Palash Bajpai

**Kafka**

Table of Contents

[**1. Intro to Kafka** 2](#_Toc203032736)

[1.1. Use Case (Zomato) 3](#_Toc203032737)

[1.2. Key Properties of Kafka 4](#_Toc203032738)

[1.3. Why Kafka 4](#_Toc203032739)

[1.4. Can Kafka replace Databases 6](#_Toc203032740)

[**2. Kafka Working** 7](#_Toc203032741)

[2.1. Architecture 7](#_Toc203032742)

[2.2. Key Components 8](#_Toc203032743)

[2.3. Key Working Principles 10](#_Toc203032744)

[2.4. Auto Balancing 11](#_Toc203032745)

[**3. Kafka with Spring** 14](#_Toc203032746)

# **1. Intro to Kafka**

**Apache Kafka** is a distributed, high-throughput, fault-tolerant messaging system used for building real-time data pipelines and streaming applications. It enables services to communicate asynchronously by publishing and subscribing to streams of records (events) in a decoupled, scalable manner.

**Key points:**

1. **Distributed Messaging System**: Apache Kafka is a **distributed publish-subscribe system** used for building real-time data pipelines and event-driven architectures.

2. **High Throughput & Low Latency**: Kafka handles **millions of messages per second** with **low latency**, making it ideal for large-scale systems.

3. **Topics and Partitions**: Data is organized into **topics**, which are split into **partitions** for scalability and parallel processing.

4. **Producers and Consumers**: **Producers** send data to topics, **Consumers** read data from topics, often as part of a **consumer group**.

5. **Fault Tolerant & Durable**: Kafka stores messages on disk and replicates them across brokers, ensuring **no data loss** even if servers crash.

6. **Replayable Events**: Messages are retained for a configured time, allowing **replay** and **reprocessing** by consumers.

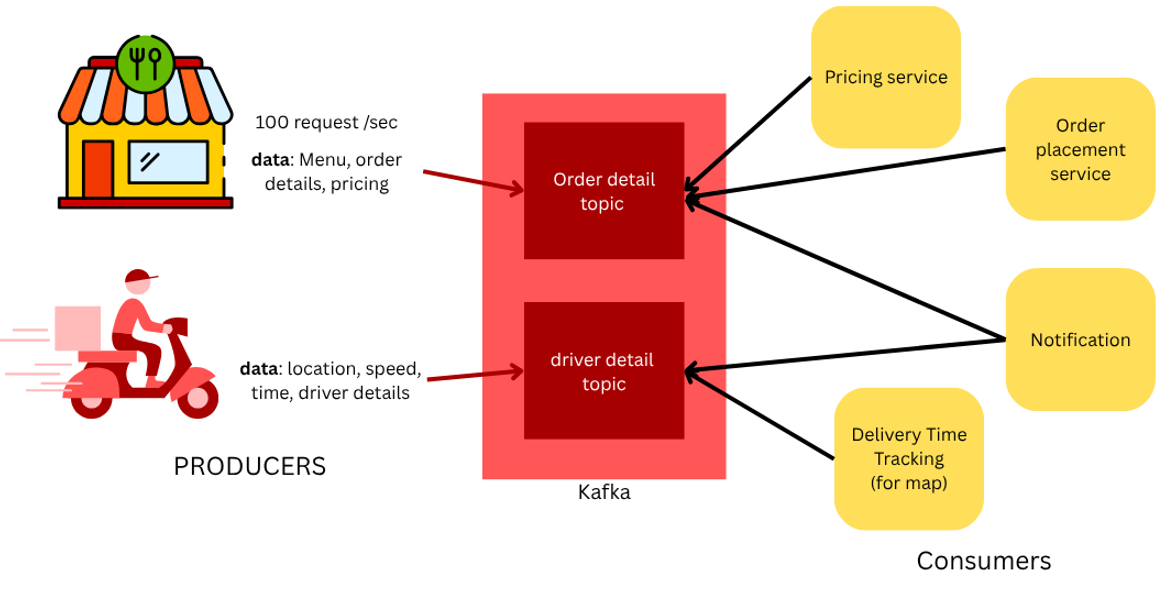
7. **Decouples Services**: Kafka **decouples microservices**, enabling asynchronous communication and reducing system dependency.

8. **Horizontally Scalable**: Kafka scales easily by adding more brokers and partitions to handle growing loads.

9. **Real-Time Data Streaming**: Kafka supports **real-time analytics and processing**, used by companies like LinkedIn, Uber, Netflix.

10. **Open Source and Extensible**: Kafka is an open-source Apache project with support for **Kafka Streams**, **Kafka Connect**, and integration with Spark, Flink, and more

## 1.1. Use Case (Zomato)

****

* **Kafka:** Has high throughput(amount of data that can pass through it), but has temporary storage.
* **DB:** Has low throughput but has permanent storage.

Data first pass through kafka as volume of data is large, and if such large data is directly handled in database, it can crash due to low throughput. So here data is passed through kafka first, later it is bulk inserted in database.

