Apache Kafka: Powering Real-Time Data Pipelines and Event-Driven Architectures

Team 12- Aditya Kocherlakota & Aryaman Jalali

Introduction-

This document presents the implementation and results of our term project "Apache Kafka: Powering Real-Time Data Pipelines and Event-Driven Architectures." It demonstrates Kafka's ability to handle continuous streaming data and asynchronous microservice communication using Python, Docker, and Streamlit.

Environment Setup-

System Environment:

- macOS 14 (Apple Silicon)
- VS Code with integrated terminal
- Docker Desktop v4.32+
- Kafka Version**:** Confluent Kafka 7.6.1 (Dockerized local setup)
- Python 3.12
- Streamlit for dashboard visualization
- kafka-python, pandas, and matplotlib for data streaming and analysis

Docker Services:

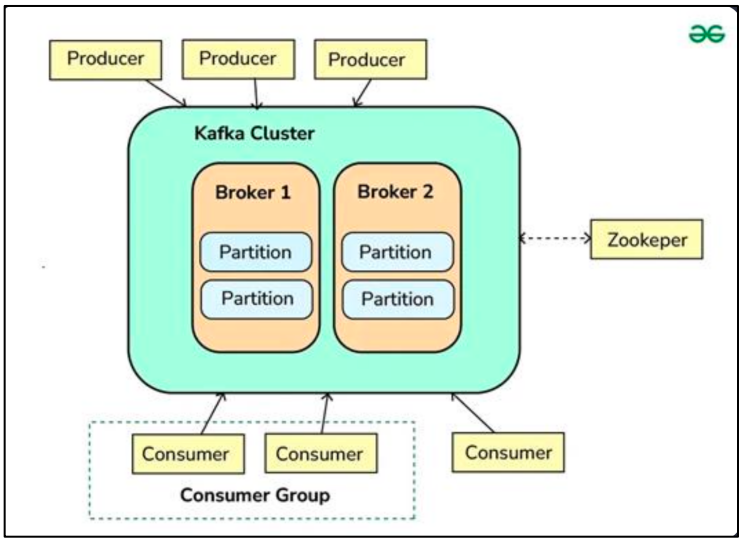
- Zookeeper: Manages Kafka broker coordination
- Kafka Broker: Handles message publishing and subscribing
- Configuration: Based on Confluent Kafka 7.6.1 image

```yaml
services:
  ▷Run Service
  zookeeper:
    image: confluentinc/cp-zookeeper:7.6.1
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  ▷Run Service
  kafka:
    image: confluentinc/cp-kafka:7.6.1
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Kafka Architecture Overview-

- Kafka's architecture follows a distributed publish–subscribe model that enables high-throughput message streaming.
- Producers publish messages to topics, which are divided into partitions for scalability.
- Brokers store these messages durably and replicate them across nodes for fault tolerance.
- Consumers subscribe to topics and pull messages at their own pace.
