**KAFKA**

**Official Doc :** https://kafka.apache.org/documentation/

### **1. Topic**

A **Topic** in Kafka is a logical name to which records are published by producers and from which records are consumed by consumers. Topics are multi-subscriber and act as a category or feed name for messages.

* Internally, a topic is split into one or more **partitions**.
* Topics are **immutable** logs, and new data is always appended.

### **2. Partition**

A **Partition** is an ordered, immutable sequence of records that is continually appended to—a structured commit log. Each message within a partition has a unique sequential ID called the **offset**.

* Partitions enable Kafka’s scalability and parallelism.
* A topic with multiple partitions allows multiple consumers to read data in parallel.
* Partitions can be **replicated** across brokers for fault tolerance.

### **3. Consumer Group**

A **Consumer Group** is a group of one or more consumers that coordinate to consume data from a Kafka topic collectively. Kafka guarantees that **each partition is consumed by only one consumer within a group**.

* Each consumer in the group is assigned one or more partitions.
* If all consumers are in the same group, the topic's partitions are effectively load-balanced among them.

### **4. Broker**

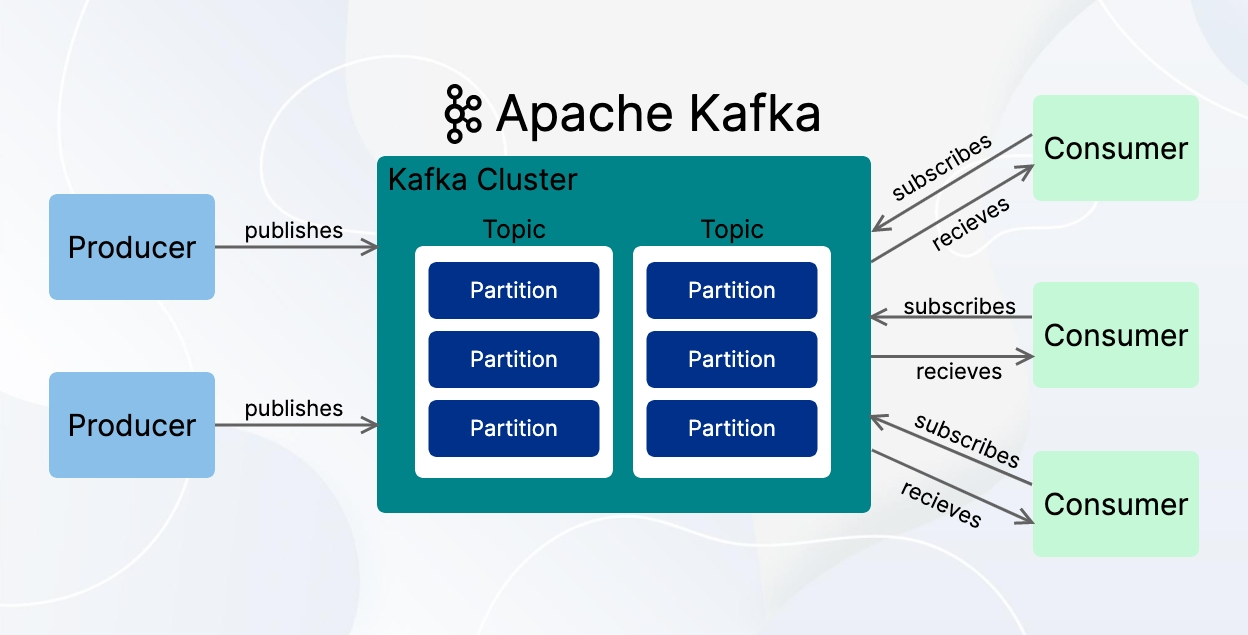
A **Kafka Broker** is a Kafka server that receives messages from producers, stores them on disk, and serves them to consumers. Each broker is identified by a unique ID and can host one or more topic partitions.

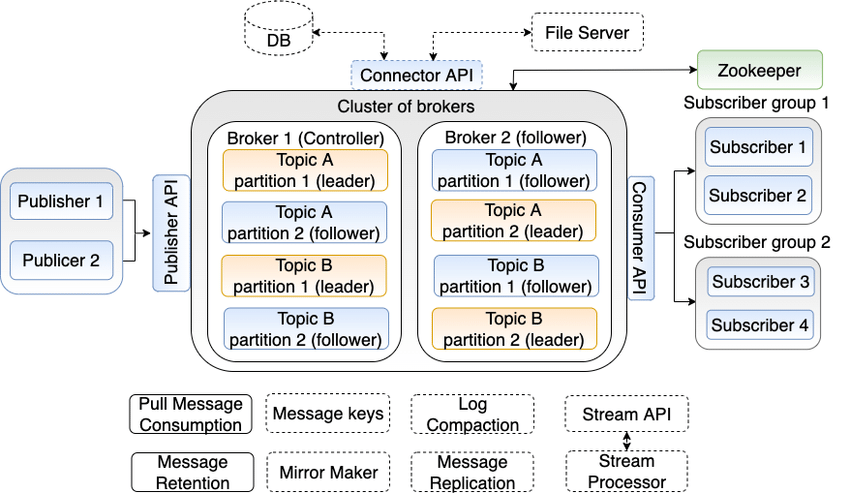
* A Kafka cluster consists of multiple brokers.
* Brokers handle **storage**, **replication**, **request handling**, and **coordination** with the controller node.

