@mohdshakir6432

Apache Kafka

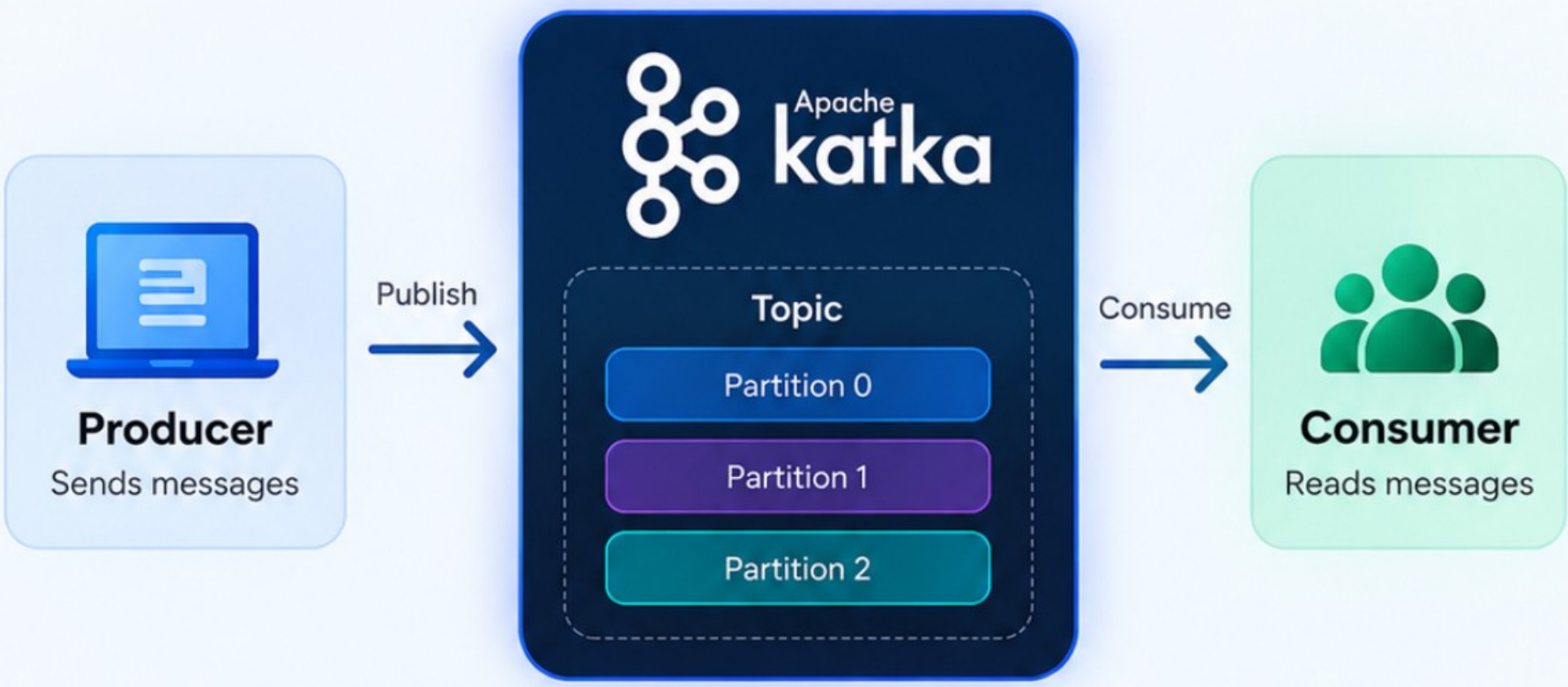
Understand Kafka from
Producer to Consumer



Swipe

What is Apache Kafka?

