🔹 What is Apache Kafka?

Apache Kafka is an open-source, **distributed streaming platform** built for **real-time data pipelines** and **stream processing**. Originally developed by LinkedIn in 2011, Kafka is now used to **process trillions of messages per day**.

🔁 Real-Life Example:

Let’s say you use a food delivery app like Swiggy or Zomato.

Here’s how Kafka could help:

1. 🍽️ **You place an order** — The order info is sent to Kafka.
2. 🧾 Kafka stores it in an “Orders” box (called a *Topic*).
3. 🚴 The delivery system checks that box and gets the order (as a *Consumer*).
4. 🧑‍🍳 At the same time, the kitchen system also reads the same order!
5. 💳 The payment service reads it too and starts processing.

One event (your order) is shared with **many services at the same time** — all thanks to Kafka.

✅ What Makes Kafka Special?

- Handles **a lot of data** very fast 🚀

- Can store data safely even if systems crash 💾
- Sends the same data to **many apps at the same time** 🔄
- Helps systems talk to each other without waiting ⏳

🧩 Summary:

Kafka is like a smart **delivery boy for data**.
It takes messages from one app, stores them safely, and delivers them to one or more other apps **quickly and reliably**.

🔹 Why is Kafka so Popular?

Kafka is the **de facto standard** for stream processing used by over **80% of Fortune 100 companies**. Here's why:

- **High Throughput** – Millions of messages per second
- **Low Latency** – As low as 2ms
- **High Scalability** – Thousands of brokers, petabytes of data
- **Durable Storage** – Distributed commit log ensures reliability
- **High Availability** – Fault-tolerant across zones and regions

---

🔹 Kafka Architecture: How it Works

Kafka is like a **central hub** where data flows in real-time between different systems. It works with the following parts:

- **Producers** – Apps or systems that **send data** into Kafka (e.g., website activity, logs, sensor data).
- **Topics** – Named "folders" inside Kafka that **store the data**.
- **Consumers** – Apps that **read data** from Kafka topics (e.g., dashboards, databases).
- **Brokers** – Kafka servers that **manage and store the data** and help Producers and Consumers talk to Kafka.
- **ZooKeeper / KRaft** – This helps **coordinate** all the Kafka servers and keep the system stable and in sync.

Kafka separates **compute** (processing data) from **storage**, so it can handle real-time data, store it safely, and scale easily.

🔹 **Key Kafka Capabilities**

- **Stream Processing** – Kafka Streams lets you **filter, join, and summarize** data while it's still moving.
- **Pub/Sub Messaging** – Producers publish data, and multiple consumers can read it at the same time.
- **Durability & Reliability** – Data is **stored safely** and can be **replayed** if needed.
- **Real-Time Analytics** – You can connect Kafka to tools like **Druid** or **Elasticsearch** to get live insights.
- **Microservices Communication** – Kafka helps microservices **talk to each other** using fast, event-based messages.

🔹 **Kafka Use Cases**

- **Data Pipelines** – Move data between systems in real time
- **Stream Processing** – Real-time filtering, transformation, enrichment
- **Streaming ETL** – Extract, transform, and load data continuously
- **Event-Driven Microservices** – Reliable inter-service messaging
- **Real-Time Analytics** – Insights and decisions on live data

---

🔹 **Who Uses Kafka?**

Major companies like **Uber, LinkedIn, Netflix, British Gas**, and many more rely on Kafka for mission-critical streaming infrastructure.

✅ **Apache Kafka Installation Steps (on Ubuntu/Linux)**

```
# 1. Update system and install Java
sudo apt update
sudo apt install openjdk-11-jdk -y

# 2. Verify Java installation
java -version

# 3. Download Kafka (example: version 3.7.0, Scala 2.13)
wget https://downloads.apache.org/kafka/3.7.0/kafka_2.13-3.7.0.tgz

# 4. Extract Kafka archive
```

tar -xvzf kafka_2.13-3.7.0.tgz

5. Navigate to Kafka directory
cd kafka_2.13-3.7.0

🚀 Microservices eCommerce Project using Flask, Kafka & MongoDB

🔍 Project Overview

This project demonstrates a beginner-friendly **microservices architecture** that simulates the full lifecycle of an online order — from placing an order to shipping it.

- **Flask** is used to build lightweight, RESTful microservices.
- **Kafka** facilitates **event-driven communication** between services.
- **MongoDB** is used to persist data at each stage of the order.

💡 Why Microservices?

Microservices split large applications into smaller, loosely coupled services that:

- Are easier to maintain and test.
- Can scale independently.

