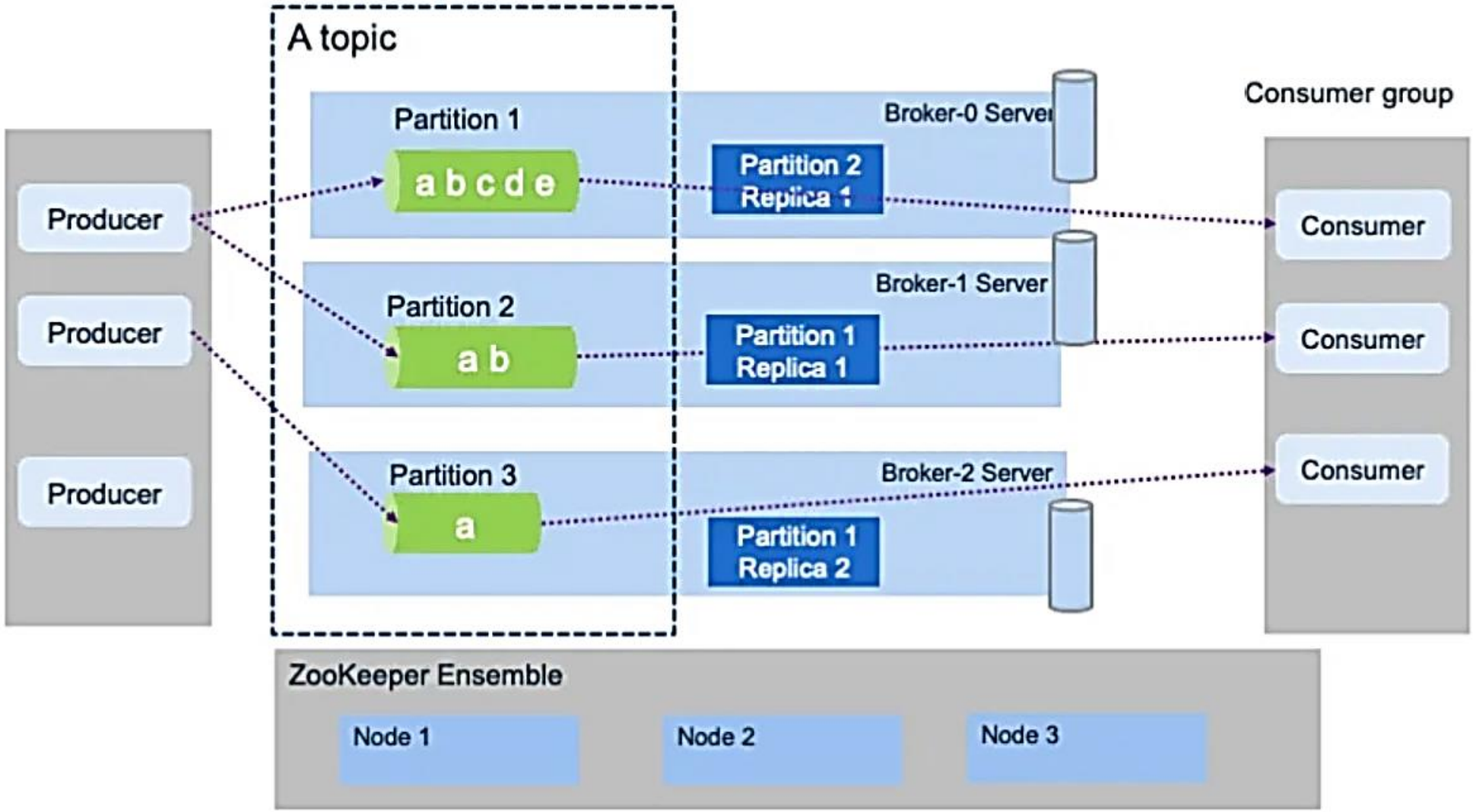
Kafka Ecosystem





Note: Both figures are same

Kafka architecture

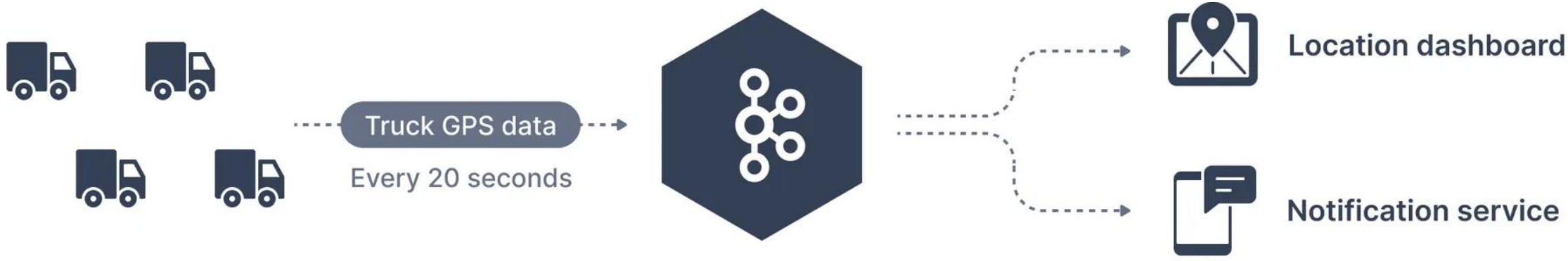


Kafka Topics

- Similar to how databases have tables to organize datasets, Kafka uses the concept of topics to organize related messages.
- A topic is identified by its name. For example, we may have a topic called **logs** that may contain log messages from our application, and another topic called **purchases** that may contain purchase data from our application as it happens.
- Kafka topics can contain any kind of message in any format, and the sequence of all these messages is called a data stream.
- Data in Kafka topics is deleted after one week by default (also called the default message retention period), and this value is configurable. This mechanism of deleting old data ensures a Kafka cluster does not run out of disk space by recycling topics over time.