**Flow**

1. **Order Placement**
   * User places an order → OrderPlaced event is **published to Kafka**.
   * Topic: order-events
2. **Inventory & Kitchen Update**
   * Inventory Service **consumes** OrderPlaced to update stock.
   * Kitchen Dashboard Service **consumes** to start preparation.
3. **Delivery Assignment**
   * Kafka topic delivery-events used for **assigning delivery partner**.
   * When food is ready → OrderReady event is published → triggers driver assignment.
4. **Real-Time Tracking**
   * Delivery partner’s location is sent to Kafka topic location-updates every few seconds.
   * Map service **consumes** this to show real-time tracking to the user.
5. **Notification System**
   * OrderConfirmed, OrderOutForDelivery, OrderDelivered events are consumed by Notification Service.
   * Sends push/email/SMS updates via Kafka.
6. **Billing and Analytics**
   * All events (order, payment, delivery) are sent to Kafka topic analytics-events.
   * Analytics Service **aggregates data** for business insights.

## 1.2. Key Properties of Kafka

* **High Throughput**  
  Kafka can handle **millions of messages per second**, making it suitable for large-scale data pipelines.
* **Durability & Persistence**  
  Messages are stored on disk and **replicated** across brokers, ensuring data is not lost.
* **Scalability**  
  Kafka is horizontally scalable via **topics and partitions** across multiple brokers.
* **Fault Tolerance**  
  Kafka replicates data across multiple brokers, so even if a node fails, data remains available.
* **Decoupled Communication**  
  Producers and consumers are **independent**, enabling asynchronous and loosely coupled systems.
* **Replay ability**  
  Consumers can **re-read past messages** by resetting their offsets, useful for debugging or reprocessing.
* **Real-Time Processing**  
  Kafka supports **low-latency** stream processing with integration to tools like Kafka Streams, Flink, and Spark.
* **Exactly Once Semantics (EOS)**  
  Kafka supports **exactly-once delivery** for critical use cases (with proper configuration).
* **Message Retention**  
  Kafka retains messages for a **configurable duration** (e.g., hours, days, or indefinitely).
* **Multiple Subscribers**  
  A Kafka topic can have **multiple consumers/groups**, each processing messages independently.

## 1.3. Why Kafka

| **Problem** | **Kafka’s Solution** | **Example** |
| --- | --- | --- |
| Tight coupling | Event-based, decoupled | Order → Inventory + Email |
| Delays in processing | Real-time streaming | Fraud detection |
| Scalability limits | Horizontal scalability | Uber location data |
| Data loss on failure | Persistent, replayable events | Reprocess failed orders |
| System integration | Central event bus | CRM + Billing + Analytics |
| Event-driven needs | Pub/Sub with multiple consumers | E-commerce workflows |

**1. Decoupling Services (Loose Coupling)**

**Problem:** In a monolithic system, service A calls service B directly. If B is down, A fails. In microservices, this is risky.

**Kafka Solution:** Kafka acts as a **buffer**. Service A just publishes to a topic, and B consumes from it later.

**Example:**

* **Order Service** publishes "Order Placed" event.
* **Inventory Service** and **Email Service** both consume this event independently.
  + No tight coupling between services.
  + If Email service is down, the event remains in Kafka and is picked up when it comes back up.

**2. Real-Time Processing:**  
**Problem:** Traditional databases or batch jobs are slow and not real-time.

**Kafka Solution:** Kafka streams data in real-time, allowing you to process it instantly.

**Example:**

* Uber uses Kafka to process location data from millions of drivers in real time.
* Fraud detection systems consume transaction events instantly and trigger alerts.

**3. Scalability**  
**Problem:** High-traffic systems can’t scale linearly using traditional message queues or REST APIs.

**Kafka Solution:** Kafka is **horizontally scalable**, handling **millions of messages per second** using partitions and brokers.

**Example:**

* LinkedIn processes **trillions** of events per day using Kafka for activity tracking.

**4. Use in Event-Driven Architecture**  
Kafka enables **event-driven** systems where business logic is based on events instead of function calls.

**Example:**

* In an ecommerce site:
* User adds item → "ItemAddedToCart" event
* Inventory updates → "StockUpdated" event
* Checkout → "PaymentProcessed" → "OrderConfirmed"

**5. Data Integration Across Systems**  
**Problem:** Different systems (e.g., CRM, billing, analytics) need to share data.

**Kafka Solution:**Kafka acts as a **central data pipeline**, broadcasting to multiple systems.

**Example:**

* A "User Registered" event is consumed by:
  + **CRM system** to add user info
  + **Billing system** to create account
  + **Analytics system** to track signup

**6. Replayable Event Store**  
**Problem:** If a service goes down or misses data, how do we reprocess events?

**Kafka Solution:** Kafka **persists messages for a configurable time (or forever)**. Consumers can rewind and replay.

**Example:**

* Data pipeline in a bank can reprocess 3 days of failed transactions just by resetting the Kafka offset.