- Zookeeper coordinates brokers and maintains cluster metadata.

This foundation allows Kafka to process millions of events per second with durability and low latency.



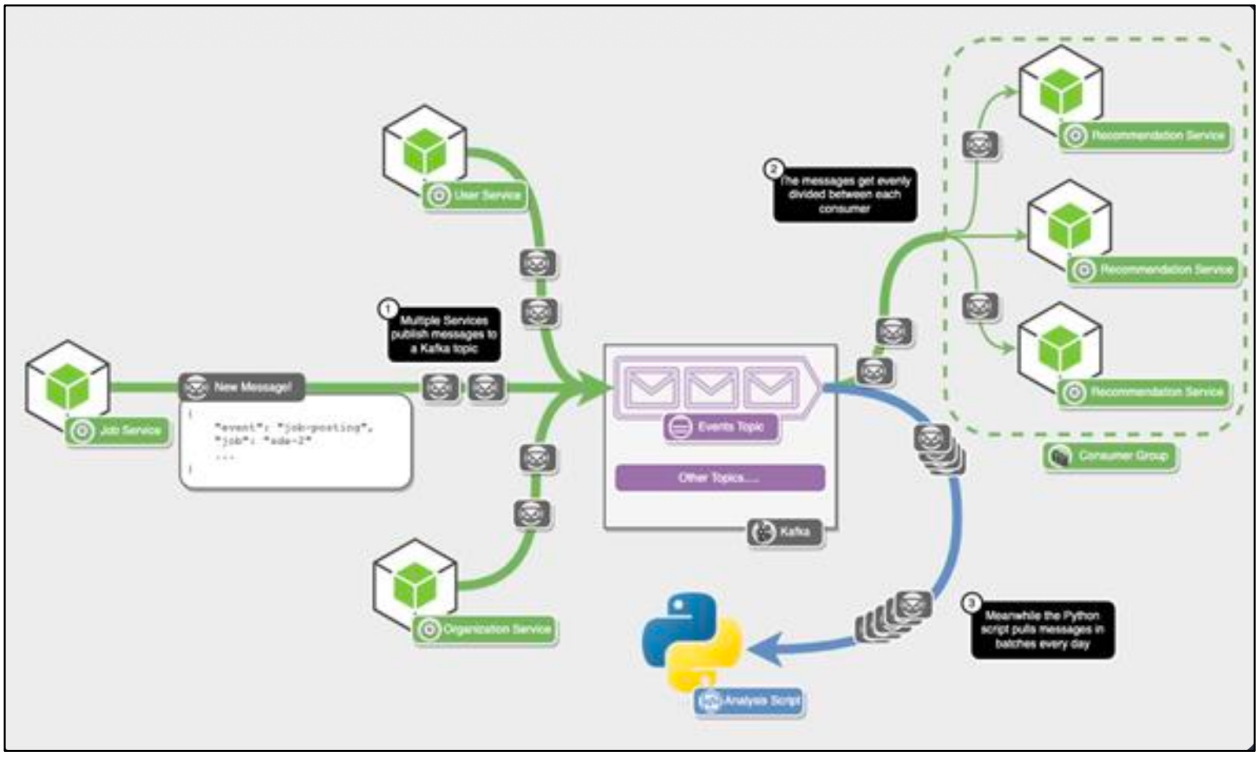
Kafka in Event-Driven Microservices-

Kafka acts as a message backbone between microservices, enabling asynchronous event flow.

In this model:

- Upstream services (e.g., User, Job, or Organization) publish domain events.
- Downstream consumers (e.g., Analytics or Recommendation engines) react to those events independently.
- Kafka decouples these services so that each can scale, restart, or fail without interrupting the others.

This event-driven approach increases resilience and ensures real-time updates across distributed systems.



How to Run the Code-

The following section outlines the steps required to execute the Kafka-based demos locally using Docker and Python:

Real-Time Data Pipeline Demo:

Simulates an IoT system streaming continuous weather sensor data.

Steps:

1. Start Kafka and Zookeeper

   docker compose up -d

2. Run the requirements.txt file

   pip install -r requirements.txt

3. Run the producer to generate live sensor data

   python producer.py

4. Run the consumer to process messages in real time

   python consumer.py

5. Launch the dashboard for visualization

   streamlit run dashboard.py

Result:

A live dashboard showing continuously updating temperature and humidity readings — with alerts when temperatures exceed 30°C.

Event-Driven Microservices Demo:

Implements a lightweight Amazon-like order → payment → notification flow.