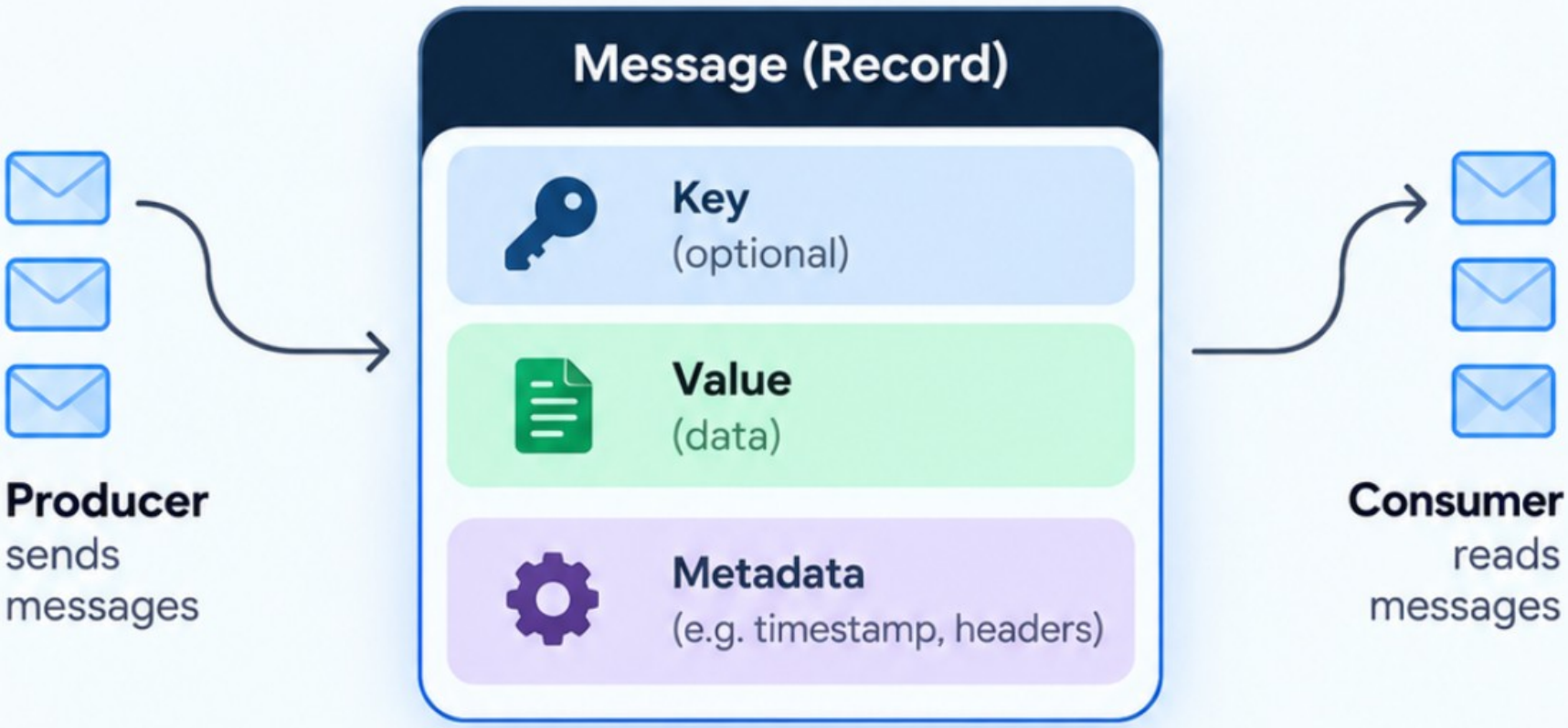
Apache Kafka is a **distributed event streaming platform.**



Helps applications send, store and process **real-time data** at scale.

Messages

Kafka's unit of data is called a **message** (or record).

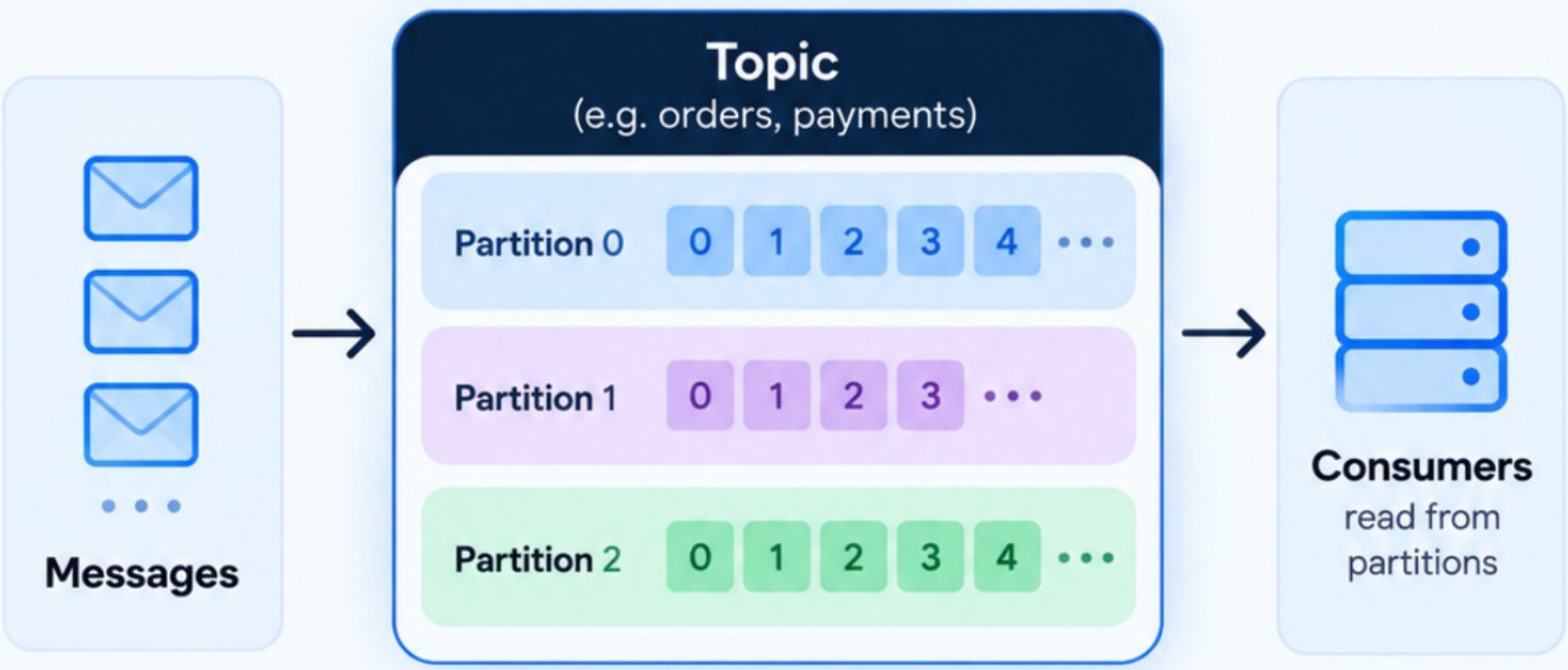


Key Points

- A message has an **optional key**.
- Key and value are just **arrays of bytes**.
- Key helps in **partitioning** (optional).