**7. Reliability and Fault Tolerance**  
**Problem:** What happens if a consumer fails or data is lost?

**Kafka Solution:** Kafka replicates data across multiple brokers. Consumers commit offsets, ensuring **no data loss**.

**Example:**

* Payment processor service restarts after a crash and resumes from last committed offset.

## 1.4. Can Kafka replace Databases

**No, Kafka is not a replacement for a database**, but in **some specific scenarios**, Kafka can **temporarily act like one** — with trade-offs. As Kafka stores data temporary only(default 7 days but configurable)

**❌ Kafka Is Not Designed For:**

1. **Querying data** (like SELECT queries in SQL)
2. **Data relationships** (joins, foreign keys, etc.)
3. **Long-term storage** (Kafka retains data for limited time)
4. **Transactional integrity** like ACID in RDBMS

**✅ When Kafka Can Act *Like* a Database**

Kafka can **temporarily serve as a source of truth or event log** in some use cases:

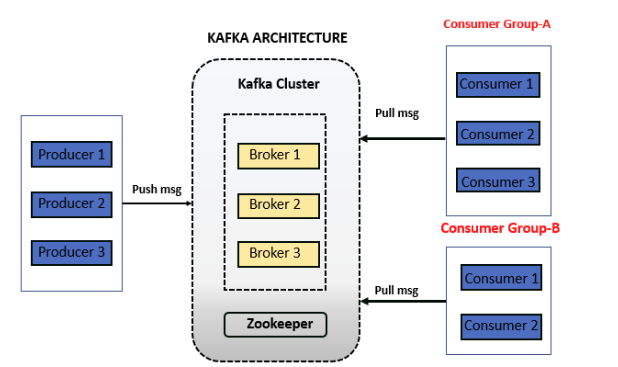
**Examples:**

* **Event Sourcing** systems: State is rebuilt by replaying events from Kafka.
* **Log-based architectures**: Store change data (CDC) events from a database.

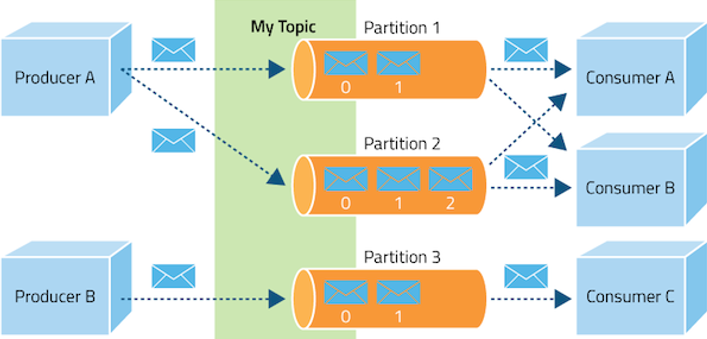
Note: By default, the maximum size of a Kafka message is **1MB** (megabyte). The broker settings allow you to modify the size. Kafka, on the other hand, is designed to handle 1KB messages as well

# **2. Kafka Working**

## 2.1. Architecture

****

1. **Producers** send messages to **Kafka topics**, which are split into **partitions** for scalability.
2. **Brokers** (Kafka servers) store these partitions and handle incoming/outgoing data.
3. **Consumers** read messages from partitions, tracking their progress using **offsets**.
4. **Consumer groups** allow multiple consumers to share the load for parallel processing.
5. Kafka ensures **high availability** through **replication** and supports **replayable, decoupled event streams**.



While Apache Kafka is often referred to and used as a message queue, it's more accurately described as a distributed streaming platform that incorporates aspects of both message queues and publish-subscribe systems.

**Message queue:** Traditional message queues typically follow a First-In, First-Out (FIFO) pattern, where messages are processed in the order they are received. Kafka can be configured to mimic this behaviour within a single partition, but it doesn't guarantee global FIFO order across multiple partitions

*Kafka provides ordering guarantees within a single partition*, but achieving global ordering across multiple partitions can be more complex.

**Publish-Subscribe:** In publish-subscribe model, messages are broadcast to multiple subscribers (consumers) interested in a specific topic. Kafka supports this through its topic-based architecture, allowing multiple consumer groups to subscribe to the same topic and receive messages.

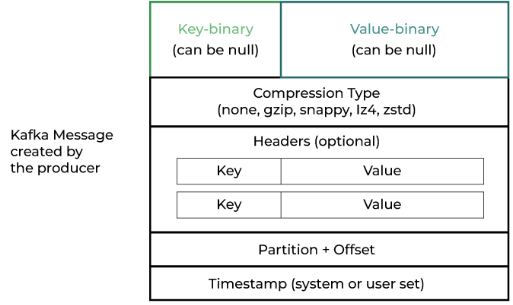
## 2.2. Key Components

**1. Producers**

* Clients that publish (send) data to Kafka topics.
* Can push data continuously (e.g., Order events, logs, metrics).