- Improve fault isolation.

🧱 Microservices Overview

Each service has a single responsibility and communicates via **Kafka topics**.

- order_service: Accepts new orders.
- inventory_service: Verifies inventory.
- payment_service: Handles payment.
- shipping_service: Ships the order.
- All order data is stored and updated in MongoDB.

📁 Project Structure

ecommerce-kafka-project/

├── docker-compose.yml # Kafka, Zookeeper, MongoDB, etc.

├── requirements.txt # Dependencies for all services

├── order_service/app.py # Handles new order placements

├── inventory_service/app.py # Verifies inventory after order

├── payment_service/app.py # Processes payment after inventory

├── shipping_service/app.py   # Marks order as shipped

🛠️ Step 1: Docker Setup (docker-compose.yml)

📌 Introduction

This file sets up Kafka, Zookeeper, and MongoDB using Docker containers for a consistent and reproducible environment.

yaml

```
version: '3.8'

services:

  zookeeper:

    image: confluentinc/cp-zookeeper:latest

    environment:

      ZOOKEEPER_CLIENT_PORT: 2181



  kafka:

    image: confluentinc/cp-kafka:latest

    ports:

      - "9092:9092"
```

```yaml
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    depends_on:
      - zookeeper

  mongo:
    image: mongo
    ports:
      - "27017:27017"

volumes:
  mongo_data:
```

🧰 Step 2: Install Python Dependencies

📌 **Introduction**

Install the necessary libraries for Flask, Kafka, and MongoDB to interact within microservices.

📄 requirements.txt:

nginx

flask

kafka-python

pymongo

Install using:

pip install -r requirements.txt

📦 **Step 3: Order Service**

📌 **Introduction**

This service receives incoming orders through an API, saves them in MongoDB, and publishes the event to Kafka.

📄 order_service/app.py:

python

from flask import Flask, request, jsonify

```
from kafka import KafkaProducer

from pymongo import MongoClient

import json


app = Flask(__name__)


producer = KafkaProducer(

    bootstrap_servers='localhost:9092',

    value_serializer=lambda v: json.dumps(v).encode('utf-8')

)


mongo = MongoClient("mongodb://localhost:27017/")

orders = mongo["ecommerce"]["orders"]


@app.route('/order', methods=['POST'])

def create_order():

    data = request.json

    data['status'] = 'Order Placed'
```

```
    orders.insert_one(data)

    producer.send('order-created', data)

    return jsonify({'message': 'Order placed successfully'}), 200



if __name__ == '__main__':

    app.run(port=5000)
```

📦 Step 4: Inventory Service

📌 Introduction

This service listens to the order-created topic. It checks inventory and then sends an event to the inventory-checked topic.

📄 inventory_service/app.py:

python

```
from kafka import KafkaConsumer, KafkaProducer

from pymongo import MongoClient

import json



consumer = KafkaConsumer(
```

```
    'order-created',
    bootstrap_servers='localhost:9092',
    group_id='inventory-group',
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda x: json.dumps(x).encode('utf-8')
)

mongo = MongoClient("mongodb://localhost:27017/")
orders = mongo["ecommerce"]["orders"]

for msg in consumer:
    order = msg.value
    print(f"[Inventory] Order received: {order}")
    order['status'] = 'Inventory Checked'
```

    orders.update_one({'order_id': order['order_id']}, {'$set': {'status': 'Inventory Checked'}})

    producer.send('inventory-checked', order)

📦 Step 5: Payment Service

📌 Introduction

This service listens to inventory-checked, processes the payment, and sends an event to the payment-processed topic.

📄 payment_service/app.py:

```python
from kafka import KafkaConsumer, KafkaProducer

from pymongo import MongoClient

import json



consumer = KafkaConsumer(

    'inventory-checked',

    bootstrap_servers='localhost:9092',

    group_id='payment-group',

    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
```

```
)

producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda x: json.dumps(x).encode('utf-8')
)

mongo = MongoClient("mongodb://localhost:27017/")
orders = mongo["ecommerce"]["orders"]

for msg in consumer:
    order = msg.value
    print(f"[Payment] Processing: {order}")
    order['status'] = 'Payment Done'
    orders.update_one({'order_id': order['order_id']}, {'$set': {'status': 'Payment Done'}})
    producer.send('payment-processed', order)
```

📦 Step 6: Shipping Service

📌 Introduction

The final service listens to payment-processed and marks the order as shipped in MongoDB.

