# Kafka

**Apache Kafka is a distributed event streaming platform used to handle high throughput, real-time data streams.**

It enables application to publish, store, process and subscribe to stream of records efficiently

timeout

eg. Chat solution


## Key features of Apache Kafka

- Scalability
- Fault tolerance
- Durability
- High throughput
- Decoupling

# Why do we need Kafka?

Before Kafka, we were using traditional systems

- Databases
- message Queue
- ETL batch processing

real time, high volume

distributed data streams

| | Traditional MQ | Kafka |
|---|---|---|
| Message retention | msgs removed after consumption | Messages can be persisted for a configurable period |
| Scalability | limited due to broker-based approach | Scales easily horizontally with partitions |
| Data Throughput | lower | High |
| Fault tolerance | Lose the message | Replication ensures durability |
| Batch Processing | Process data in chunks at intervals | Process data continously in real time |

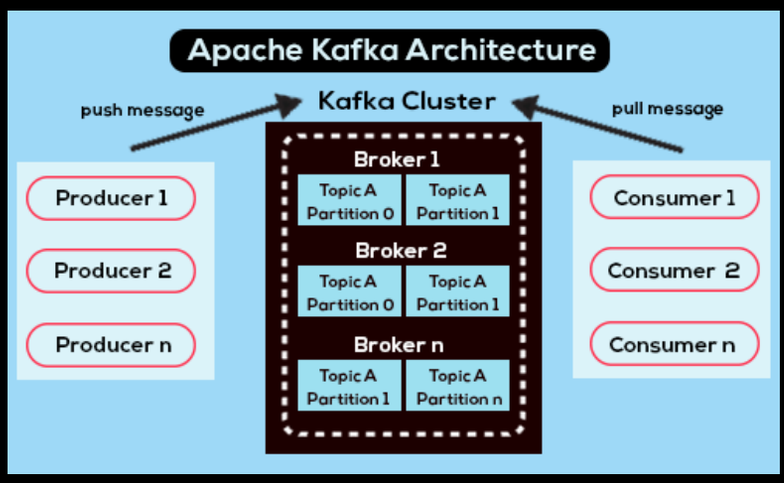
| | Batch Processing | Kafka |
|---|---|---|
| Data Latency | High | Low |
| Processing type | Process accumulated data over interval | Real time as it arrives |
| Example | Newspaper | live news channel |

## Kafka Use Cases

- E-commerce
- Finance
- IoT and Telecom
- Social Media
- log management

# Kafka Architecture

distributed and scalable architecture for real time data




## 1. Broker

Server storing and managing the messages.

## 2. Topic

Category, logical grouping or channel where messages are published.

Community → Grp3
↙ ↘
Grp1 Grp2

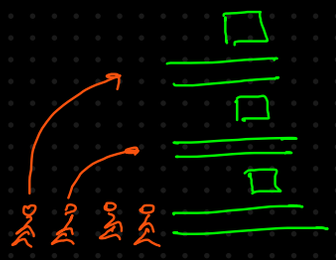
Stock_prices ⇩ trading app

new restaurant ⇩ food delivery apps

## 3. Partition

way to divide the data within topic to improve scalability and parallelism

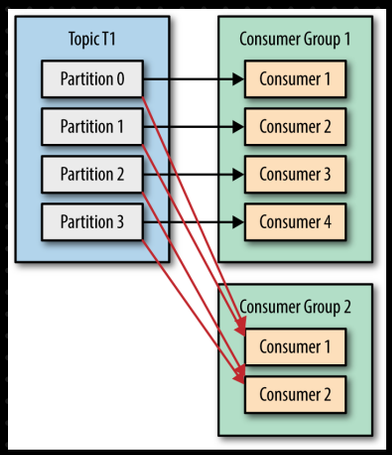
1. Round robin
2. Key-Based Partitioning



## 4. Consumer Groups

4 people throwing ball

6 are catching




Customer support

Agent ⇔ consumer
Calls ⇔ messages

⇒ each consumer from a consumer group reads from a seperate Partition

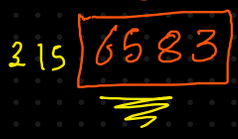
⇒ We cannot have no. of cons. in a grp > no. of partitions

# Offset management

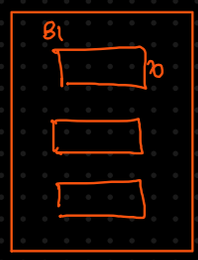
Bookmark

Offset is a unique ID assigned to each message inside the portion.