- Kafka topics are **immutable**: once data is written to a partition, it cannot be changed

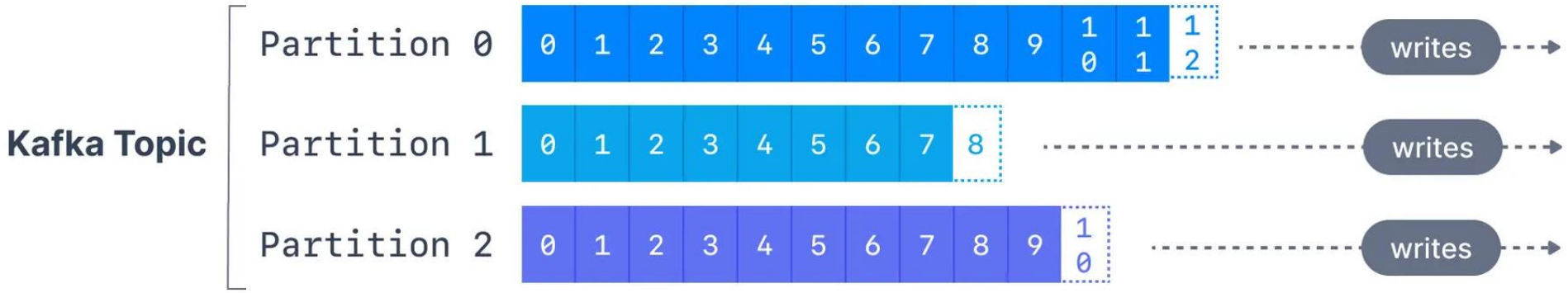
Kafka Topic Example

- A traffic company wants to track its fleet of trucks.
- Each truck is fitted with a GPS locator that reports its position to Kafka.
- Create a topic named - **trucks_gps** to which the trucks publish their positions.
- Each truck may send a message to Kafka every 20 seconds, each message will contain the truck ID and the truck position (latitude and longitude).
- The topic may be split into a suitable number of partitions, say 10.
- There may be different consumers of the topic. For example, an application that displays truck locations on a dashboard or another application that sends notifications if an event of interest occurs.