Steps:

1. Run all services (each in a separate terminal):

   python order_service.py
   python payment_service.py
   python notification_service.py

2. Observe event logs:

   order_created → payment_processed → notification_sent

Result:

Demonstrates asynchronous, decoupled microservice communication via Kafka topics.

Results of Running the Code-

Kafka Microservices Demo (Order–Payment–Notification Flow):

The following screenshots show the **end-to-end event flow** in the Kafka-based microservices demo.
Each service runs independently but communicates asynchronously via Kafka topics.

- **Order Service:**
  Generates continuous order events (e.g., order_created) with unique order IDs, user IDs, and order amounts.
  Kafka produces these messages to the **orders topic**.
- **Payment Service:**
  Consumes messages from the orders topic, simulates payment validation, and publishes new messages to the payment's topic.
  Payment status is randomly marked as *"successful"* or *"failed"* to simulate real-world outcomes.
- **Notification Service:**
  Subscribes to the payments topic and sends confirmation events (e.g., email notification) back to users depending on payment results.

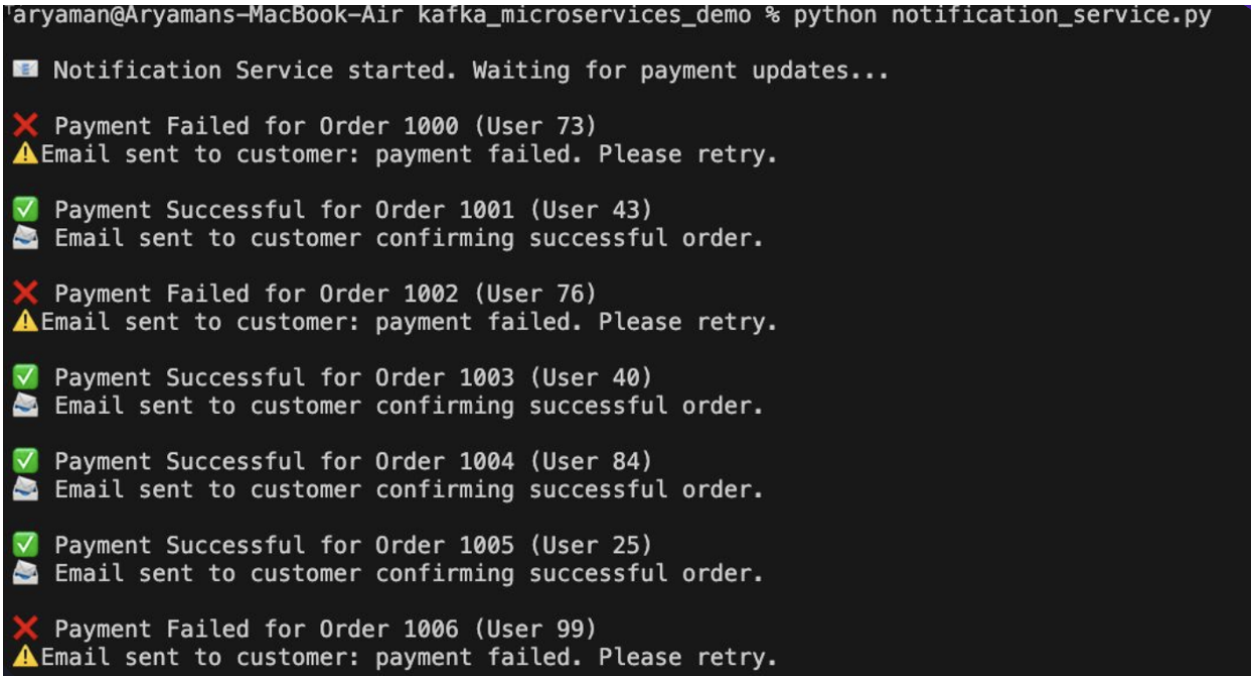
Key Result:
The screenshot confirms successful event propagation through all three services:

1. Orders are created and published to Kafka.
2. Payments are processed and published to the next topic.
3. Notifications are triggered based on the payment results.

This validates Kafka's reliability as a **message broker** enabling **decoupled microservices communication**.

