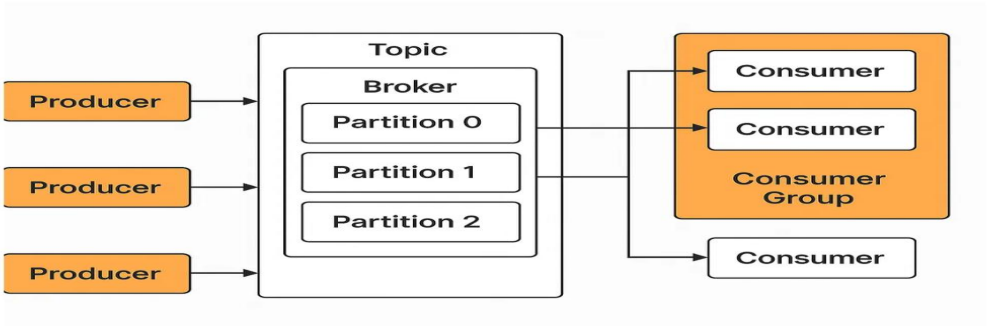
Apache Kafka

- Apache Kafka is a distributed event streaming platform widely used for building real-time data pipelines and event-driven architectures.
- It operates as a highly scalable, fault-tolerant, and durable commit log where
  - events are published to topics,
  - partitioned for speed, and
  - stored across a cluster of brokers.



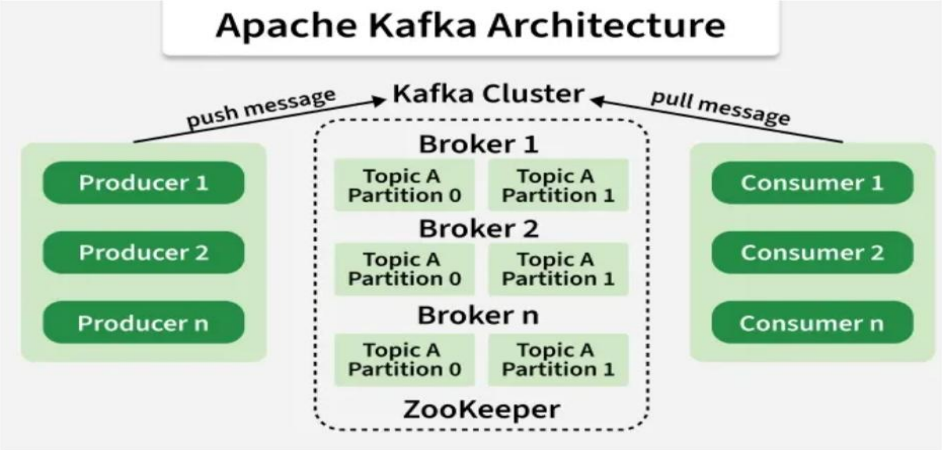
1. Kafka Architecture Overview

- Kafka follows a publish-subscribe model where producers send messages to Kafka topics, and consumers read from these topics.

- **Key Components:**
-
- **Broker**: Kafka runs as a cluster of brokers (servers) where each broker handles storage, message distribution, and replication.
- **Topic:** A logical entity where messages are stored.
- **Partition:** A topic is split into multiple partitions for scalability and parallel processing.
- **Producer:** Sends messages to Kafka topics.
- **Consumer:** Reads messages from Kafka topics.
- **Consumer Group:** A set of consumers sharing a topic's partitions.
- **Zookeeper:** Manages metadata, leader election, and cluster coordination.
- **Offset:** Keeps track of the last read message in a partition.
- **Replication:** Ensures fault tolerance by duplicating partitions across multiple brokers.
- **Controller Node:** Manages partition leader elections and cluster metadata.

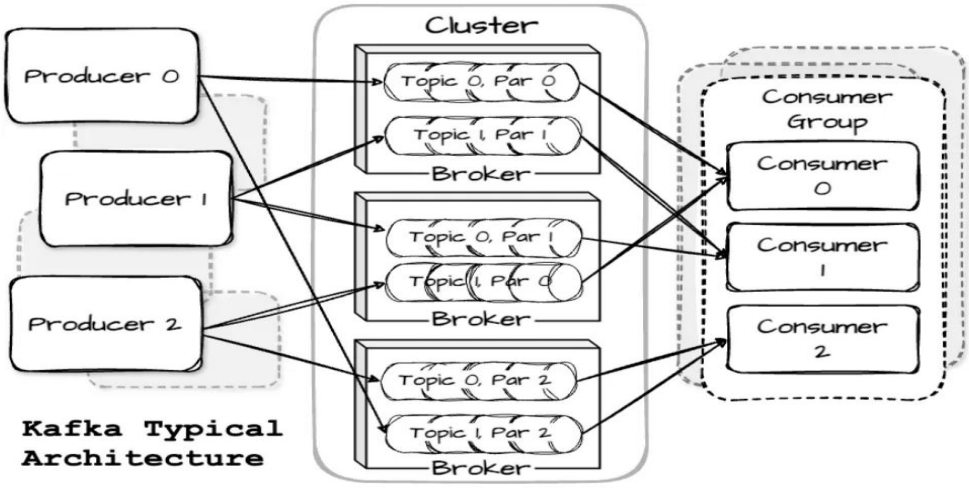
Kafka Architecture Diagram:

- (Diagram showing producers sending messages to Kafka topics, which are partitioned and stored in brokers, and consumers consuming data as part of consumer groups.)



