

Kafka Interview Preparation

What is Apache Kafka?

Apache Kafka is a **distributed, fault-tolerant, and high-throughput messaging system** based on the **publish-subscribe model**. It was originally developed at LinkedIn and open-sourced in 2011.

- **Send and receive real-time data**

- → Like a chat app, where one service sends a message and others instantly get it.

- **Store data safely**

→ If a service is offline, Kafka saves the data until the service comes back and reads it.

- **Process data as it flows**

→ Kafka can also analyze data as it moves, using something called the **Kafka Streams API**.

Example:

Imagine you are getting live order data from an online store:

- **Without Kafka Streams:** You just collect and store all the orders.
- **With Kafka Streams:** You can **count orders, find totals, or detect fraud** — **while** the data is still coming in.

So, Kafka Streams helps you **do smart things with data in real time**, without needing to store and analyze it later.

Kafka Advantages

- **Loose Coupling:** Services communicate without needing to know about each other.
- **Durability:** Messages are stored on disk and can be replayed even if the consumer is down.
- **Scalability:** Easily handles large volumes of data and can scale horizontally.
- **Flexibility:** New consumers can be added without modifying the producers.

Explain Horizontal scaling and vertical scaling?

Horizontal Scaling (Scale Out)

- **Add more machines** to handle more load.
- Each machine (or server) handles part of the work.
- Common in **distributed systems** like Kafka.

Example:

You have 1 Kafka broker. If traffic increases, you add 2 more brokers — now 3 Kafka servers share the load.

✅ **Pros:** More flexible, fault-tolerant, and easy to scale.

❌ **Cons:** Slightly complex to manage.

Vertical Scaling (Scale Up)

- **Add more power** (CPU, RAM, storage) to your **existing machine**.
- You're making a single server stronger.

Example:

You upgrade your Kafka server from 8 GB RAM to 32 GB RAM to handle more messages.

✅ **Pros:** Simpler setup.

❌ **Cons:** There's a limit to how much you can upgrade one machine, and if it fails — everything fails.

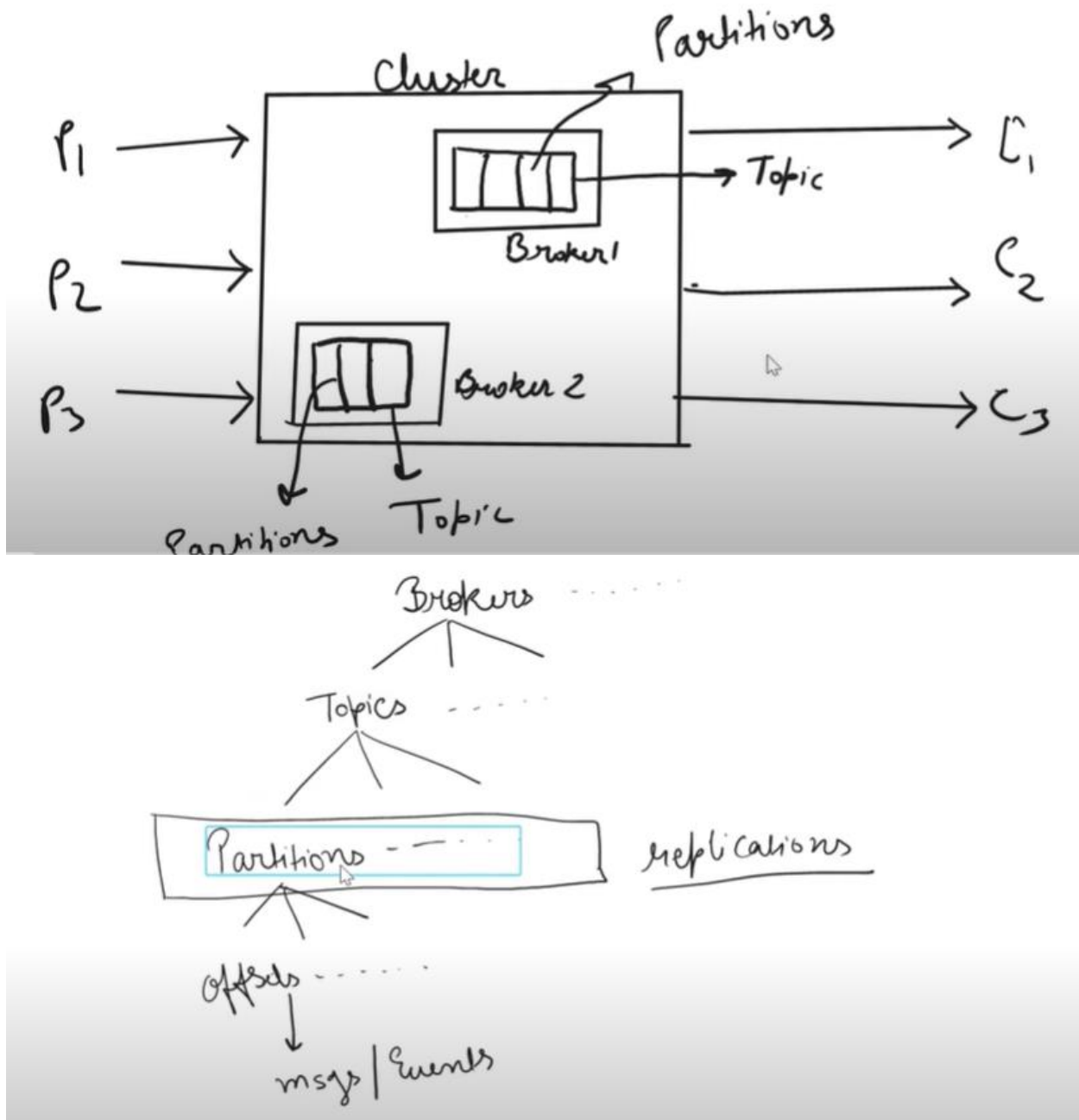
Kafka uses Horizontal Scaling - Because it can run on multiple servers (called **brokers**) and share the data among them.