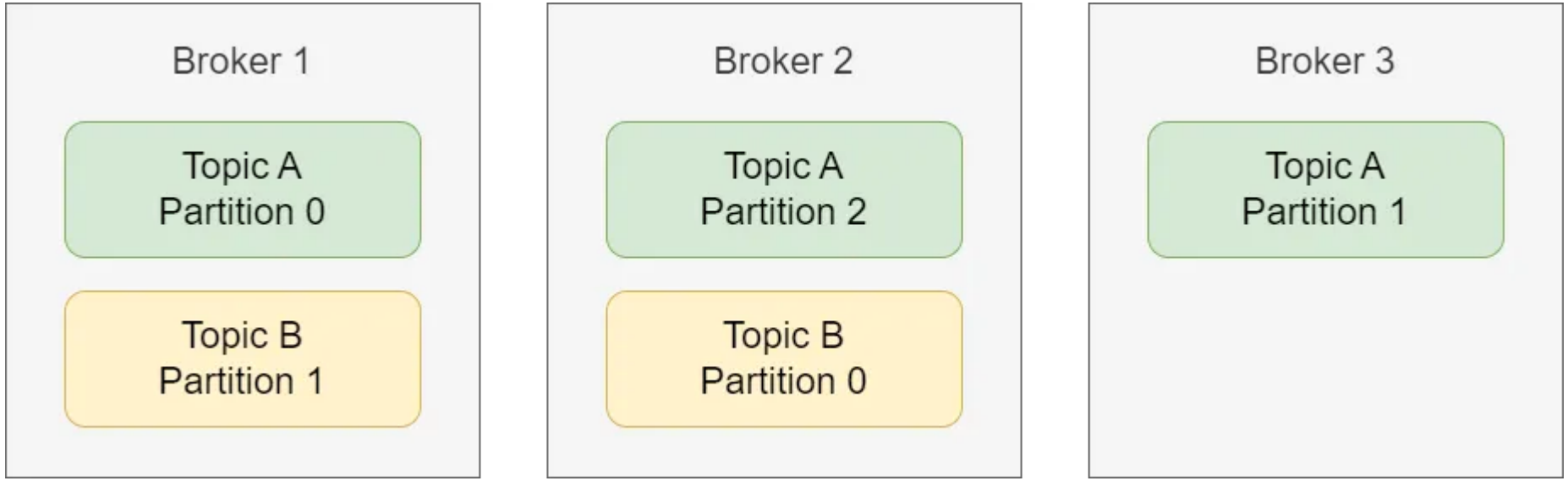
Partitions

- Topics are broken down into a number of partitions. A single topic may have more than one partition, it is common to see topics with 100 partitions.
- The number of partitions of a topic is specified at the time of topic creation. Partitions are numbered starting from 0 to N-1, where N is the number of partitions.
- Apache Kafka offsets represent the position of a message within a Kafka Partition.
- Offset numbering for every partition starts at 0 and is incremented for each message sent to a specific Kafka partition. This means that Kafka offsets only have a meaning for a specific partition, e.g., offset 3 in partition 1 doesn't represent the same data as offset 3 in partition 2.
- Kafka topics deleted over time but offsets are not reused. They incremented continuously.



How brokers, Kafka topics, and partitions interconnect?

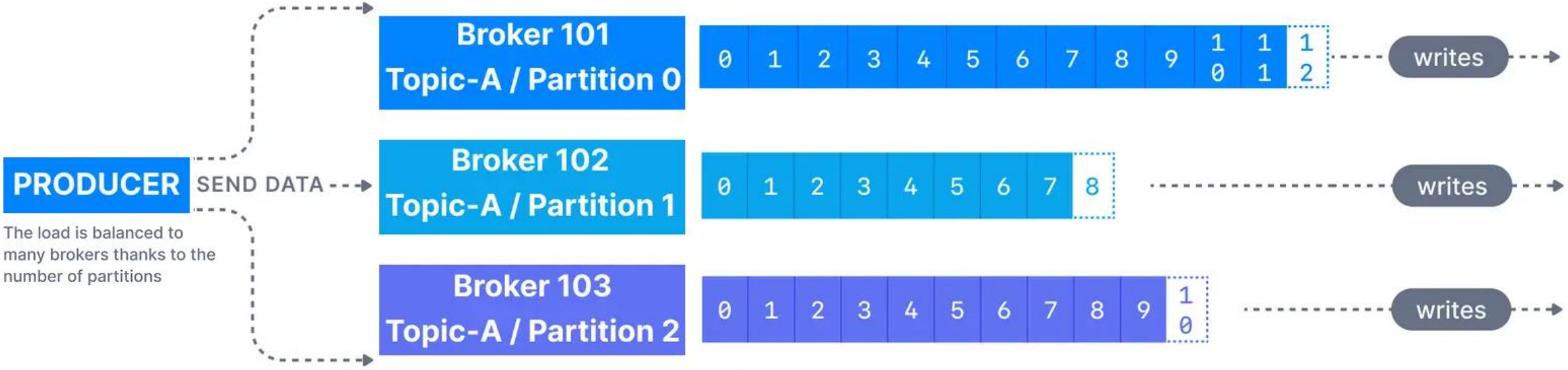
- Topic-A (3-partitions: partition-0, partition-1, partition-2)
- Topic-B (2-partitions: partition-0, partition-1)
- 3-Kafka brokers: Brokers 1, 2, and 3.
- Broker 1 manages: (Topic-A, Partition 0), (Topic-B, Partition 0)
- Broker 2 manages: (Topic-A, Partition 2), (Topic-B, Partition 1)
- Broker 3 manages: (Topic-A, Partition 1)