### **5. Zookeeper**

**Zookeeper** is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. In Kafka (pre-KRaft), it is used for managing and coordinating Kafka brokers.

* Used for:  
  + Keeping track of broker metadata (e.g., which brokers are alive).
  + Electing a Kafka controller node.
  + Storing topic and partition metadata.
* Kafka is moving toward **KRaft mode**, which eliminates the need for Zookeeper in newer versions.





### 

### **✅ Why Kafka?**

Apache Kafka is widely used in modern software systems for **high-throughput, low-latency, real-time data streaming**. Here's **why Kafka is chosen** over other messaging systems or traditional data pipelines:

## **🔹 1. High Throughput and Low Latency**

Kafka can handle **millions of messages per second** with **low latency** (milliseconds) even with modest hardware.

🏎️ Kafka is built for speed — ideal for real-time analytics, event processing, and streaming.

## **🔹 2. Scalability (Horizontally Scalable)**

Kafka can **scale out** by adding more brokers, partitions, and consumers.

* Topics can have **multiple partitions** spread across brokers.
* Consumers can be grouped into **consumer groups** for parallel processing.

📈 Kafka is suitable for both small-scale systems and large-scale enterprise applications.

## **🔹 3. Durability and Fault Tolerance**

Kafka persists data to disk and **replicates it across multiple brokers**.

* If a broker fails, other replicas take over.
* Messages are not lost — durable storage + replication = resilience.

💾 Messages are safe, even in case of hardware failures.

## **🔹 4. Decoupling of Systems**

Kafka acts as a **buffer layer** between producers (writers) and consumers (readers), enabling **loose coupling**.

* Producers don’t need to know who the consumers are.
* Consumers can read data independently, even at different speeds.

🔄 Perfect for microservices, where components evolve independently.

## **🔹 5. Replayability of Events**

Kafka stores messages with **offsets** and allows consumers to **re-read** or **replay** data from any point in time.

🔁 Helps in debugging, auditing, and rebuilding systems from scratch.

## **🔹 6. Stream Processing**

Kafka integrates tightly with stream processing tools like:

* **Kafka Streams**
* **ksqlDB**
* **Apache Flink**
* **Apache Spark**

🔄 Enables transformation, filtering, aggregation of streaming data in real-time.

## **🔹 7. Multiple Consumer Support**

Kafka supports **multiple independent consumer groups**, allowing the same stream to be processed in multiple ways.

📊 One group could do logging, another does fraud detection — all from the same topic.

## **🔹 8. Ecosystem Integration**

Kafka integrates well with:

* Databases (PostgreSQL, MySQL)
* Big Data tools (Hadoop, Spark)
* Monitoring (Prometheus, Grafana)
* Cloud platforms (AWS, GCP, Azure)

🧩 Kafka fits into almost any modern data stack.

## **🔹 9. Backpressure Handling**

Kafka decouples producer and consumer rates:

* Producers can publish messages even if consumers are slow.
* Consumers can catch up later from disk-based logs.

## **🔹 10. Open Source and Community Driven**

* Backed by the **Apache Software Foundation**.
* Large, active community with support from companies like **Confluent**.

### **🔚 In short:**

Kafka is **fast**, **durable**, **scalable**, and **flexible** — making it ideal for **real-time data pipelines**, **event-driven architectures**, **log aggregation**, **IoT**, **microservices communication**, and much more.

Great question! These are fundamental performance metrics in **distributed systems** like Kafka, networks, APIs, databases, etc.

## **🔹 Throughput**

**Throughput** is the amount of data processed over a specific period of time.

* **Measured in**:  
  + messages per second
  + requests per second (RPS)
  + MB/s or GB/s
* **In Kafka**:  
   Throughput is how many **messages** a producer can send or a consumer can read **per second**.

✅ **Example**:  
 If Kafka processes **10,000 messages per second**, that’s its throughput.

✅ **Analogy**:  
 Think of water flowing through a pipe. The **throughput** is how many liters per second flow through the pipe.

## **🔹 Latency**

**Latency** is the **time delay** between an action and its result.

* **Measured in**: milliseconds (ms) or microseconds (µs)
* **In Kafka**:  
   Latency can be:  
  + **End-to-end latency**: time between when a producer sends a message and a consumer receives it.
  + **Producer latency**: time to write a message to a broker.
  + **Consumer latency**: time to fetch messages after they’re available.

✅ **Example**:  
 If a message is produced at 12:00:00.100 and consumed at 12:00:00.120, latency = **20 ms**.

✅ **Analogy**:  
 In a restaurant, **latency** is the time between placing your order and the food arriving.