Step by step Kafka Flow:

Step-by-step Flow:

1. **Producer** creates and sends events.

2. Kafka **producer batches the events** (internally).
3. These **batches are sent directly to Kafka brokers**.
4. The **broker receives and stores them** in the topic partition on disk.
5. **Consumers** later pull from these topic partitions.



Example Flow (RabbitMQ):

Producer sends message → Broker queues it.

1. Broker delivers the message to a consumer.
2. Consumer processes it and sends back an **acknowledgment** (ACK).
3. Broker **removes** the message from the queue.

If **no ACK is received**, broker will:

- Assume failure (e.g., consumer crashed).
- **Redeliver** the message to another consumer.

👉 This process is called **message tracking** or **delivery state management**.

Differences between ActiveMQ/RabbitMQ and kafka?

| Feature | Active MQ/Rabbit MQ | kafka |
|------------------|--|--|
| Type | Message Broker (traditional) | Distributed Event Streaming Platform |
| Message Delivery | Push-based — Broker sends (pushes) messages to consumers. | Pull-based — Consumers fetch (pull) messages when they're ready. |
| Throughput | Lower Throughput — Each message needs acknowledgment, message state is tracked. | High Throughput — Producers don't wait for acknowledgments. Messages are written fast in batches. |
| Message Storage | Short-term — Once consumed, the message is deleted. | Long-term — Messages are stored for a defined time, even after being read. |
| Message Replay | Not possible — Once a message is consumed, it's gone. | Possible — Consumers can re-read old messages using offsets. |
| Ordering | Maintains order within a queue . | Maintains order within a partition . |
| Scaling | Limited Horizontal Scaling — Difficult to scale across nodes. | Easy Horizontal Scaling — Add more brokers and partitions. |

| | | |
|---------------------|--|--|
| Delivery Guarantees | Supports at-most-once, at-least-once, exactly-once (with plugins). | Supports at-least-once and exactly-once (with config). |
| Latency | Low latency for small workloads | Slightly higher latency, but optimized for throughput . |
| Stream Processing | Not supported natively. | Built-in Kafka Streams API for real-time processing. |
| Use Cases | Ideal for traditional messaging: commands, notifications, tasks | .Ideal for data pipelines, analytics, log aggregation, and real-time apps. |

message state is tracked means →

The broker keeps track of the **message state**:

- Has it been delivered?
- Has it been acknowledged (successfully processed)?
- Should it be redelivered if no ACK is received?

Consumers can re-read old messages using offsets. →

In **Kafka**, every message in a **partition** is stored with a unique **offset** — a number that tells the position of the message.