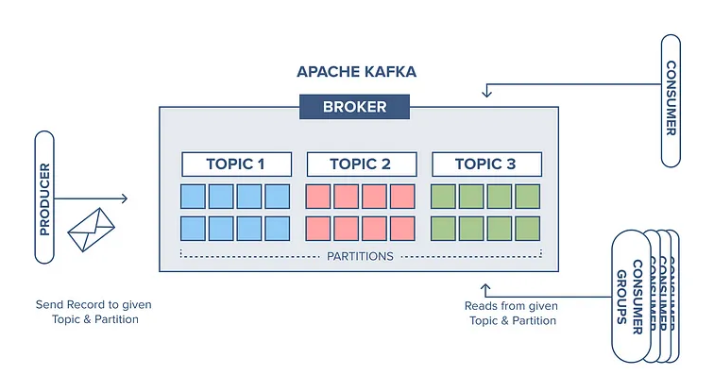
**2. Topics**

* Named channels where messages are published.
* Each topic is split into partitions for scalability.
* Logical grouping of message



**3. Partitions**

* Ordered, immutable logs of messages within a topic.
* Each message has an offset (like a unique ID).
* Enables parallel processing**.**
* Can partition data based on region, category etc.

****

**4. Offsets**

* Unique identifiers for each message **within a partition**.
* Consumers track offsets to know where they left off.

**5. Consumers**

* Clients that **subscribe and read** data from Kafka topics.
* Can belong to a **consumer group** for parallel consumption.

**6. Consumer Groups**

* A group of consumers that **share the work** of reading from a topic.
* Each partition is read by **only one consumer** within a group.

**7. Brokers**

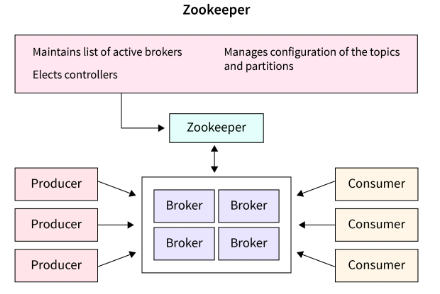
* Kafka servers that **store and serve** messages.
* Each broker handles partitions of multiple topics.
* A Kafka cluster = Multiple broker

**8. Replication**

* Kafka **replicates partitions** across multiple brokers.
* Ensures **high availability** and **fault tolerance**.

**9. Controller (via Zookeeper or KRaft)**

* **Zookeeper manages Kafka cluster metadata**, like broker info, topics, and partitions.
* It **elects the controller broker** and **monitors broker health** for fault tolerance.
* It also helps coordinate **leader election and rebalancing** across the cluster.



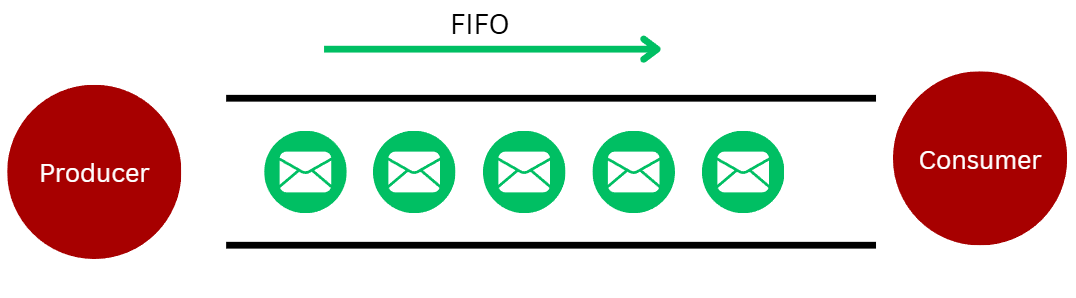
## 2.3. Key Working Principles

1. **Producer Chooses Partition**  
   A producer decides **which partition** of a topic to write to — either by:
   * Using a **key** (deterministic hash → partition), or
   * Kafka **randomly selects** one if no key is provided.
2. **Each Topic Has Multiple Partitions**  
   Partitions allow **parallelism and scalability**. More partitions = more parallel writes and reads.
3. **Messages Are Written with Offsets**  
   Every message in a partition gets a unique **offset**, used by consumers to track reading progress.
4. **Producer Doesn’t Know Consumers**  
   Producers just send data to Kafka topics — they are **completely decoupled** from consumers.
5. **Consumer Groups Control Read Logic**  
   A **consumer group** shares the workload:
   * Each partition is read by **only one consumer** in the group
   * But a consumer can read **from multiple partitions**
6. **One Partition = One Consumer (per group)**  
   Within a group, **a single partition is never read by multiple consumers** at the same time.
7. **One Partition Can Have Many Consumers (in different groups)**  
   Multiple consumer groups can read the **same partition independently**, enabling **pub/sub behavior**.
8. **Brokers Store Data, Not Route It**  
   Kafka brokers **store partitions** and respond to pull requests from consumers; they don’t push data.
9. **Kafka Stores Data Regardless of Consumption**  
   Kafka retains messages for a set time (e.g., 7 days), whether they are consumed or not.
10. **Consumers Track Their Own Offsets**  
    Consumers are responsible for **managing and committing offsets** (automatically or manually), to resume reading where they left off.