Kafka architecture

Producer

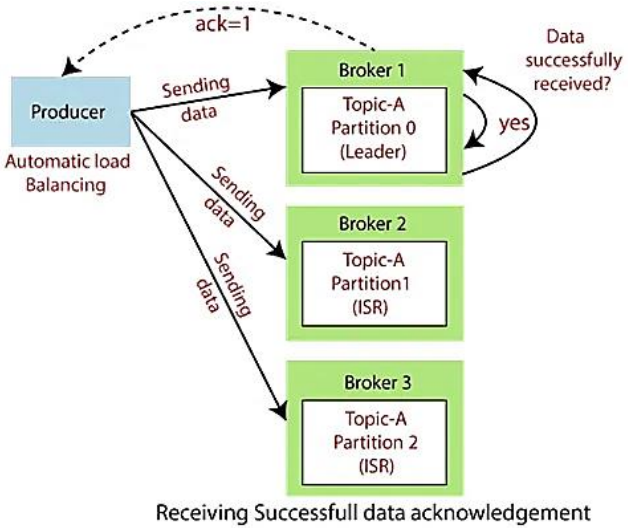
- Once a topic has been created with Kafka, the next step is to send data into the topic. This is where Kafka Producers come in. Applications that send data into topics are known as Kafka producers.
- Messages are stored in the form of key-values in the partitions of the topics.
- Messages produced with the same key are written to the same partition.



Kafka architecture

Producer

- Some configurations that may be needed for producer configuration are as follows:
  - **ACK:** Minimum number of acknowledgments that must come from the broker for the producer to accept a message as sent to the Kafka cluster.
  - **max.in.flight.requests.per.connection:** controls how many requests can be made in parallel to any partition, and so, by default setting is 5.
  - **Linger.ms:** is the number of milliseconds a producer is willing to wait before sending a batch out. And by default, it's zero.
  - **Batch.size:** It is represents the maximum size (in bytes) of a single batch..



Kafka architecture

Message Keys

- Each event message contains an optional key and a value.
- In case the key (key=null) is not specified by the producer, messages are distributed evenly across partitions in a topic. This means messages are sent in a round-robin fashion (partition p0 then p1 then p2, etc... then back to p0 and so on...).
- If a key is sent (key != null), then all messages that share the same key will always be sent and stored in the same Kafka partition. A key can be anything to identify a message - a string, numeric value, binary value, etc.
- Kafka message keys are commonly used when there is a need for message ordering for all messages sharing the same field.