Kafka UI:

Kafka UI is a **web-based tool** that helps you **interact with Kafka visually**. (like we will open <https://localhost:8080> in browser right.. that only kafka UI)

Instead of using command-line tools, Kafka UI lets you **see and manage Kafka data easily** from your browser.

Kafka UI Lets You:

1. See all topic names in the Kafka cluster
2. Check how many partitions each topic has
3. View replication factor (how many copies per partition)

4. See how many messages are in each topic
5. Send (produce) messages to any topic manually
6. Read (consume) messages from a topic
7. Test message flow without writing code → Kafka UI allows you to send and read messages directly from your browser —

👉 without needing to write Java, Python, or CLI scripts.

8. View all consumer groups
9. Track offsets (how far each group has read) → If a partition has 100 messages (offsets 0 to 99), and a consumer group has read up to offset 79, then:

The group's offset = 79

It has read 80 messages

10. Check consumer lag (how behind the group is) → Lag means the difference between the latest message in Kafka and the consumer's current offset.

✓ **Example:**

- a. Latest message in topic = offset 99
- b. Group has read up to = offset 79
- c. 👉 Lag = $99 - 79 = 20$

So the consumer is 20 messages behind. That's the consumer lag.

11. Monitor group activity/status
12. See all Kafka brokers and their status (online/offline)
13. Create new topics
14. Delete topics
15. Edit topic settings (like partitions, retention time) → Retention Time
 - a. This defines how long Kafka should keep messages before deleting them.

Why Use Kafka UI?

- Easy to use — no command-line needed
- Helps developers test Kafka features
- Useful for debugging and monitoring
- Saves time during development and support

What is a Kafka Topic?

A **Kafka topic** is like a **container** where messages (also called events) are stored.

- **Producers** send (write) messages into the topic.
- **Consumers** read messages from the topic.
- You can have **many producers and many consumers** for a single topic — this is called **multi-producer, multi-consumer**.


Important point:

In most traditional systems, once a message is read, it's deleted.

But in Kafka, **messages are not deleted after being read**.

Instead, Kafka keeps them for a specific amount of time (like 7 days or more) — this is called **retention**.

Think of it like this:

 *Topic = Folder*

 *Message = File inside the folder*

What are Partitions?

Kafka divides a topic into smaller parts called **partitions**.

- Each **partition** stores messages in the order they arrive.
- Once a message is added to a partition, it **can't be changed**.
- Every message inside a partition has a unique number called an **offset** (used to identify the message).

Why use partitions?

- So that **multiple consumers** can read from the same topic **at the same time**.
- This improves **speed and performance**.

 Example:

If a topic has 3 partitions, then 3 consumers can read from it **in parallel**.

What is Replication?

Kafka makes **copies** of each partition to make the system more reliable. These copies are called **replicas**.

- Replication is done **at the partition level**.
- One of the replicas is the **leader** → it handles all reads and writes.
- The other replicas are **followers** → they copy data from the leader.

If the leader fails, one of the followers will automatically take over as the new leader.

This keeps the system **running without data loss**.

Leader and Follower Behavior ?

- **Each partition** has **one broker acting as the leader**.
- The other brokers for that partition are **followers**.
- The **leader handles all client requests** (reading or writing messages).
- **Followers replicate** (copy) the leader's data to stay in sync.

👉 **If the leader fails**, Kafka automatically **promotes a follower** to become the new **leader**, ensuring **no interruption** in service.

How a Producer Sends a Message

1. The producer sends a message to a **topic**.
2. Kafka routes it to the **leader of a partition** within that topic.
3. The leader:
 - a. **Appends** the message to the partition.
 - b. Assigns it a **unique offset**.
 - c. Increments the offset for the next message.

In Simple Terms

- **Topic** = a folder to hold messages.

- **Partitions** = split the topic into parts for better performance.
- **Offsets** = unique ID for each message in a partition.
- **Replication** = backup copies of partitions to prevent data loss
- **Leader** = handles all writes/reads.
- **Followers** = sync data from leader.
- **Producers send to the leader**, and the leader stores the message with an offset.

Will all replicas be on different brokers?

✅ **Yes — ideally, Kafka tries to place each replica on a different broker.**

This is done to ensure **high availability**.