Topics & Partitions

A **topic** is a logical stream of messages, divided into multiple **partitions**.



Key Takeaway

- A topic can have multiple partitions.
- Each partition is an **ordered, immutable log** of messages.
- New messages are appended to the end of a partition.

How Kafka Stores Data

Kafka uses the local **filesystem** and OS **page cache** for storage.

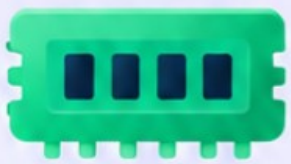
Kafka Broker



Receives messages from producer

Write →

OS Page Cache



Data is first written to memory (page cache)

Flush →

Disk



Data is stored as segment files on disk

Partition on Disk



Key Takeaway

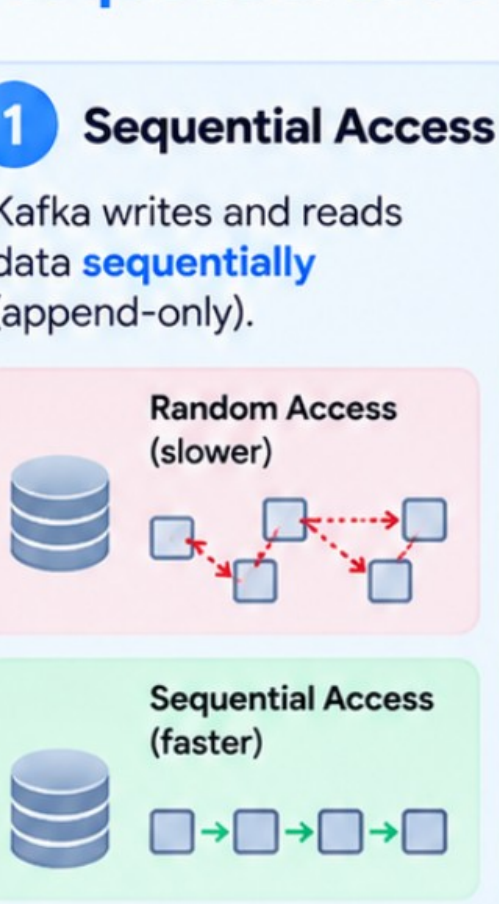
Kafka relies on the local filesystem with OS page cache to achieve **high throughput** and **durability**.

Why Kafka is Fast?

Kafka achieves high performance due to **sequential access**, **zero-copy** and **batching**.

1 Sequential Access

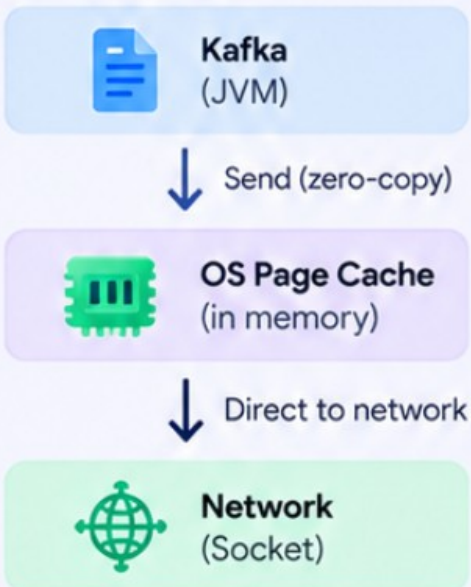
Kafka writes and reads data **sequentially** (append-only).



Sequential writes are much **faster** than random writes.

2 Zero-Copy

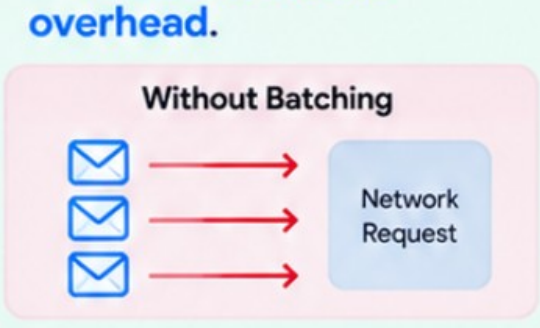
Kafka uses OS page cache to avoid **unnecessary data copies**.



Data is sent directly from page cache to network **without extra copies**.

3 Batching

Multiple messages are sent together in a batch to **reduce network overhead**.



Batching reduces the number of network requests, improving **throughput**.

Key Takeaway

Sequential access, zero-copy and batching help Kafka achieve **high throughput** and **low latency**.

How a Producer Sends a Message?

A producer goes through multiple steps before a message is written to a Kafka broker.