```
aryaman@Aryamans-MacBook-Air kafka_microservices_demo % docker compose up -d
[+] Running 2/2
 ✔ Container kafka_microservices_demo-zookeeper-1  Running
 ✔ Container kafka_microservices_demo-kafka-1      Started
aryaman@Aryamans-MacBook-Air kafka_microservices_demo % docker ps
CONTAINER ID   IMAGE                             COMMAND                  CREATED      STATUS
847a8141337a   confluentinc/cp-kafka:7.6.1       "/etc/confluent/dock…"   8 days ago   Up 6 seconds
_demo-kafka-1
315ec646bafc   confluentinc/cp-zookeeper:7.6.1   "/etc/confluent/dock…"   8 days ago   Up 3 minutes
_demo-zookeeper-1
aryaman@Aryamans-MacBook-Air kafka_microservices_demo % python order_service.py
🛒 Order Service started. Generating new orders...
📦 Order Created: {'order_id': 1000, 'user_id': 73, 'amount': 453.91, 'status': 'created'}
📦 Order Created: {'order_id': 1001, 'user_id': 43, 'amount': 41.58, 'status': 'created'}
📦 Order Created: {'order_id': 1002, 'user_id': 76, 'amount': 412.97, 'status': 'created'}
📦 Order Created: {'order_id': 1003, 'user_id': 40, 'amount': 219.16, 'status': 'created'}
📦 Order Created: {'order_id': 1004, 'user_id': 84, 'amount': 111.55, 'status': 'created'}
📦 Order Created: {'order_id': 1005, 'user_id': 25, 'amount': 427.34, 'status': 'created'}
📦 Order Created: {'order_id': 1006, 'user_id': 99, 'amount': 390.53, 'status': 'created'}
📦 Order Created: {'order_id': 1007, 'user_id': 87, 'amount': 309.26, 'status': 'created'}
📦 Order Created: {'order_id': 1008, 'user_id': 1, 'amount': 159.22, 'status': 'created'}
📦 Order Created: {'order_id': 1009, 'user_id': 61, 'amount': 17.69, 'status': 'created'}
📦 Order Created: {'order_id': 1010, 'user_id': 95, 'amount': 50.73, 'status': 'created'}
📦 Order Created: {'order_id': 1011, 'user_id': 86, 'amount': 295.61, 'status': 'created'}
📦 Order Created: {'order_id': 1012, 'user_id': 6, 'amount': 398.75, 'status': 'created'}
📦 Order Created: {'order_id': 1013, 'user_id': 81, 'amount': 115.58, 'status': 'created'}
📦 Order Created: {'order_id': 1014, 'user_id': 82, 'amount': 322.48, 'status': 'created'}
📦 Order Created: {'order_id': 1015, 'user_id': 81, 'amount': 22.5, 'status': 'created'}
📦 Order Created: {'order_id': 1016, 'user_id': 63, 'amount': 459.19, 'status': 'created'}
```