- On a high level, in one kafka cluster, there can be more than one Brokers. DIfferent partitions of topic are distributed along all the brokers. look at below image, little complex to understand.

2. Producers: Writing Messages to Kafka



3. <u>Kafka Topics and Partitions</u>

- Kafka topics are split into partitions to enable parallelism and scalability.
- Each partition is stored on a broker.
- Each message in a partition is assigned a sequential offset.
- Partitions enable load balancing by distributing data across brokers.

4. <u>Producers: Writing Messages to Kafka</u>

- Producers are responsible for sending messages to Kafka topics.
- Key-based partitioning: If a message has a key (e.g., UserID), Kafka ensures all messages with the same key go to the same partition.

- Round-robin partitioning: If no key is provided, messages are distributed randomly across partitions.
- Acks & Durability: The producer can wait for acknowledgments (acks=all) to ensure reliability.
- Compression: Producers can compress messages (gzip, snappy, etc.) to reduce network load and storage.
- Batching: Kafka allows batching messages to improve efficiency and reduce latency.

5. Consumers and Consumer Groups

- Consumers read messages from Kafka topics, and they belong to consumer groups for better scalability.
- Each consumer in a group reads from a unique set of partitions.
- If a topic has 3 partitions and a consumer group has 3 consumers, each consumer gets one partition.
- If there are more consumers than partitions, some consumers will be idle.
- If there are fewer consumers than partitions, some consumers will handle multiple partitions.

6. Rebalancing in Kafka

- Rebalancing occurs when a consumer joins, leaves, or fails in a consumer group. Kafka redistributes partitions among the remaining consumers.
- Example:
- Initially, Consumer A is assigned Partition-0, and Consumer B is assigned Partition-1.
- If Consumer A fails, Kafka reassigns Partition-0 to Consumer B.
- To avoid frequent rebalancing, Kafka provides Sticky Assignor and Incremental Cooperative Assignor to minimize disruptions

7. Kafka Offset Management

- Offsets track the last consumed message in each partition.
- Kafka stores offsets in a special topic (__consumer_offsets).
- Each consumer group maintains its own offsets, ensuring different groups consume data independently.
- Consumers can manually commit offsets (for precise control) or automatically commit (enable.auto.commit=true).
- Offset Retention & Recovery
- If a consumer crashes and restarts, it can resume from the last committed offset.
- If offsets are not found, Kafka follows auto.offset.reset policy:
  - earliest → Start from the beginning.
  - latest (default) → Start from the newest messages.

8. Kafka Replication & Fault Tolerance

- To ensure high availability, Kafka supports replication.
- Each partition has one leader and multiple followers.

- Only the leader handles reads/writes, and followers replicate data.
- If a leader fails, Kafka elects a new leader from the followers.
- The min.insync.replicas setting controls the number of replicas that must acknowledge a write for durability.

Kafka APIs

- Kafka provides several APIs to interact with the system:
- Producer API: Allows applications to send streams of data to topics in the Kafka cluster. It handles the serialization of data and the partitioning logic.
- Consumer API: Allows applications to read streams of data from topics. It manages the offset of the data read, ensuring that each record is processed exactly once.
- Streams API: A Java library for building applications that process data in real-time. It allows for powerful transformations and aggregations of event data.
- Connector API: Provides a framework for connecting Kafka with external systems.
    - Source connectors import data from external systems into Kafka topics, while sink connectors export data from Kafka topics to external systems.

Interactions in the Kafka Architecture

- Producers to Kafka Cluster: Producers send data to the Kafka cluster. The data is published to specific topics, which are then divided into partitions and distributed across the brokers.
- Kafka Cluster to Consumers: Consumers read data from the Kafka cluster. They subscribe to topics and consume data from the partitions assigned to them. The consumer group ensures that the load is balanced and that each partition is processed by only one consumer in the group.
- ZooKeeper to Kafka Cluster: ZooKeeper coordinates and manages the Kafka cluster. It keeps track of the cluster's metadata, manages broker configurations and handles leader elections for partitions.

Three common Kafka Architectures

- Pub-Sub Systems,
- Stream Processing Pipelines and
- Log Aggregation Architectures.

1. **Pub-Sub Systems**

- In a publish-subscribe (pub-sub) system, producers publish messages to topics and consumers subscribe to those topics to receive the messages. Kafka's architecture is well-suited for pub-sub systems due to its ability to handle high volumes of data and provide reliable message delivery.
- Key Components
- Producers: Applications that send data to Kafka topics.
- Topics: Logical channels to which producers send data and from which consumers read data.
- Consumers: Applications that subscribe to topics and process the data.