ProducerRecord
(Topic, Key, Value, ...)

Serializer
Converts data to bytes

Partitioner
Decides which partition

Batch
Messages are batched

Kafka Broker
Receives and stores

Response
Acknowledgement (based on acks)

1 ProducerRecord
You create a ProducerRecord with **topic**, **key** (optional), **value** and other metadata.

2 Serializer
The key and value are converted into **byte arrays** using a serializer (e.g. StringSerializer, AvroSerializer).

3 Partitioner
If a key is provided, the partitioner **decides which partition** the message goes to (otherwise round-robin or sticky partitioning).

4 Batch
Messages are accumulated into a **batch** to reduce network overhead.

5 Kafka Broker
The batch is sent to the selected broker, which **writes the messages** to the partition on disk (via page cache).

6 Response
The broker sends an **acknowledgement** based on the acks configuration.

Key Takeaway

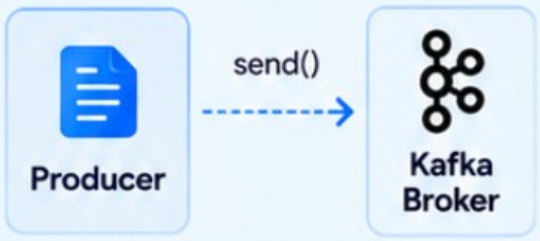
A producer **serializes**, **partitions** and **batches** messages before sending them to the broker, and receives an acknowledgement based on the **acks** setting.

How can a Producer Send a Message?

Kafka provides different ways to send a message, depending on your **performance** and **reliability** needs.

1 Fire-and-Forget

Send the message and **don't wait** for any response.

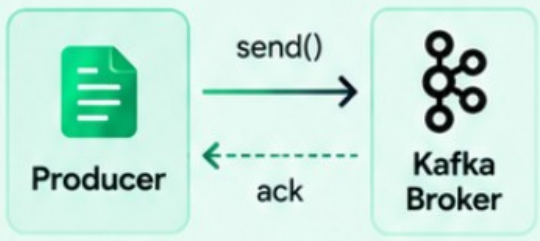


- Fastest method
- No acknowledgement
- Use when occasional message loss is acceptable

```
producer.send(record);
```

2 Synchronous

Send the message and **wait** for the broker response.

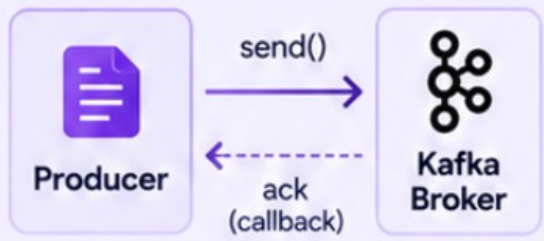


- Waits for acknowledgement
- Can handle errors
- Slower than fire-and-forget
- Use when reliability is important

```
producer.send(record)
        .get();
```

3 Asynchronous

Send the message and continue without waiting (use a **callback**).



- Non-blocking
- Uses a callback to handle success or failure
- Better performance
- Use in most real-world applications

```
producer.send(record,
  (metadata, exception) -> {
    if (exception == null) {
      System.out.println("Success");
    } else {
      System.out.printld("Failed");
    }
 });
```

Key Takeaway

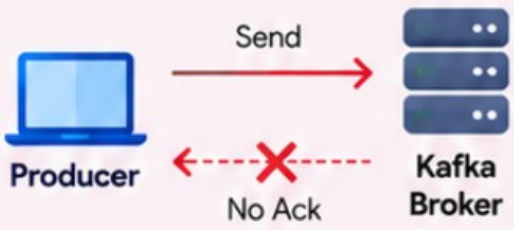
Choose the sending method based on your use case — **speed vs reliability**.

Was the message delivered successfully?

The **acks** configuration controls when the producer receives an acknowledgement from Kafka.

acks = 0

Don't wait for any acknowledgement.

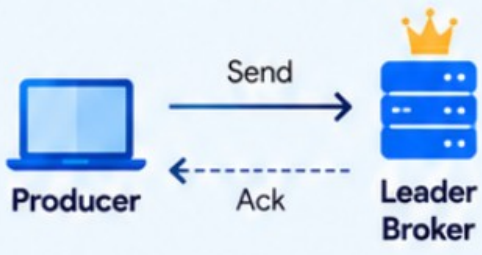


- Producer does not wait for a response.
- Message may be lost if a broker fails.
- Lowest latency (highest performance).
- Use when occasional message loss is acceptable.

Best for: High throughput and non-critical data.

acks = 1

Wait for acknowledgement from the leader broker.

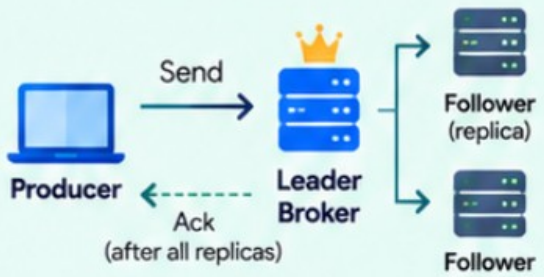


- Producer waits for ack from the leader broker only.
- Message is safe if leader has received it.
- May be lost if the leader fails before replicating to followers.
- Good balance between performance and reliability.