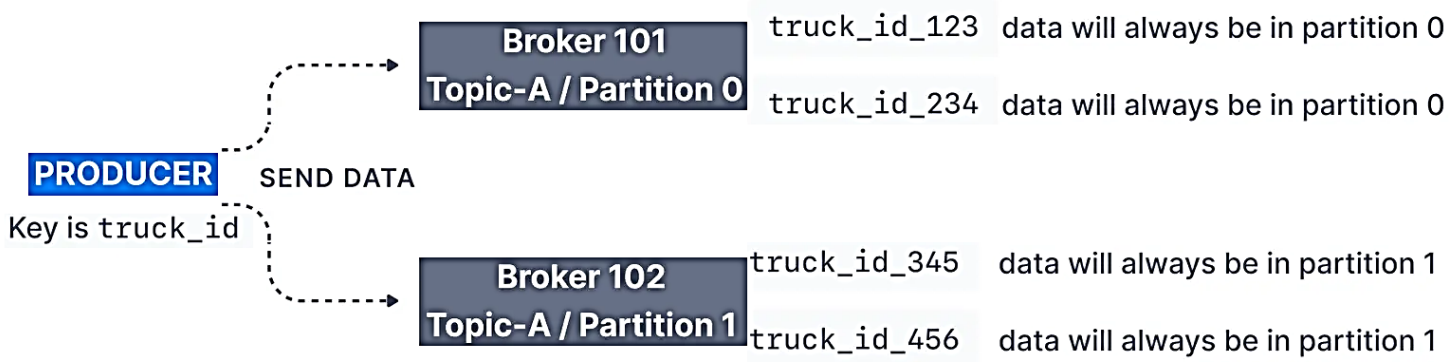
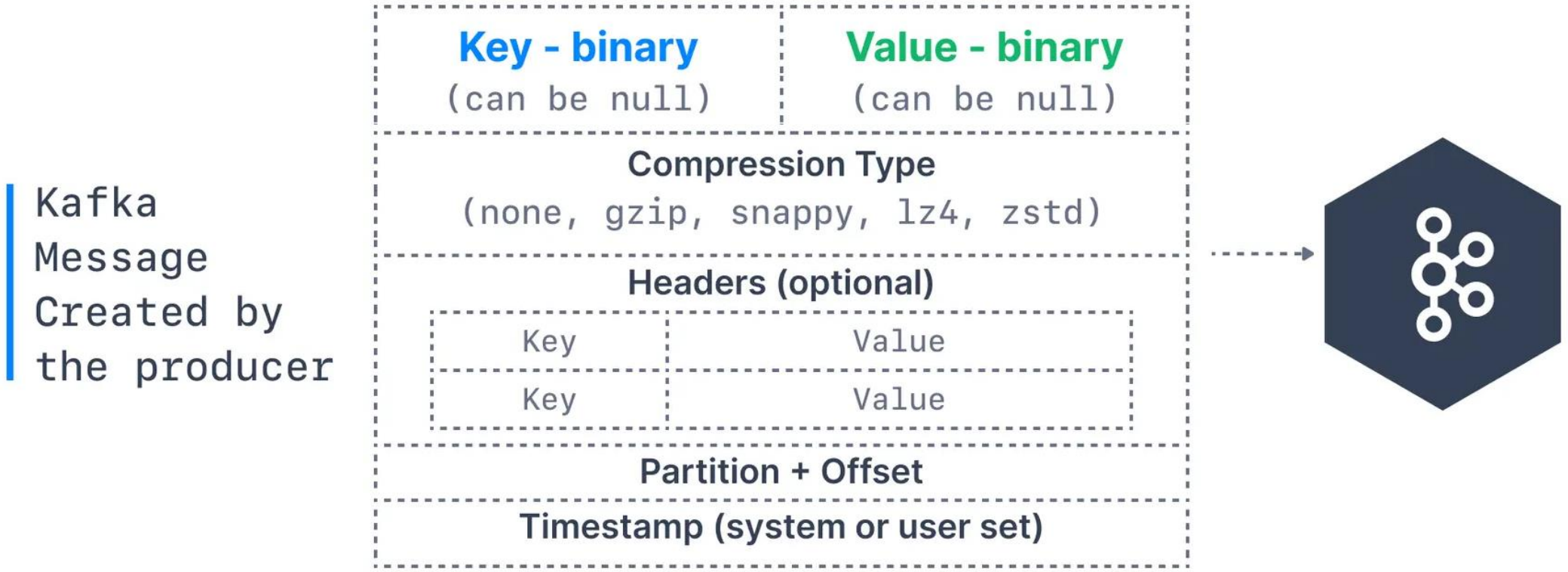


Figure:
In the scenario of tracking trucks in a fleet, we want data from trucks to be in order at the individual truck level.

Structure of kafka message

- Kafka messages are created by the producer. A Kafka message consists of the following elements:



Kafka architecture

Consumer

- Consumers are client applications that read event messages from topics.
- Some configurations that may be needed for consumer configuration are as follows.
  - **auto.offset.reset:** It is used to determine from where the consumer will read the data. If set to “earliest”, the data in the topic is read again from the beginning. If set to “latest” it continues reading from the last offset. If “none”, it means the starting offset will be determined manually.
  - **enable.auto.commit:** After a consumer reads a message, the offset is advanced by 1 so that another consumer or the same consumer can continue from the next message. The message must be committed to increase the offset by 1. If this parameter is set to true, if the consumer does not fail in any case after reading the message, it is automatically committed. If set to false, manual commit is required with the commitSync() method.
  - **fetch.min.bytes:** Minimum number of bytes of data to be fetched at a time. Default value: 1.
  - **fetch.max.bytes:** Maximum number of bytes of data to be fetched at a time. The default value is 52428800(50MB).
  - **max.poll.records:** How many records will be captured at once. default value is 500.

Kafka architecture

Consumer group

- Consumer groups allow Kafka consumers to work together and process events from a topic in parallel.