## **🔄 Throughput vs Latency (Key Difference)**

| **Metric** | **Meaning** | **Goal** |
| --- | --- | --- |
| Throughput | Volume of data processed per second | **Maximize** (handle more) |
| Latency | Time delay in data delivery or response | **Minimize** (respond faster) |

## **💡 Kafka Optimization Insight:**

* Kafka is optimized for **high throughput**, even if it adds a **little latency**.
* Ideal for **real-time pipelines** where you want **fast & massive data ingestion**.

**Is Kafka is used in replacement of Database**

❌ **Kafka is not a replacement for a database.** ✅ But Kafka **can complement** databases or **act as a buffer** before data is written to them.

### **🔍 Let's break it down:**

| **Feature** | **Kafka** | **Database** |
| --- | --- | --- |
| **Purpose** | Real-time streaming / event messaging | Persistent, queryable data storage |
| **Data Access** | Sequential read via consumers | Random access via queries (SQL/NoSQL) |
| **Storage** | Log-based, append-only (short/long-term) | Long-term, structured storage |
| **Querying** | No rich querying (only by offset/topic) | Supports SQL, indexing, filtering |
| **Mutability** | Immutable messages | Supports updates, deletes |
| **Transactions** | Limited transaction support | Full ACID transactions (for relational DBs) |
| **Use Case** | Streaming, messaging, decoupling | Storing structured business-critical data |

### **✅ Use Kafka for:**

* Real-time **event streaming**
* **Data pipelines** between services or systems
* **Log aggregation**
* **Change Data Capture** (e.g., listen to DB changes and act on them)
* **Decoupling microservices**

### **✅ Use a Database for:**

* **Persistent** structured data
* **Transactional consistency**
* **Querying/filtering/searching**
* Reporting, dashboards, and analytics

### **✅ Common Pattern: Kafka + Database**

Many systems use **Kafka as a buffer or ingestion layer**, and then write the data to a database for long-term storage and querying.

📦 Example:  
 Producer (e.g., e-commerce app) → Kafka → Consumer (ETL Service) → PostgreSQL

### **🔁 In Summary:**

**Kafka is a real-time data streaming platform**, not a storage and query engine.  
 Use **Kafka + Database** together to build powerful, scalable, and real-time systems.

### **✅ KRaft in Kafka (Kafka Raft Metadata Mode)**

### **🔹 What is KRaft?**

**KRaft** (Kafka Raft Metadata Mode) is a **Zookeeper-free** mode introduced in Apache Kafka to manage metadata **natively within Kafka itself** using the **Raft consensus protocol**.

🧠 It stands for **Kafka Raft**, and it replaces the need for **Zookeeper** in Kafka.

**🔹 Why was KRaft introduced?**

Kafka traditionally relied on **Zookeeper** for:

* Broker registration
* Controller election
* Topic metadata management

But this setup had **complexities**:

* Hard to scale/manage Zookeeper clusters
* Operational overhead and failure points
* Kafka and Zookeeper needed **separate configs and deployments**

🔄 KRaft simplifies the architecture by **removing Zookeeper entirely**.

### **🔹 How KRaft Works (Simplified)**

* Kafka uses a **Raft-based quorum** (KRaft quorum) for:  
  + Metadata storage
  + Leader election
  + Cluster coordination
* One broker acts as the **KRaft controller leader**, others are followers.
* Metadata is stored in an **internal Kafka topic**, replicated across KRaft quorum nodes.

### **🔹 Key Components of KRaft Mode**

| **Component** | **Role** |
| --- | --- |
| **Controller** | Elected via Raft; manages metadata and broker coordination |
| **Quorum nodes** | Brokers with process.roles=controller; participate in Raft quorum |
| **Metadata log** | Internal log topic (@metadata) where Kafka stores cluster metadata |

### **🔹 Benefits of KRaft Mode**

✅ **Zookeeper not required** ✅ **Simplified deployment and configuration** ✅ **Unified metadata management** ✅ **Faster failover** ✅ **Scales better** with larger clusters

### **🔹 Kafka Version Support**

| **Kafka Version** | **KRaft Support** |
| --- | --- |
| 2.8.x | Experimental |
| 3.0–3.3 | Production-ready (optional) |
| 3.4+ | Recommended |
| 4.0+ (future) | **Zookeeper fully removed** |

### **🔹 KRaft Deployment Modes**

1. **KRaft Mode Only** – no Zookeeper (recommended from Kafka 3.3+)
2. **Zookeeper Mode** – legacy, still supported (until Kafka 4.0)
3. **Dual Mode** – for migration/testing (not recommended long-term)

### **🔹 How to Enable KRaft**

In server.properties:

process.roles=controller,broker

node.id=1

controller.quorum.voters=1@localhost:9093,2@localhost:9094

* Start brokers with --override flags for roles and ports
* Ensure a quorum of controllers is up and reachable

### **🔚 Summary**

**KRaft simplifies Kafka by removing Zookeeper** and internally managing metadata with a Raft-based consensus protocol.  
 It's more scalable, reliable, and easier to operate.

📌 Producers send messages to **topics**,  
 🔁 Topics are split into **partitions**,  
 📍 Partitions are stored and managed by **brokers**,  
 📥 Consumers fetch messages from brokers via **topics**.