Best for: Most use cases (default in many setups).

acks = all

Wait for acknowledgement from all in-sync replicas (ISR).



- Producer waits for acks from all in-sync replicas.
- Message is highly durable.
- Lowest risk of data loss.
- Higher latency compared to acks=1.
- Recommended for critical data.

Best for: Critical data where durability is important.

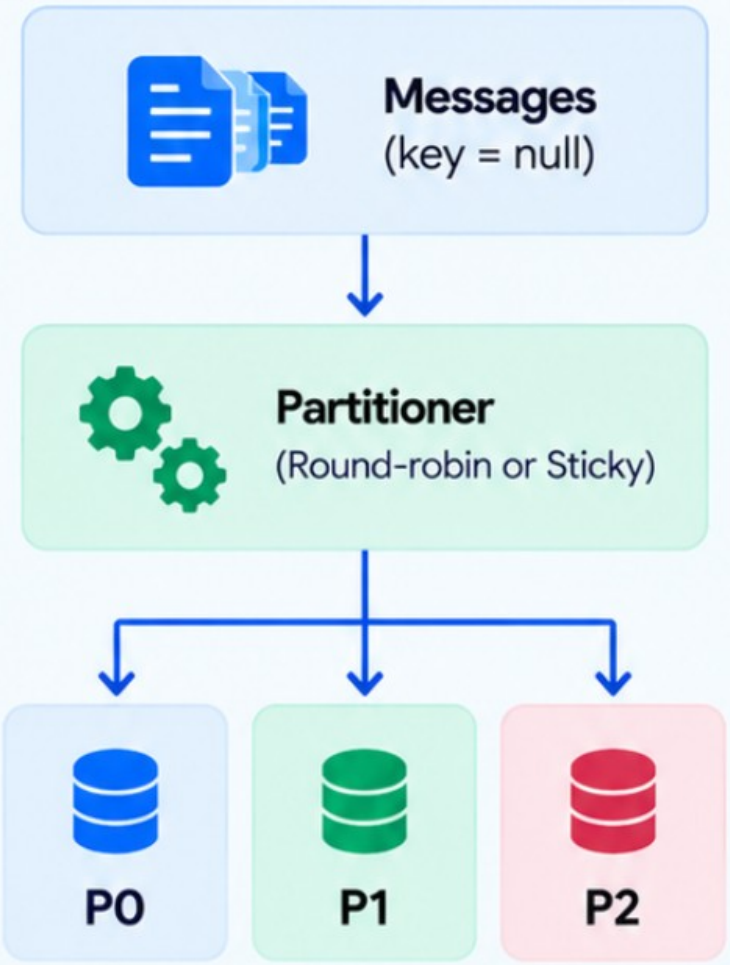
Key Takeaway

The **acks** setting controls the trade-off between **performance** and **durability**.

How does Kafka distribute a message?

When a message is sent without a key (key = null), Kafka uses a **partitioning strategy** to decide which partition the message should go to.

No-Key Message Flow



1 Round-Robin Partitioning

In older versions, when no key is provided, Kafka uses a **round-robin algorithm** to distribute messages across partitions sequentially.



- Messages are sent to partitions in sequence.
- After the last partition, it starts again from the first.
- Ensures even distribution of messages.

2 Sticky Partitioning

In newer versions, Kafka uses **sticky partitioning** for better performance. It keeps sending messages to the same partition until the batch is full, then switches to the next partition.



- Sends multiple consecutive messages to the same partition.
- Reduces network overhead and improves throughput.
- Still ensures good distribution across partitions over time.

Key Takeaway

When key is null, Kafka uses **round-robin** (older) or **sticky partitioning** (newer) to distribute messages across partitions.

What happens when a message has a key?

When a message has a key, Kafka uses a **hashing algorithm** to determine which partition the message should go to.

Key-Based Message Flow

Message
(key = user123, value = ...)

↓

Key
user123

↓

Hash Function
Converts the key to a numeric hash value (e.g. hash("user123"))
hash = 432781

↓

Partition
partition = hash % number_of_partitions
e.g. 432781 % 3 = 1 → **Partition 1**

1 Hashing Algorithm

Kafka applies a hash function (e.g. Murmur2) on the message key to generate a numeric hash value.

```
int hash = Utils.murmur2(keyBytes);
int partition = Math.abs(hash) % number_of_partitions;
```

2 Same Key → Same Partition

The same key will always be mapped to the same partition (as long as the number of partitions doesn't change). This ensures that all messages for a given key are stored in order.

user123	→	Partition 1
user123	→	Partition 1
user123	→	Partition 1

Same key always goes to the same partition

3 Different Keys → Different Partitions

Different keys can be mapped to different partitions depending on their hash values.

user123	→	Partition 1
user456	→	Partition 2
user789	→	Partition 0

Different keys may go to different partitions

Key Takeaway

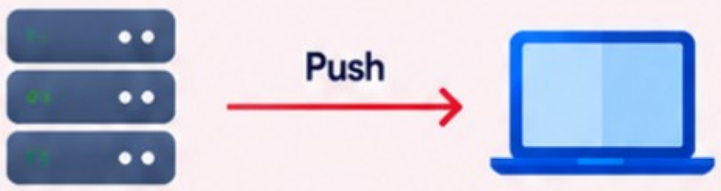
Kafka uses a hash of the message key to determine the partition, ensuring that the **same key always maps to the same partition** and maintains message ordering.