- Consumer Groups: Groups of consumers that share the load of reading from topics.
- A real-world example of a pub-sub system using Kafka could be a news feed application where multiple news sources (producers) publish articles to a topic and various user applications (consumers) subscribe to receive updates in real-time.

2. Stream Processing Pipelines

- Stream processing pipelines involve continuously ingesting, processing and transforming data in real-time. Kafka's ability to handle high-throughput data streams and its integration with stream processing frameworks like Apache Flink and Apache Spark make it ideal for building such pipelines.
- Key Components
- Producers: Applications that send raw data streams to Kafka topics.
- Topics: Channels where raw data is stored before processing.
- Stream Processors: Applications or frameworks that consume raw data, process it and produce transformed data.
- Sink Topics: Topics where processed data is stored for further use.

3. Log Aggregation Architectures

- Log aggregation involves collecting log data from various sources, centralizing it and making it available for analysis. Kafka's durability and scalability make it an excellent choice for log aggregation systems.
- Key Components
- Log Producers: Applications or services that generate log data.
- Log Topics: Kafka topics where log data is stored.
- Log Consumers: Applications that read log data for analysis or storage in a centralized system.
- A real-world example of a log aggregation architecture using Kafka could be a microservices-based application where each microservice produces logs. These logs are sent to Kafka topics and a centralized logging system (like ELK Stack) consumes the logs for analysis and monitoring.

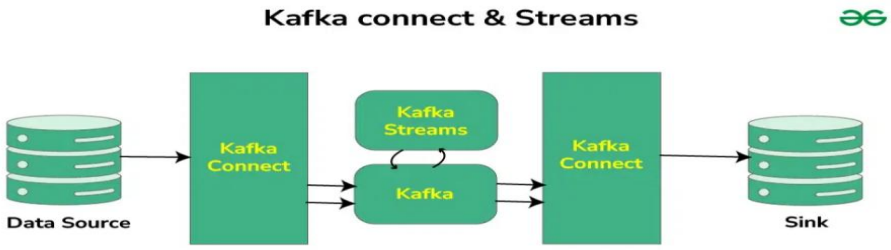
Key Features of Kafka Architecture

- High Throughput and Low Latency: Kafka is designed to handle high volumes of data with low latency. It can process millions of messages per second with latencies as low as 10 milliseconds.
- Fault Tolerance: Kafka achieves fault tolerance through data replication. Each partition can have multiple replicas and Kafka ensures that data is replicated across multiple brokers. This allows the system to continue operating even if some brokers fail.
- Durability: Kafka ensures data durability by persisting data to disk. Data is stored in a log-structured format, which allows for efficient sequential reads and writes.
- Scalability: Kafka's distributed architecture allows it to scale horizontally by adding more brokers to the cluster. This enables Kafka to handle increasing amounts of data without downtime.

- Real-Time Processing: Kafka supports real-time data processing through its Streams API and ksqlDB, a streaming database that allows for SQL-like queries on streaming data.

Apache Kafka Frameworks

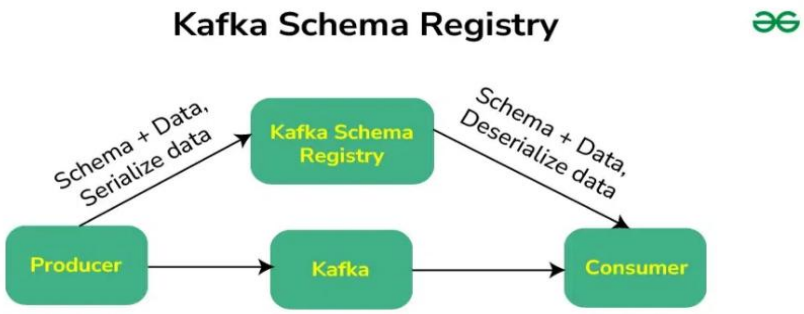
- Kafka is a distributed streaming platform that can be extended and integrated with various frameworks to extend its capabilities and integrate with other systems.



Apache Kafka Frameworks

- Some of the key frameworks in Kafka ecosystem include:
- 1. Kafka Connect is a tool in the Kafka ecosystem that enables reliable and scalable data integration between Kafka and external systems like databases or file systems. It offers built-in connectors to simplify the process of moving data in and out of Kafka.
- 2. Kafka Streams is a client library for building applications that process and analyze data in Kafka topics. It provides easy-to-use APIs for tasks like filtering, joining and aggregating streaming data.

3.Schema Registry (part of the Confluent Platform) is a centralized service that manages Avro schemas for Kafka messages, ensuring producers and consumers use compatible data formats during serialization and deserialization.



Schema Registry

Advantages of Kafka Architecture

- Decoupling of Producers and Consumers: Kafka decouples producers and consumers, allowing them to operate independently. This makes it easier to scale and manage the system.
- Ordered and Immutable Logs: Kafka maintains the order of records within a partition and ensures that records are immutable. This guarantees the integrity and consistency of the data.
- High Availability: Kafka's replication and fault tolerance mechanisms ensure high availability and reliability of the data.