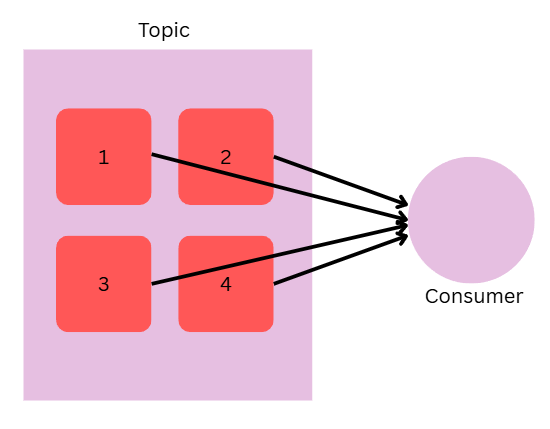
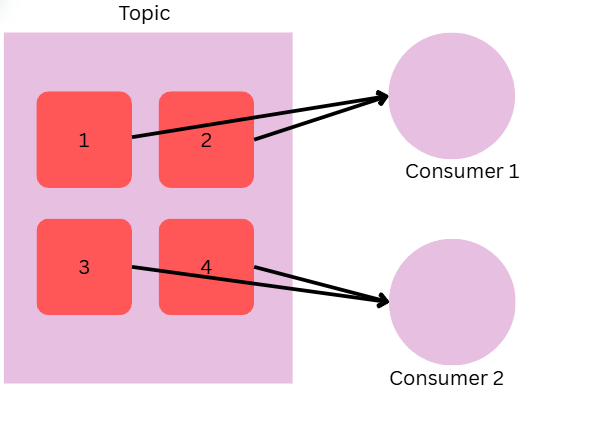
## 2.4. Auto Balancing

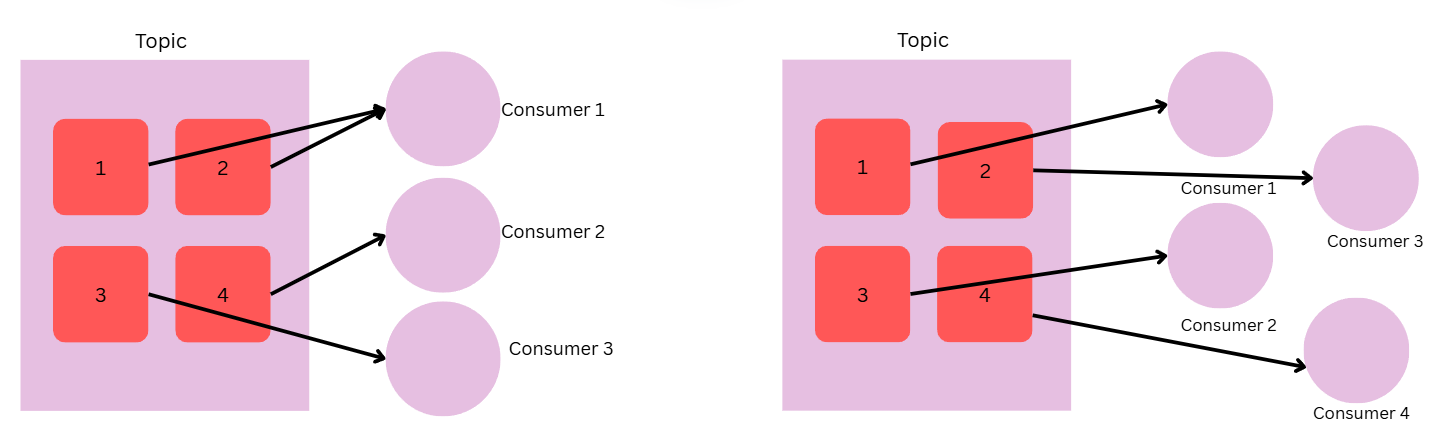
**Auto-balancing in Kafka** refers to the automatic **reassignment of partitions** among consumers in a **consumer group** when changes occur, like:

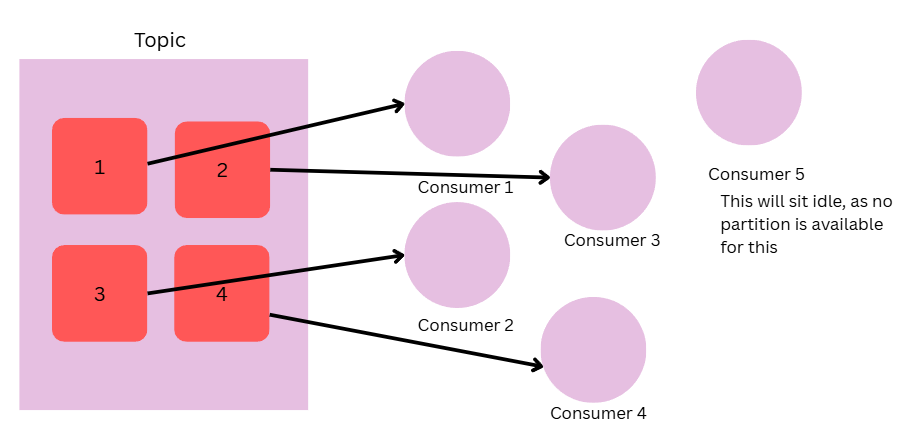
* A new consumer **joins**
* A consumer **leaves or crashes**
* A consumer **restarts**
* 1 consumer: - Can consume multiple partition
* 1 partition: - Can be consumed by only one consumer (in single consumer group), this helps to act it like queue, helps it to act as queue