Consumer Model in Kafka
Push vs Pull

Kafka uses a **pull model** (not push) for consumers, which gives more control and better reliability.

PUSH

Broker pushes messages to consumer

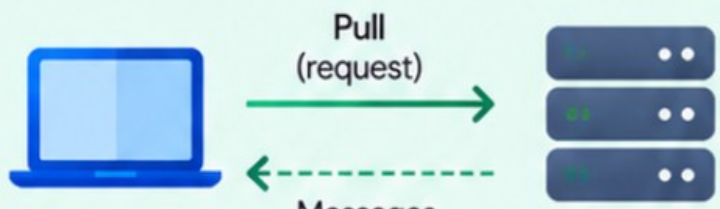


- Broker actively sends messages to the consumer.
- Consumer has limited control over the rate of messages.
- If consumer is slow, it can get overwhelmed.
- Can lead to higher risk of failures or message loss (in some systems).
- Not used in Kafka.

VS

PULL

Consumer pulls messages from broker



- Consumer sends a request to fetch messages.
- Consumer can read messages at its own pace (based on its capacity).
- Prevents consumer from getting overwhelmed.
- More control, better reliability and fault tolerance.
- **This is the model used in Kafka.**

Key Takeaway

Kafka uses a **pull model**, allowing consumers to read messages according to their **own processing capacity**, which improves reliability and system stability.

How does a Consumer Read Messages?

A consumer sends a request to the broker with a **starting offset**, and the broker returns the messages from that offset.

Consumer Read Flow



Consumer

1 **poll(offset)**
Send request with starting offset to the broker.



Kafka Broker

2 Fetch messages from that offset.



Messages

What is Offset?

- Offset is a unique, sequential number assigned to each message within a partition.
- It indicates the position of a message in the partition.
- Consumer specifies the starting offset in its request.
- After reading, the consumer can commit the offset to keep track of the progress.

Messages in a Partition (with offsets)

0	Message A
1	Message B
2	Message C
3	Message D
4	Message E

Read in sequence

Sequential Consumption (Within a Partition)

- Messages are read in order of offsets (0, 1, 2, 3 ...).
- A consumer always reads from the specified offset onwards.
- Each message in a partition is delivered only once to a consumer group (unless the offset is reset).
- If a consumer requests offset 2, it will receive messages 2, 3, 4 ... in that partition.

Example Request (Conceptual)

```
{
  "topic": "orders",
  "partition": 0,
  "offset": 2,
  "maxRecords": 500
}
```

- **topic:** which topic to read
- **partition:** which partition
- **offset:** starting offset
- **maxRecords:** maximum number of messages to fetch



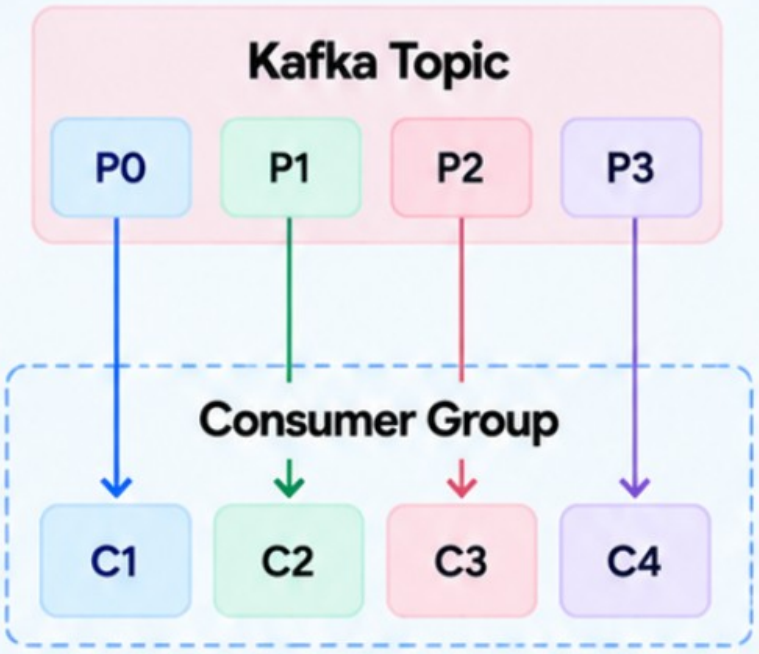
Key Takeaway

A consumer reads messages by sending a request with a **starting offset**, and it consumes messages sequentially within a partition using **polling**.

What is a Consumer Group?

A consumer group is a logical group of one or more consumers that subscribe to topics and work together to consume messages.

How Consumer Groups Work



Each partition is consumed by **one consumer** within the group.

Key Points

1. **One group → one or more consumers**
   A consumer group can have multiple consumers working together.
2. **Group subscribes to topics**
   All consumers in the group subscribe to the same topic(s).
3. **Consumers share partitions**
   Partitions of the topic are distributed among consumers in the group.
4. **One partition → one consumer (within a group)**
   A partition can be consumed by only one consumer in a group at a time.
5. **Scalability**
   Adding more consumers to the group allows parallel processing and higher throughput.