```
aryaman@Aryamans-MacBook-Air kafka_microservices_demo % python payment_service.py

💳 Payment Service started. Listening for new orders...
📥 Received Order: {'order_id': 1000, 'user_id': 73, 'amount': 453.91, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1000, 'user_id': 73, 'amount': 453.91, 'status': 'payment_failed'}

📥 Received Order: {'order_id': 1001, 'user_id': 43, 'amount': 41.58, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1001, 'user_id': 43, 'amount': 41.58, 'status': 'payment_successful'}

📥 Received Order: {'order_id': 1002, 'user_id': 76, 'amount': 412.97, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1002, 'user_id': 76, 'amount': 412.97, 'status': 'payment_failed'}

📥 Received Order: {'order_id': 1003, 'user_id': 40, 'amount': 219.16, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1003, 'user_id': 40, 'amount': 219.16, 'status': 'payment_successful'}

📥 Received Order: {'order_id': 1004, 'user_id': 84, 'amount': 111.55, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1004, 'user_id': 84, 'amount': 111.55, 'status': 'payment_successful'}

📥 Received Order: {'order_id': 1005, 'user_id': 25, 'amount': 427.34, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1005, 'user_id': 25, 'amount': 427.34, 'status': 'payment_successful'}

📥 Received Order: {'order_id': 1006, 'user_id': 99, 'amount': 390.53, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1006, 'user_id': 99, 'amount': 390.53, 'status': 'payment_failed'}

📥 Received Order: {'order_id': 1007, 'user_id': 87, 'amount': 309.26, 'status': 'created'}
✅ Payment Event Sent: {'order_id': 1007, 'user_id': 87, 'amount': 309.26, 'status': 'payment_failed'}
```



Real-Time Data Streaming (Weather Sensor Pipeline):

The outputs below demonstrate Kafka's ability to handle high-throughput real-time streaming data.

Here, a Kafka Producer simulates IoT weather sensor readings — continuously sending temperature and humidity data every few seconds.

Each record follows the format:

{"sensor_id": 123, "temperature": 29.5, "humidity": 56.8, "timestamp": "2025-10-30T19:02:43"}

The Kafka Consumer subscribes to this data stream and processes it in real time, which is later visualized in a dashboard.

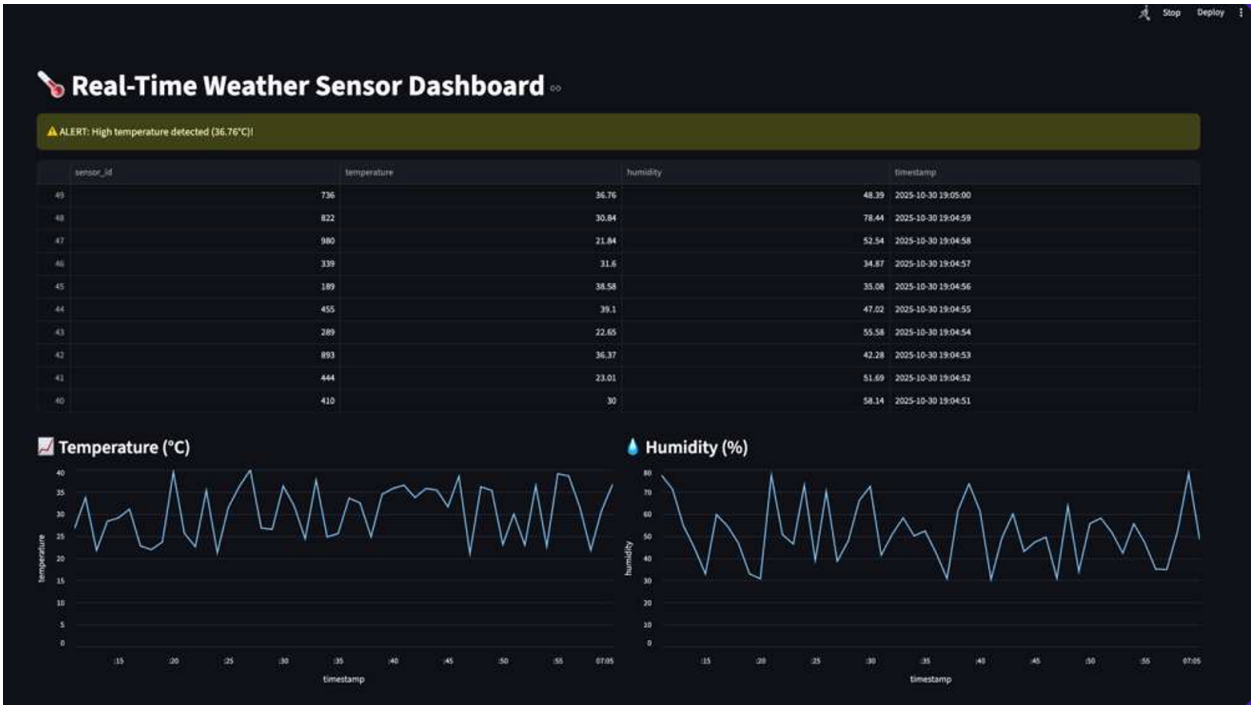
Key Result:

Data streamed seamlessly with sub-second latency.

Producer–Consumer communication verified via localhost:9092.

Real-time ingestion pipeline functioning as designed.