**Cases of Auto balancing**





This is all happening for consumers in same consumer group, if we have consumers in different consumer group, they can share same partition.

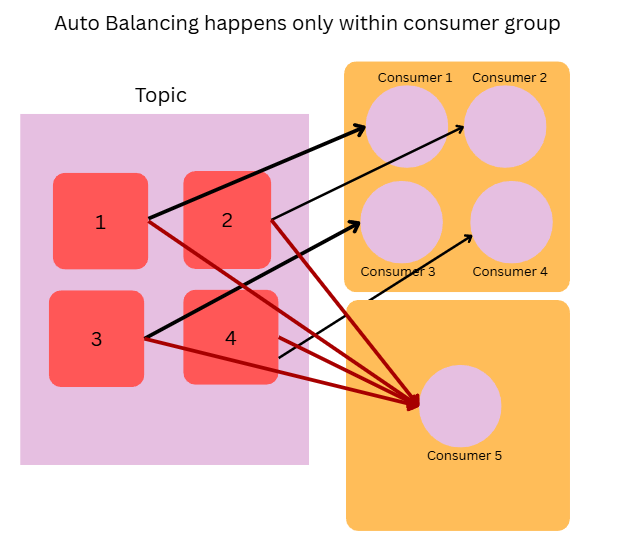
**How it works**

1. **Initial Assignment**  
   When a consumer group starts, Kafka **evenly assigns partitions** of a topic across available consumers.
2. **Dynamic Rebalancing**  
   If a consumer joins or leaves the group:
   * Kafka **triggers a rebalance**
   * Partitions are **re-distributed automatically** among the remaining consumers
3. **Coordinated by Group Coordinator**
   * One broker acts as the **Group Coordinator**
   * It manages the rebalance process using a **leader consumer** in the group

**Impact of Rebalancing**

* During rebalance, consumers **temporarily stop consuming**
* This causes a **brief pause** in data processing
* Too frequent rebalancing can lead to instability

**Behaviour with multiple consumer groups**



# **3. Kafka with Spring**

**1. Prerequisites:**

* Java 17+
* Spring Boot 3.x
* Apache Kafka running locally (or use Docker)

**2. Install Kafka(Docker way):**

docker run -d --name zookeeper -p 2181:2181 zookeeper

docker run -d --name kafka -p 9092:9092 --env KAFKA\_ZOOKEEPER\_CONNECT=host.docker.internal:2181 --env KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://localhost:9092 --env KAFKA\_LISTENERS=PLAINTEXT://0.0.0.0:9092 kafka

**3. Add Dependencies (Maven)**

<dependency>  
 <groupId>org.springframework.kafka</groupId>  
 <artifactId>spring-kafka</artifactId>  
</dependency>

**4. application.yml**

spring:

  kafka:

    bootstrap-servers: localhost:9092

    consumer:

      group-id: my-group

      auto-offset-reset: earliest

      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

    producer:

      key-serializer: org.apache.kafka.common.serialization.StringSerializer

      value-serializer: org.apache.kafka.common.serialization.StringSerializer

**5. Kafka Producer**

@Service  
public class KafkaProducer {  
 private final KafkaTemplate<String, String> kafkaTemplate;  
  
 public KafkaProducer(KafkaTemplate<String, String> kafkaTemplate) {  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 public void sendMessage(String message) {  
 kafkaTemplate.send("test-topic", message);  
 }  
}

**6. Kafka Consumer**

@Service  
public class KafkaConsumer {  
 @KafkaListener(topics = "test-topic", groupId = "my-group")  
 public void listen(String message) {  
 System.*out*.println("Received: " + message);  
 }  
}

**7. REST Endpoint to Send Message**

@RestController  
@RequestMapping("/kafka")  
public class KafkaController {  
 private final KafkaProducer producer;  
  
 public KafkaController(KafkaProducer producer) {  
 this.producer = producer;  
 }  
  
 @PostMapping("/publish")  
 public ResponseEntity<String> send(@RequestParam String message) {  
 producer.sendMessage(message);  
 return ResponseEntity.ok("Sent: " + message);  
 }  
}

**8. Test it**

* Run Spring Boot app
* Hit: POST http://localhost:8080/kafka/publish?message=hello-kafka
* See output in console from the consumer.