Example



Each partition is assigned to one consumer in the group.

Consumers in a different group can read the same topic independently.

Key Takeaway

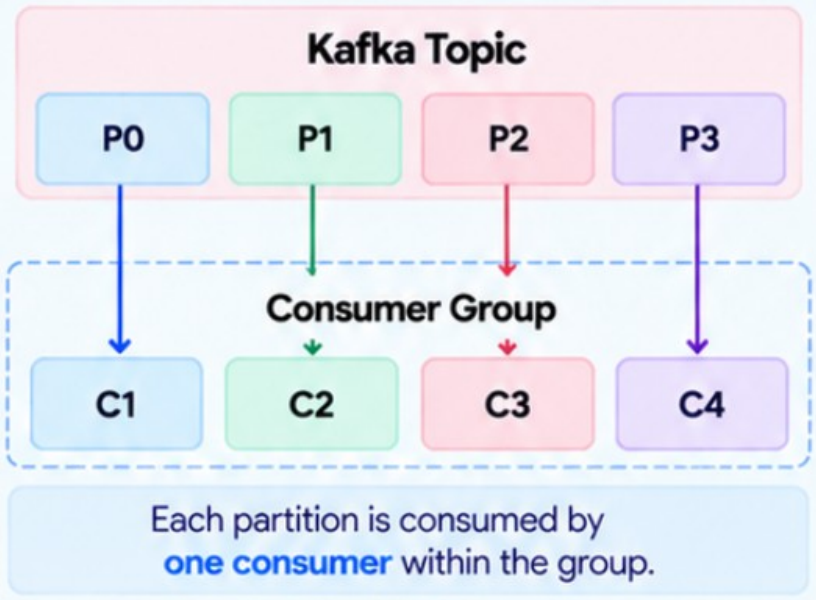
A consumer group allows **multiple consumers** to work together, with each partition consumed by **only one consumer** within the group.

Consumer Group Architecture

Let's **see different scenarios** to understand how partitions are assigned to consumers in a consumer group.

1 Equal Consumers and Partitions

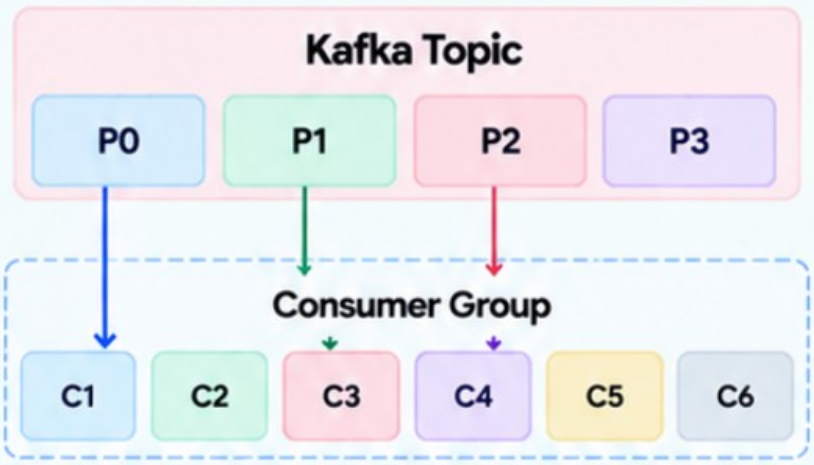
(4 partitions, 4 consumers)



Each partition is consumed by **one consumer** within the group.

2 More Consumers than Partitions

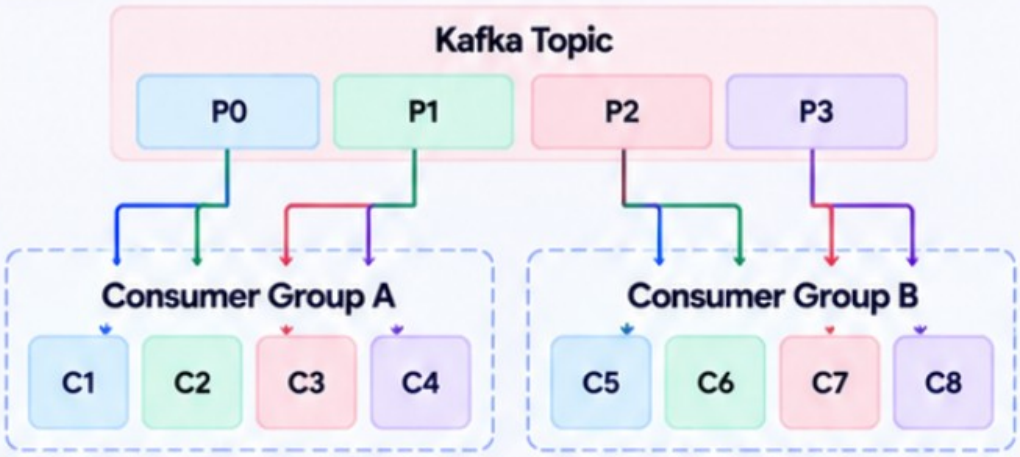
(4 partitions, 6 consumers)



Only 4 consumers will consume (one per partition). Extra consumers (C5, C6) will be idle.