```
🚀 Kafka Producer started... streaming temperature & humidity data
Produced: {'sensor_id': 979, 'temperature': 34.84, 'humidity': 79.47, 'timestamp': '2025-10-30 19:02:33'}
Produced: {'sensor_id': 387, 'temperature': 31.33, 'humidity': 72.8, 'timestamp': '2025-10-30 19:02:34'}
Produced: {'sensor_id': 198, 'temperature': 35.67, 'humidity': 66.24, 'timestamp': '2025-10-30 19:02:35'}
Produced: {'sensor_id': 616, 'temperature': 32.38, 'humidity': 60.81, 'timestamp': '2025-10-30 19:02:36'}
Produced: {'sensor_id': 228, 'temperature': 28.62, 'humidity': 62.56, 'timestamp': '2025-10-30 19:02:37'}
Produced: {'sensor_id': 243, 'temperature': 23.51, 'humidity': 41.97, 'timestamp': '2025-10-30 19:02:38'}
Produced: {'sensor_id': 228, 'temperature': 21.27, 'humidity': 75.02, 'timestamp': '2025-10-30 19:02:39'}
Produced: {'sensor_id': 436, 'temperature': 23.1, 'humidity': 50.03, 'timestamp': '2025-10-30 19:02:40'}
Produced: {'sensor_id': 492, 'temperature': 37.15, 'humidity': 36.14, 'timestamp': '2025-10-30 19:02:41'}
Produced: {'sensor_id': 224, 'temperature': 32.47, 'humidity': 49.07, 'timestamp': '2025-10-30 19:02:42'}
Produced: {'sensor_id': 540, 'temperature': 38.71, 'humidity': 31.43, 'timestamp': '2025-10-30 19:02:43'}
Produced: {'sensor_id': 635, 'temperature': 25.99, 'humidity': 39.88, 'timestamp': '2025-10-30 19:02:44'}
Produced: {'sensor_id': 592, 'temperature': 21.73, 'humidity': 47.49, 'timestamp': '2025-10-30 19:02:45'}
Produced: {'sensor_id': 847, 'temperature': 23.04, 'humidity': 66.43, 'timestamp': '2025-10-30 19:02:46'}
Produced: {'sensor_id': 597, 'temperature': 30.83, 'humidity': 65.59, 'timestamp': '2025-10-30 19:02:47'}
Produced: {'sensor_id': 593, 'temperature': 34.57, 'humidity': 54.17, 'timestamp': '2025-10-30 19:02:48'}
Produced: {'sensor_id': 785, 'temperature': 39.26, 'humidity': 51.04, 'timestamp': '2025-10-30 19:02:49'}
Produced: {'sensor_id': 362, 'temperature': 39.89, 'humidity': 31.26, 'timestamp': '2025-10-30 19:02:50'}
Produced: {'sensor_id': 541, 'temperature': 22.99, 'humidity': 74.48, 'timestamp': '2025-10-30 19:02:51'}
Produced: {'sensor_id': 495, 'temperature': 39.63, 'humidity': 36.27, 'timestamp': '2025-10-30 19:02:52'}
Produced: {'sensor_id': 469, 'temperature': 37.1, 'humidity': 71.31, 'timestamp': '2025-10-30 19:02:53'}
Produced: {'sensor_id': 566, 'temperature': 39.72, 'humidity': 36.07, 'timestamp': '2025-10-30 19:02:54'}
Produced: {'sensor_id': 439, 'temperature': 27.7, 'humidity': 61.46, 'timestamp': '2025-10-30 19:02:55'}