# Replication and Fault tolerance




Kafka ensures data availability and fault tolerance replication




(cloud backup)

# How to run and use Kafka

1. Local
2. Confluent → Kafka → LinkedIn (2010)
   Open Sourced (2011)
   founders of Kafka created confluent → 2014

   Easy to setup
   Cloud native
   managed service

Amazon (AWS) → MSK
GCP → Pub sub + Kafka on GKE
Azure → Event hubs + Confluent cloud

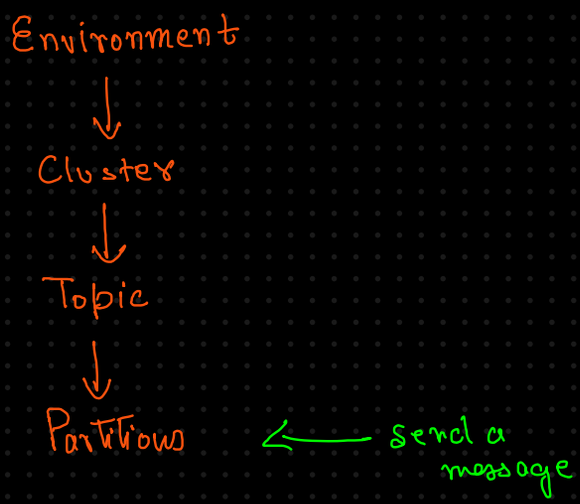
# Kafka on Confluent

Use coupon code
CONFLUENTDEV1
to avoid entering
credit card

Valid as of
Jan 25

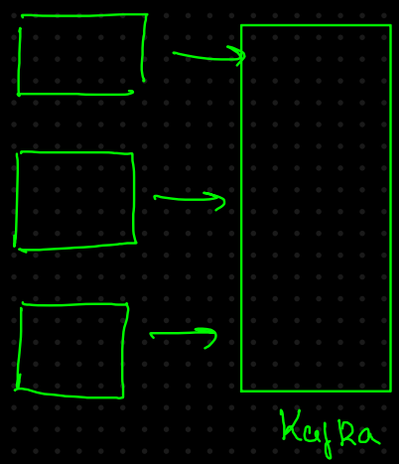
If not valid
try
KAFKA101

else enter card details
to avail $400 credits



# Producing messages to a Kafka

1. Kafka Confluent UI
2. Produce data via Python
   (a) Create a client on Confluent for details
   (b) Copy the secret key and save it
   (c) get the data changed to json




## Callback, poll and Flush

| Function | Purpose | Why is it Needed? | How it Works? |
|---|---|---|---|
| `callback` in `produce()` | **Handles message delivery status** | Kafka produces messages asynchronously, meaning we don't know immediately if the message was successfully delivered or failed. The `callback` function is used to **track message success or failure**. | When a message is sent, Kafka does not confirm its delivery immediately. The callback function is triggered when the message is acknowledged (or failed). |
| `poll()` | **Executes callback functions** | Kafka does not call the `callback` function automatically. Instead, it queues them until `poll()` is called. | When `poll()` is executed, Kafka processes all pending callbacks (e.g., checking if messages were successfully sent). |
| `flush()` | **Ensures all messages are sent before shutting down the producer** | Since Kafka sends messages asynchronously, messages might still be in the queue when the script ends. If we don't flush, unsent messages could be lost. | It **waits** for all in-flight messages to be delivered before shutting down. |

| Step | Function Used | What It Does? | What Happens If We Don't Use It? |
|---|---|---|---|
| **1. Send Message** | `producer.produce(topic, key, value, callback=delivery_report)` | Sends message asynchronously to Kafka | The message will be sent, but we won't know if it was successfully delivered. |
| **2. Trigger Callbacks** | `producer.poll(1)` | Ensures the callback function is executed to check delivery status | The callback function will never execute, and we won't know if the message was delivered or failed. |
| **3. Finalize Sending** | `producer.flush()` | Ensures all messages are sent before shutting down the producer | Unsent messages may be lost when the script exits. |

# Consuming messages from Kafka cluster

Python

Do make sure to install confluent before running

| Command | Purpose |
| --- | --- |
| `confluent login` | Log into Confluent Cloud |
| `confluent environment create <name>` | Create a new environment |
| `confluent environment use <name>` | Set active environment |
| `confluent kafka cluster create <name> --cloud aws --region us-east-1` | Create a Kafka cluster |
| `confluent kafka cluster use <cluster-id>` | Select active Kafka cluster |
| `confluent kafka topic create <topic-name> --partitions <num>` | Create a topic with partitions |
| `confluent kafka topic list` | List all topics |
| `confluent kafka topic describe <topic-name>` | Describe topic details |
| `confluent kafka topic produce <topic-name>` | Send messages to Kafka topic |
| `confluent kafka topic consume <topic-name> --from-beginning` | Read messages from Kafka topic |
| `confluent kafka topic delete <topic-name>` | Delete a Kafka topic |

Set the Key will be required for the first time