3 Multiple Consumer Groups

(Each group has its own consumers)



Each consumer group gets its own copy of the data.
Different groups can consume the same topic independently.

Key Takeaways

1. Partitions are distributed among **consumers** within a group.
2. One partition is consumed by only **one consumer** in a group.
3. If there are more consumers than partitions, extra consumers remain idle.
4. **Multiple consumer groups** can consume the same topic, and each group gets its own copy of the data.
5. Adding more consumers to a group allows parallel processing and higher throughput.

Complete Kafka Flow From Producer to Consumer

End-to-end flow of a message in Kafka

1 Producer
Creates a message and sends it to a Kafka topic.

2 ProducerRecord
Message wrapped with metadata (topic, key, value, etc.).

3 Serializer
Converts the message into bytes for transmission.

4 Partitioner
Determines the target partition using the message key (or round-robin if key is null).

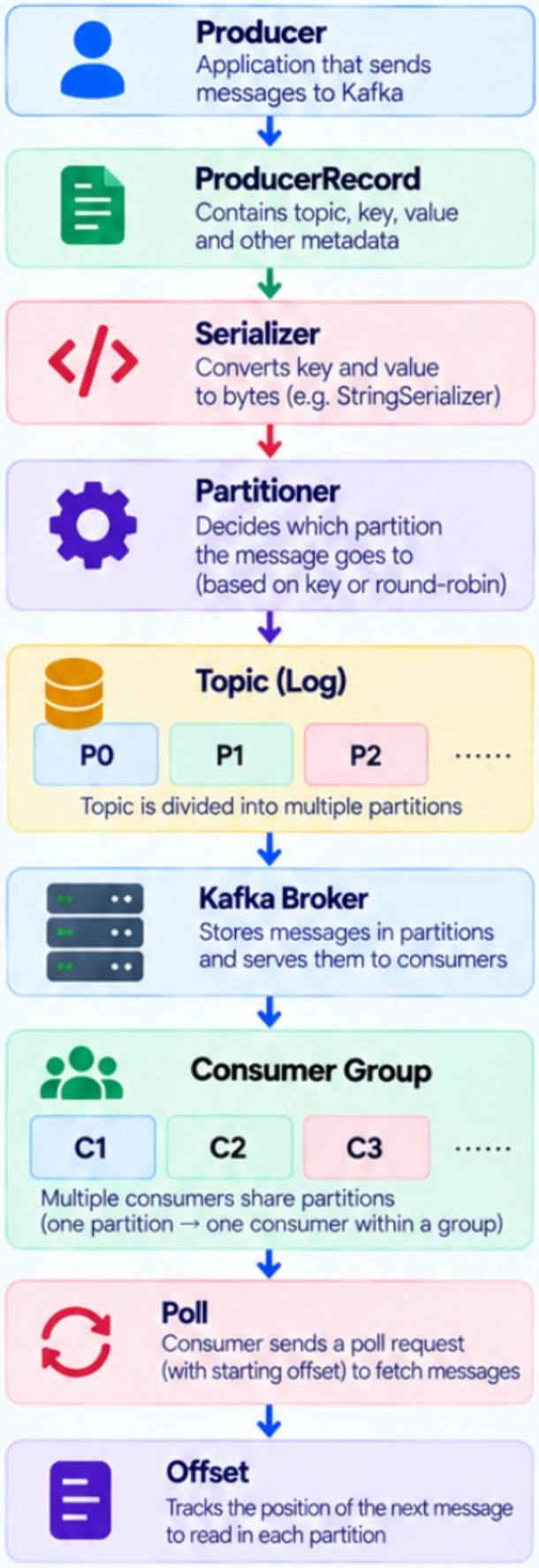
5 Topic & Partitions
Topic is split into multiple partitions. Each partition is an ordered, immutable log.

6 Broker
Kafka brokers store the messages in the partitions and replicate them (as per replication factor).

7 Consumer Group
A group of consumers that work together to read from the topic. One partition is consumed by only one consumer within the group.

8 Poll
Consumers send a poll request (with starting offset) to fetch messages from the broker.

9 Offset
Kafka tracks the offset to know which messages have been read. After processing, the consumer commits the offset.



Key Concepts (Quick Recap)

Message
A unit of data (key + value + metadata) sent to Kafka.

Partition
A topic is divided into multiple partitions. Each partition maintains an ordered log of messages.

Batch
Messages are sent and stored in batches for better performance.

Acks
Producer can configure acks to control durability (0, 1, all).

Offset
A unique, sequential number for each message in a partition.

Consumer Group
A group of consumers that share partitions. One partition is consumed by one consumer within a group.

Final Takeaway

- Producer sends a message to a topic.
- Message is serialized and sent to the correct partition.
- Brokers store the message in the partition.
- Consumers (in a group) poll messages from the broker.
- Offsets keep track of the read position.
- Kafka provides a scalable, fault-tolerant and high-throughput messaging system.
- **From Producer to Consumer — Kafka makes it reliable, scalable and simple!**

Keep Learning | Keep Building