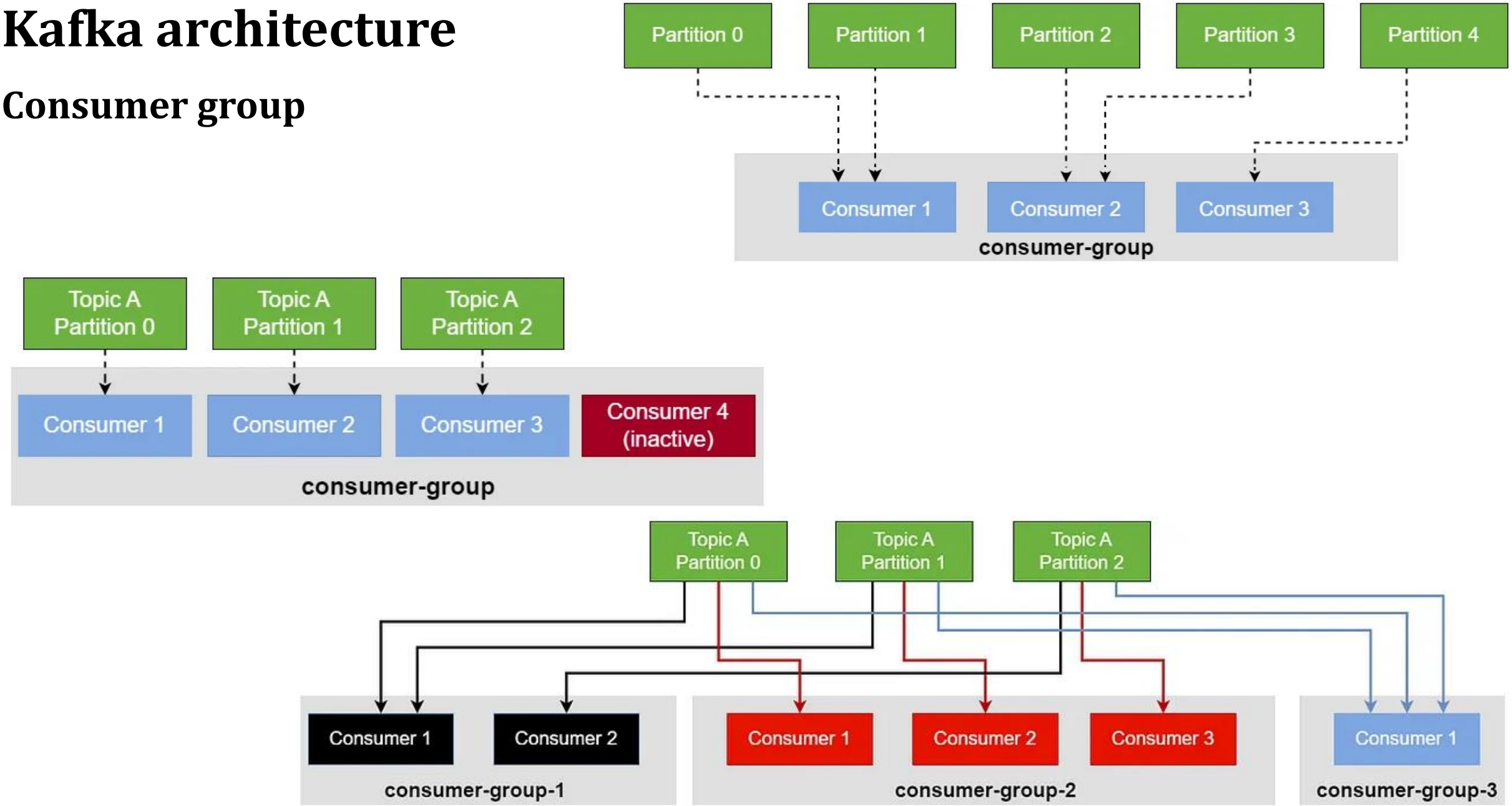
- They serve two primary purposes:

    - Parallelism: Consumer groups allow you to distribute the consumption of data across multiple consumers. This parallelism is essential for handling high volumes of data in real-time.

    - Fault Tolerance: Consumer groups ensure fault tolerance. If a consumer within the group fails, Kafka automatically reassigns its partitions to other consumers, ensuring that no data is lost, and processing continues seamlessly.

Kafka architecture

Consumer group



Kafka Broker

- A broker is a server, but in Kafka terminology, they're known as brokers because they handle data (sending and receiving). It is the most basic component of Kafka systems.
- Kafka cluster consists of several Kafka brokers.
- Each broker is identified with an ID consisting of a number.
- They are physically connected to the server on which they are installed,
- Each broker stores specific topic partitions. This means the data is spread across all brokers in the cluster.
- Broker is aware of all topics and partitions in the cluster, even if it is not on it.
- Brokers serve as connection points for both producers and consumers.
- Each Kafka Broker is also called a "bootstrap server"