- If one broker goes down, another broker still has the data.
- This is why a replication factor **must be less than or equal to the number of brokers**.

How Does the Producer Choose the Partition?

When a producer sends a message to a Kafka topic, it decides **which partition** the message should go to using one of the following methods:

1. Manual Partitioning

You **explicitly specify** the partition number:

```
javaCopyEditnew ProducerRecord<>("topic", 2, "key", "value");
```

- The message is sent **directly to partition 2**.
- Kafka does **not interfere**.

2. Key-based Partitioning

You **provide a key**, but not a partition:

```
javaCopyEditnew ProducerRecord<>("topic", "user123", "value");
```

- The producer uses:

```
iniCopyEditpartition = hash(key) % total_partitions
```

- All messages with the same key (e.g., "user123") go to the **same partition**.
- Ensures **order is preserved** for that key.

3. Round-Robin Partitioning (No Key)

You don't provide a key or partition:

```
javaCopyEditnew ProducerRecord<>("topic", "value");
```

- The producer distributes messages in a **round-robin fashion**.
- Ensures **load is balanced** across all partitions.

iniCopyEditpartition = hash(key) % total_partitions → here total partitions value how we will get ?

When the producer sends a message:

- It **automatically contacts the Kafka cluster**
- Gets the **metadata** of the topic
 - This includes how many **partitions** the topic has

👉 So, **you don't need to manually set total_partitions** in your code.

The **Kafka producer client** fetches it from the broker.

Consumers?

- A **Kafka consumer** is an application that reads messages from a Kafka topic. Just like how a producer sends data, a consumer receives it.
- When a topic has multiple **partitions**, a consumer can read from one or more of those partitions. But in most cases, consumers work together in a **consumer group**. → Each consumer in the group takes care of **some partitions**, so they **share the work** and process messages **faster and more efficiently**.
- A **consumer group** is a set of consumers that share the work of reading messages from a topic. Within a group, **each partition is read by only one consumer** at a time. This ensures that no message is read twice within the same group. However, if there are more partitions than consumers, some consumers may handle multiple

partitions. If there are more consumers than partitions, some consumers will be idle.

- Every message in a partition has a unique number called an **offset**. The consumer keeps track of the offset to know which messages it has already read. This way, it avoids processing the same message again. Kafka can auto-commit this offset or let the developer commit it manually.
- Kafka does **not send messages to consumers automatically**.

Instead, **consumers ask (poll)** Kafka regularly to check if there are new messages.

This is called **pull-based communication**.

- You can have many **consumer groups** reading from the **same topic**, and **each group will get all the messages**.

So if you have two different services (like billing and notification), both can read the same topic **independently**, using their own group IDs.

- Each message in Kafka has a unique **offset** (like a line number).

Consumers keep track of the **last offset they read**, so they don't read the same message again.

Kafka can **auto-save** this or let you save it manually.


Broker

A **broker** is a **Kafka server**.

Kafka runs as a **cluster of brokers** — usually 3 or more.

Each broker:

- Stores **topic data (partitions)**
- Handles **read/write requests** from producers and consumers
- Coordinates with other brokers in the cluster

 Example:

If you have 3 brokers: Broker 1, Broker 2, Broker 3 — Kafka will spread partitions and replicas across these brokers.


Partition

A **partition** is a part (or slice) of a **Kafka topic**.

Messages inside a partition are:

- Stored in **order**
- Identified by a unique number called an **offset**

A topic can have **one or many partitions**.

 Example:

If a topic has 3 partitions, Kafka will split messages across **Partition 0**, **Partition 1**, and **Partition 2**.

What is ZooKeeper?

ZooKeeper is a **coordination service** used by Kafka (in older versions) to **manage and monitor the Kafka cluster**.

Think of ZooKeeper like a **manager** who keeps track of:

- Which **Kafka brokers are alive**
- Who is the **leader** of each **partition**
- Helps in **electing a new leader** if one broker goes down

What Does ZooKeeper Do in Kafka?

1. **Keeps track of all brokers** in the cluster

(e.g., Broker 1, Broker 2, etc.)

2. **Manages leader election**

For each partition, there is one **leader** broker.

If the leader fails, ZooKeeper helps pick a **new leader** from the followers.

3. **Stores metadata**

Like topic names, partition counts, replica info, etc.

4. Helps Kafka maintain cluster health