Produced: {'sensor_id': 809, 'temperature': 38.51, 'humidity': 77.44, 'timestamp': '2025-10-30 19:02:56'}
Produced: {'sensor_id': 400, 'temperature': 35.45, 'humidity': 57.46, 'timestamp': '2025-10-30 19:02:57'}
Produced: {'sensor_id': 678, 'temperature': 31.51, 'humidity': 60.09, 'timestamp': '2025-10-30 19:02:58'}
```



The above screenshot captures the Streamlit dashboard, which visualizes live Kafka data streams for temperature and humidity.

Dashboard Features:

- Real-time data table updating continuously with new sensor readings.
- Line charts for **Temperature (°C)** and **Humidity (%)** plotted over time.
- Alert system highlighting high-temperature conditions (above 30°C).

Key Result:

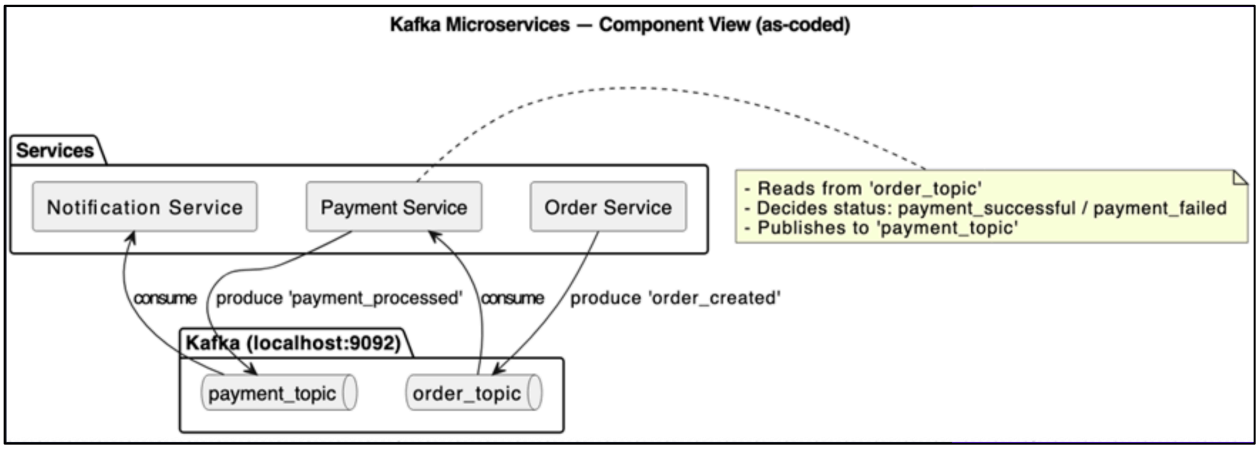
- Demonstrated complete data flow from **Kafka Producer → Kafka Consumer → Streamlit Visualization.**
- Real-time alerts validated when temperature crossed thresholds.
- Proved Kafka's integration capability with external visualization tools.

Kafka Microservices Demo Architecture:

The paper's implementation extends Kafka's architecture into a three-service workflow:

- Order Service generates order_created events and publishes them to the order_topic.
- Payment Service consumes these messages, processes payments, and emits payment_processed results to the payment_topic.
- Notification Service listens to payment_topic and triggers confirmation events for successful transactions.

This design showcases Kafka's reliability and scalability, enabling each service to function independently while remaining fully synchronized through topic-based event streams.



Explanation of Dataset and Results:

The dataset used in this demo was synthetic real-time data generated by Python producers. For the microservice demo, orders and payments were randomly simulated to reflect a typical e-commerce flow. For the sensor demo, temperature and humidity readings were generated every few seconds to simulate IoT telemetry.

Kafka successfully streamed, processed, and visualized all events in real time, confirming its high-throughput, low-latency, and fault-tolerant architecture.

Conclusion:

Through the implementation of two Kafka demos — a real-time IoT data pipeline and an event-driven microservices system — we validated Kafka's strengths in scalability, fault tolerance, and real-time processing. The project successfully bridged theoretical architecture with hands-on execution, aligning fully with the objectives of the MET CS 777 term paper.