# **4. Interview Questions**

**Q1. Explain in brief what Kafka is and how it works.**

**Apache Kafka** is a distributed event streaming platform where:

* **Producers** send messages to **topics**.
* Each **topic** is split into **partitions**, enabling parallelism.
* Messages within a partition are **ordered** and stored **durably**.
* **Brokers** (Kafka servers) manage storage and serve messages to clients.
* **Consumers** read messages from partitions, usually as part of a **consumer group**.
* Kafka ensures **high throughput**, **fault tolerance** via **replication**, and **scalability** by adding more partitions and brokers.

**Workflow:**

1. **Producer** → sends message to **topic** (optionally with a key).
2. Kafka chooses a **partition** (based on key or round-robin).
3. Message is **appended** to that partition log on a **broker**.
4. **Consumer** (from a consumer group) reads messages in order from partition.
5. Kafka retains messages for a configurable **retention period**, regardless of consumption.

**Q2. What are different ways to implement asynchronous communication in microservices?**

* **Message Queues / Brokers**

Tools: **Kafka**, **RabbitMQ**, **ActiveMQ**

Services publish messages to a topic/queue; consumers process them independently.

* **Event-Driven Architecture**

Services emit events (like OrderPlaced, PaymentProcessed) that other services listen to and react upon.

Promotes decoupling and scalability.

* **HTTP with Callback URLs / Webhooks**

Service A calls Service B and passes a callback URL. B calls back when processing is complete.

* **Polling**

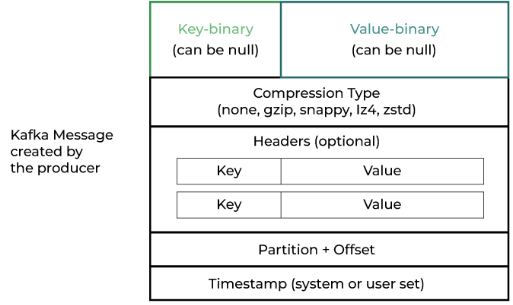
Service A sends a request and periodically checks another endpoint or resource for the result.

* **Reactive Programming (WebFlux / RxJava)**

Asynchronous, non-blocking APIs with backpressure support.

Useful for high-concurrency systems.

**Q3. Explain components of Kafka message.**



**Mandatory Fields in a Kafka Message:**

1. **Value** –

* Required. It is the actual payload/data.
* (Can be null only in case of log compaction to indicate a delete/tombstone message.)

1. **Topic** –

* Required. The destination category to which the message is sent.

**Optional Fields:**

* **Key** – Optional (used for partitioning logic or log compaction).
* **Partition** – Optional (Kafka decides based on key or round-robin if not specified).
* **Headers** – Optional metadata (key-value pairs).
* **Timestamp** – Optional (Kafka will assign if not set).

Notes:

* **Message Key is optional** – it can be null; used to determine the target partition when provided.
* **Message Value is mandatory** – represents the actual data payload (can be null for tombstone messages in log compaction).
* **Each message has an offset** – a unique identifier within a partition for ordering and tracking.
* **Kafka messages include metadata** – such as timestamp, key, value, headers, and offset.
* **Messages are immutable** – once written, they cannot be changed; they stay in the log until retention or compaction removes them.

**Q4. How does Kafka ensure message durability?**

**Kafka ensures message durability** by **persisting messages to disk**, **replicating them across brokers**, and using **acknowledgment settings** to confirm successful writes.

**Key Mechanisms:**

1. **Disk-Based Storage**
   * Messages are written to disk immediately and stored in **append-only logs**.
2. **Replication**
   * Each partition is replicated to multiple brokers (replication.factor) to avoid data loss on broker failure.
3. **Acks Setting in Producer**
   * acks=all ensures that the message is written to all **in-sync replicas** before acknowledgment is returned.
4. **In-Sync Replicas (ISR)**
   * Kafka only considers data committed when all ISR replicas have written it, ensuring reliability.
5. **Retention Policies**
   * Messages are kept for a defined time (or size), even after being consumed, ensuring recoverability.

**Q5. How Kafka achieves high speed and throughput?**

1. **Sequential Disk Writes**

Messages are written to disk in an **append-only** manner—sequential I/O is faster than random I/O.

1. **Zero-Copy Mechanism**

Kafka uses the **sendfile** system call to transfer data directly from the disk buffer to the network, bypassing user space.

1. **Batching of Messages**

Producers batch multiple records before sending, and consumers fetch in batches, reducing I/O and network overhead.

1. **Efficient Storage Format (Segmented Logs)**

Topics are split into **segments** and **partitions**, allowing parallel reads/writes and easier log management.

1. **Asynchronous Processing**

Producers and consumers operate asynchronously, freeing up threads and resources for other tasks.

1. **Compression Support**

Supports message compression (e.g., snappy, lz4, zstd) to reduce payload size and increase throughput.