Kafka architecture

Zookeeper

- It is an open source service with distributed, key-value data storage capability, used in many open source projects, especially big data projects.
- Zookeeper for Kafka; adding and removing brokers to the cluster, determining the leader/controller broker, keeping topic configurations, etc.
- Responsible for cluster management issues.
- Nowadays, Kafka has ended the use of zookeeper in its new versions. But it is also available in systems that currently use it.
- Without Zookeeper, the following benefits are observed in Kafka:
  - Ability to scale to millions of partitions, easier to maintain and set up
  - Improved stability, easier to monitor, support, and administer
  - Single process to start Kafka
  - Single security model for the whole system
  - Faster controller shutdown and recovery time

Serialization and Deserialization in Kafka Streams

Serialization (Object to Binary)

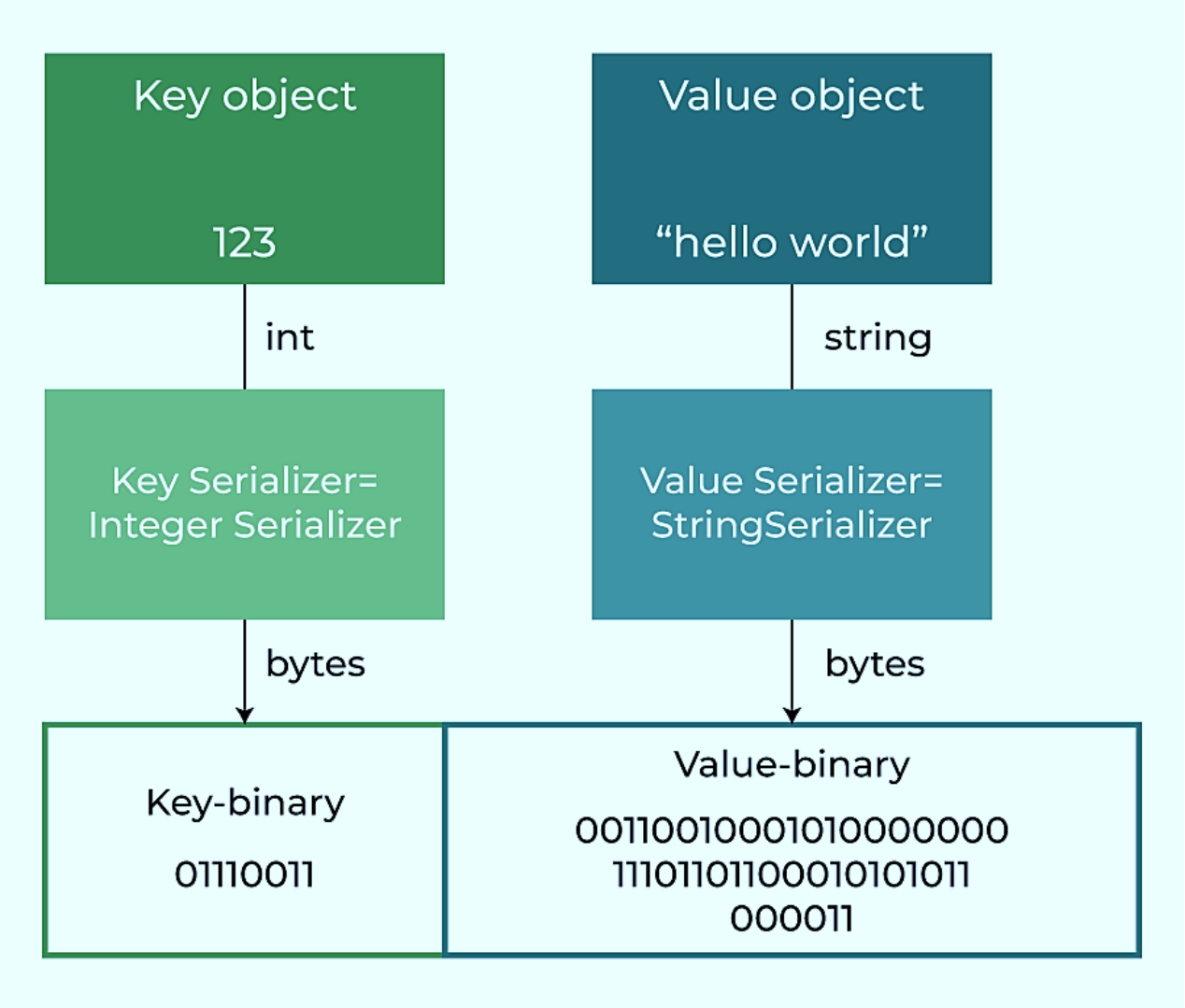
- It is the process of converting data objects into a binary format suitable for transmission or storage.

- Purpose: Data in Kafka is stored and transmitted in binary format for efficiency. Serialization converts objects into bytes for network transmission or storage in Kafka topics.