📄 shipping_service/app.py:

python

```python
from kafka import KafkaConsumer

from pymongo import MongoClient

import json



consumer = KafkaConsumer(

    'payment-processed',

    bootstrap_servers='localhost:9092',

    group_id='shipping-group',

    value_deserializer=lambda x: json.loads(x.decode('utf-8'))

)
```

```python
mongo = MongoClient("mongodb://localhost:27017/")

orders = mongo["ecommerce"]["orders"]


for msg in consumer:

    order = msg.value

    print(f"[Shipping] Shipped: {order}")

    orders.update_one({'order_id': order['order_id']}, {'$set': {'status': 'Shipped'}})
```

🏁 Step 7: Running the Project

📌 Introduction

You’ll now bring everything together: start services, run microservices, and test the end-to-end flow.

✅ Start Docker Services:

```
docker-compose up -d
```

✅ Run Python Services (Each in a separate terminal):

```
python order_service/app.py

python inventory_service/app.py
```

python payment_service/app.py

python shipping_service/app.py

🧪 Step 8: Test the Workflow

📌 Introduction

Simulate a real user placing an order and watch it move through all services.

Place an Order via curl:

```
curl -X POST http://localhost:5000/order \
  -H 'Content-Type: application/json' \
  -d '{"order_id": "101", "user": "Alice", "item": "Laptop"}'
```

📊 Step 9: View Orders in MongoDB

📌 Introduction

Verify all updates at each microservice stage from within the MongoDB container.

```
docker exec -it $(docker ps -qf "name=mongo") mongosh
```

Inside the shell:

js

use ecommerce

db.orders.find().pretty()

🎯 **Final Notes**

- This project teaches you how to build a **decoupled, event-driven system**.
- Services **only communicate through Kafka**, ensuring flexibility and scalability.
- MongoDB provides **centralized state** and traceability of the order.

:

> 🔧 **DevOps Project: Real-Time Monitoring & Alerting System with Kafka**
> All setups, configuration files, and introductions are included. You can directly run this with **Docker Compose**, visualize logs with **Kibana**, view metrics in **Grafana**, and get alerts when something goes wrong.

📝 Project Name:

Kafka-Powered Monitoring Pipeline

✅ Goal:

To build a real-time monitoring system that:

- Collects logs and metrics from microservices
- Streams data using **Kafka**
- Indexes logs with **Elasticsearch**
- Visualizes logs (Kibana) and metrics (Grafana)
- Triggers alerts on issues (like CPU > 80%) via Slack/Email

🛠️ Tools Used:

- **Kafka** (message broker)
- **Zookeeper** (Kafka dependency)
- **Filebeat** (log shipper)
- **Logstash** (log processor)
- **Elasticsearch** (log storage & search)
- **Kibana** (log visualization)
- **Prometheus** (metrics collection)
- **Grafana** (dashboard)

- **Alertmanager** (for alerts)
- **Docker + Docker Compose** (container orchestration)

⚙️ How the System Works:

🔸 Step 1: Spin Up the Core Services with Docker Compose

🔍 **Introduction:**
We use Docker Compose to set up all services: Kafka, Zookeeper, Prometheus, Grafana, Elasticsearch, Kibana, Filebeat, and Logstash. This allows a complete monitoring system on your local machine.

yaml

```yaml
# docker-compose.yml
version: '3.7'

services:

  zookeeper:
    image: confluentinc/cp-zookeeper:7.2.1
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.2.1
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
```

```yaml
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:8.6.2
  environment:
    - discovery.type=single-node
    - xpack.security.enabled=false
    - bootstrap.memory_lock=true
    - ES_JAVA_OPTS=-Xms512m -Xmx512m
  ports:
    - "9200:9200"
  ulimits:
    memlock:
      soft: -1
      hard: -1
  volumes:
    - esdata:/usr/share/elasticsearch/data

kibana:
  image: docker.elastic.co/kibana/kibana:8.6.2
  environment:
    - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
  ports:
    - "5601:5601"
  depends_on:
    - elasticsearch

prometheus:
  image: prom/prometheus
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
```

```yaml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"

  logstash:
    image: docker.elastic.co/logstash/logstash:8.6.2
    volumes:
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf
    depends_on:
      - kafka
      - elasticsearch
    ports:
      - "5044:5044"

  filebeat:
    image: docker.elastic.co/beats/filebeat:8.6.2
    volumes:
      - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
      - ./logs:/logs
    depends_on:
      - kafka

volumes:
  esdata:
```

🔸 Step 2: Configure Filebeat to Ship Logs to Kafka

🔍 Introduction:

Filebeat monitors log files (like /logs/app.log) and sends them to Kafka.

yaml

```yaml
# filebeat.yml
filebeat.inputs:
  - type: log
    enabled: true
    paths:
      - /logs/*.log

output.kafka:
  hosts: ["kafka:9092"]
  topic: "logs"
  codec.json:
    pretty: false
```

🔸 **Step 3: Configure Logstash to Process Logs from Kafka and Send to Elasticsearch**

🔍 **Introduction:**
Logstash reads logs from Kafka and forwards them to Elasticsearch for indexing and Kibana visualization.

conf

```
# logstash.conf
input {
  kafka {
    bootstrap_servers => "kafka:9092"
    topics => ["logs"]
    codec => "json"
  }
}

filter {
```

```
  # Add transformations or filters here
}

output {
  elasticsearch {
    hosts => ["http://elasticsearch:9200"]
    index => "app-logs-%{+YYYY.MM.dd}"
  }
  stdout { codec => rubydebug }
}
```

🔸 **Step 4: Create Prometheus Configuration**

🔍 **Introduction:**
Prometheus scrapes metrics from your services (e.g., Node Exporter, apps) and stores time-series data.

yaml

```yaml
# prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

🔸 **Step 5: Launch Grafana and Add Prometheus Data Source**

🔍 **Introduction:**
Grafana connects to Prometheus and visualizes your metrics with dashboards.

- Login at http://localhost:3000
- Default credentials: admin / admin
- Add Prometheus as a data source (http://prometheus:9090)
- Import dashboards or create your own

🔸 **Step 6: Add Alerting Rules**

🔍 **Introduction:**
Alertmanager integrated with Prometheus can send alerts when thresholds are breached.

yaml

```yaml
# alerts/alert.rules
groups:
- name: system-alerts
  rules:
  - alert: HighCPUUsage
    expr: node_cpu_seconds_total > 80
    for: 1m
    labels:
      severity: warning
    annotations:
      summary: "High CPU usage detected"
      description: "CPU usage is over 80% for 1 minute"
```

You can plug this into Prometheus config and use Alertmanager to notify via Slack/Email.

🔸 **Step 7: Create Kafka Topics (Logs, Metrics)**

🔍 **Introduction:**
Kafka needs topics where logs and metrics will be published.

```
docker exec -it kafka kafka-topics --create --topic logs --bootstrap-server
kafka:9092 --replication-factor 1 --partitions 1
```

Repeat this for other topics if needed.

🔸 **Step 8: View Logs in Kibana**

🔍 **Introduction:**
Kibana provides a UI to view, filter, and visualize logs coming from Elasticsearch.

- Access: http://localhost:5601
- Go to “Stack Management” → “Index Patterns” → Add app-logs-*

💡 Apache Kafka Commands

Start Zookeeper Server

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Start Kafka Broker Server

```
bin/kafka-server-start.sh config/server.properties
```

Create a Kafka Topic

bin/kafka-topics.sh --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

List Kafka Topics

bin/kafka-topics.sh --list --bootstrap-server localhost:9092

Start Kafka Producer (send messages)

bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092

Start Kafka Consumer (read messages)

bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server localhost:9092

RabbitMQ and **Kafka** are **message brokers**, but they are used for **different use-cases**.

🆚 RabbitMQ vs Kafka: Key Differences

Feature	RabbitMQ	Apache Kafka
Type	Message Queue (traditional broker)	Distributed Event Streaming Platform
Message Pattern	**Push-based** (broker pushes to consumers)	**Pull-based** (consumers pull from Kafka)

Ordering	Not guaranteed	Guaranteed per partition
Speed	Slower for high throughput	Designed for high throughput (millions/sec)
Storage	Transient (short-lived messages)	Durable (messages stored for days/weeks)
Use-case	Job queues, async processing	Real-time analytics, event streaming
Replay	Not possible after ACK	Yes, you can replay old messages
Built-in Features	Routing, priority queues, TTL, retries	Partitioning, replication, stream replay
Ease of Use	Easier for beginners	Needs deeper understanding (like partitions, offsets)
Dependencies	No Zookeeper	Needs Zookeeper (or KRaft mode)

🔥 **Summary: Which is best?**

For	Best Choice
Simple task queues	✅ RabbitMQ
Complex routing	✅ RabbitMQ
Real-time streaming	✅ Kafka
High data volume	✅ Kafka
Replay old messages	✅ Kafka

Beginners	✅ RabbitMQ