1. **Horizontal Scalability via Partitions**

More partitions mean more parallelism across brokers, producers, and consumers.

1. **High-Performance Network I/O**

Built using **non-blocking I/O (NIO)** and a **highly optimized protocol** over TCP.

**Q6. What is role of Zookeeper in Kafka?**

1. **Broker Registration & Metadata Management**
   * Keeps track of all active Kafka brokers and their metadata in the cluster.
2. **Leader Election for Partitions**
   * Helps elect the **Kafka controller** (a broker that manages partition leadership and reassignments).
3. **Configuration Management**
   * Stores configuration data for topics, brokers, and access control.
4. **Cluster Health Monitoring**
   * Monitors broker status and triggers rebalancing if a broker fails.
5. **Enables Distributed Coordination**
   * Provides synchronization and coordination between brokers to ensure consistency.

**Q7. What is role of Kafka Broker?**

A Kafka Broker is a Kafka server responsible for storing, receiving, and serving messages to producers and consumers.

1. **Message Storage**

* Stores incoming messages on disk for each partition in an **append-only log**.

2. **Message Serving**

* Delivers stored messages to consumers on request.

3. **Leader for Partitions**

* Acts as a **leader** for some partitions and handles read/write operations for them.

4. **Handles Client Connections**

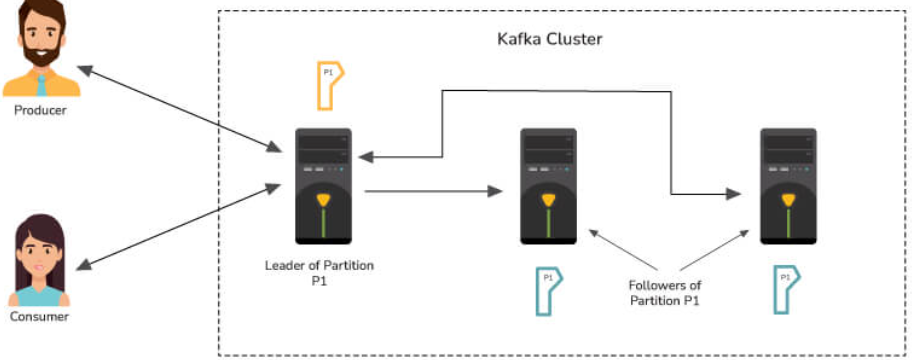
* Accepts connections from **producers**, **consumers**, and **other brokers**.

5. **Replication & Coordination**

* Participates in **replicating partitions** and keeping **in-sync replicas (ISR)** updated.

**Q8. Explain the concept of Leader and Follower in Kafka.**

In Kafka, each partition has one server that acts as a Leader and one or more servers that operate as Followers. The Leader is in charge of all read and writes requests for the partition, while the Followers are responsible for passively replicating the leader. In the case that the Leader fails, one of the Followers will assume leadership. The server's load is balanced as a result of this.



**Q9. Differentiate between RabbitMQ and Kafka.**

In Kafka, each partition has one server that acts as a Leader

**Apache Kafka**

1. **Architecture**:
   * Distributed **log-based** system with **publish-subscribe** model.
2. **Message Storage**:
   * **Persistent, append-only logs**, retained for a configurable time regardless of consumption.
3. **Ordering & Replay**:
   * Maintains strict **ordering within partitions**; supports **message replay** by offset.
4. **Throughput & Performance**:
   * Optimized for **high throughput** and **streaming large volumes of data**.
5. **Use Case**:
   * Best for **real-time data pipelines**, **event sourcing**, **analytics**, and **stream processing**.

**RabbitMQ**

1. **Architecture**:
   * **Message broker** based on **queues** using **AMQP (Advanced Message Queuing Protocol)**.
2. **Message Storage**:
   * **Queue-based**, messages are **removed after consumption** unless persisted manually.
3. **Ordering & Replay**:
   * No guaranteed ordering across consumers; **no native replay** once a message is consumed.
4. **Throughput & Performance**:
   * Optimized for **low-latency transactional messaging** with **routing flexibility**.
5. **Use Case**:
   * Ideal for **short-lived tasks**, **background jobs**, **RPC**, and **real-time notifications**.

**Q10. Explain the four core API architecture that Kafka uses**

* **Producer API:** The Producer API in Kafka allows an application to publish a stream of records to one or more Kafka topics.
* **Consumer API:** An application can subscribe to one or more Kafka topics using the Kafka Consumer API. It also enables the application to process streams of records generated in relation to such topics.
* **Streams API**: The Kafka Streams API allows an application to use a stream processing architecture to process data in Kafka. An application can use this API to take input streams from one or more topics, process them using streams operations, and generate output streams to transmit to one or more topics. The Streams API allows you to convert input streams into output streams in this manner.
* **Connect API:** The Kafka Connector API connects Kafka topics to applications. This opens up possibilities for constructing and managing the operations of producers and consumers, as well as establishing reusable links between these solutions. A connector, for example, may capture all database updates and ensure that they are made available in a Kafka topic.