- Example: When producing messages to a Kafka topic in Kafka Streams, you serialize the key and value objects into byte arrays using a serializer.

- For instance, if you have a Java object representing a message with a key and value, you would use a serializer like StringSerializer, IntegerSerializer, JsonSerializer, AvroSerializer, etc., to convert these objects into bytes before sending them to Kafka.

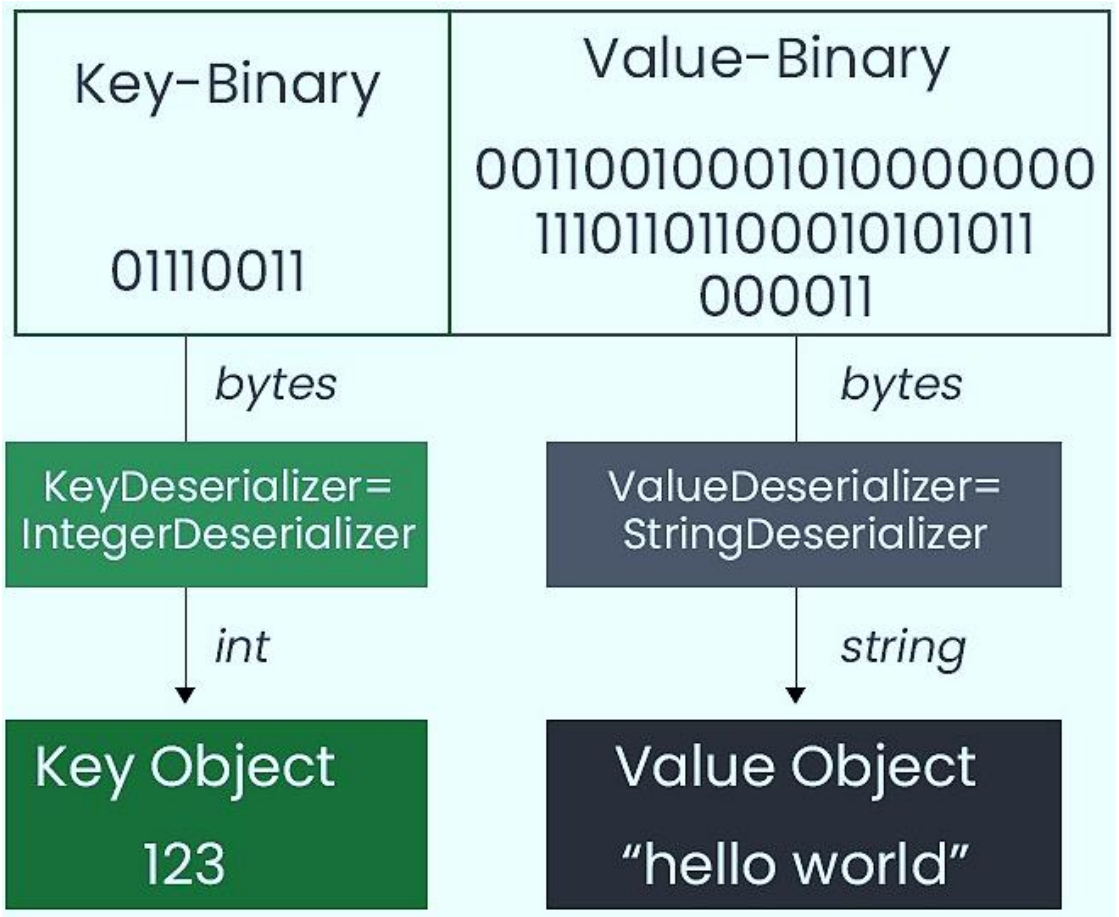
Kafka Serialization (Object to Binary) **Example**



Serialization and Deserialization in Kafka Streams

Deserialization (Binary to Object)

- Deserialization is the reverse process, where byte streams are converted back into structured data. It's like unpacking your suitcase after a journey, restoring items to their original state.



Why Kafka is a good choice?

- Kafka is highly scalable. It can handle millions of events per second and is designed to handle high throughput and low latency data streams. This makes it an ideal choice for large-scale data processing and real-time data streaming.

- Kafka is fault-tolerant. It can handle node failures and maintain data consistency across a cluster of nodes. This ensures that data is not lost in the event of a failure and that the system can continue to operate without interruption.

- Kafka is highly configurable. It offers a wide range of configuration options that can be used to fine-tune the system to meet the specific needs of an organization. This allows organizations to optimize Kafka for their specific use case.

- Kafka can be used in a variety of different use cases. It can be used for data processing, real-time data streaming, data integration, and more.