**Real-Time Streaming Data Pipeline with Kafka, Docker, and Python**

**Introduction**

In this project, I demonstrate the creation and deployment of a real-time streaming data pipeline using Apache Kafka, Docker, and Python. The pipeline is designed to ingest, process, and store user activity data in real-time, ensuring continuous and efficient data processing.

**Objectives**

1. Set up a local development environment using Docker, including Kafka and any other necessary components.
2. Design and implement a Kafka consumer to consume data from a Kafka topic, perform basic processing, and identify interesting insights.
3. Configure another Kafka topic to store the processed data.
4. Ensure the pipeline can handle streaming data continuously and efficiently, handling error messages and missing fields.

**Project Setup:**

**Prerequisites**

Before starting, ensure you have the following installed:

* Docker
* Docker Compose
* Python

**Steps:**

1. Setting Up Docker

I used Docker to create isolated environments for Kafka and Zookeeper instances. Docker Compose simplifies the process by allowing me to define and manage multi-container Docker applications with a single file.

* **Docker Installation**: Download and install Docker from Docker's official website.

2. Create a project Directory:

* Create a new directory for your project. Open terminal or command prompt and run the following commands that are given below,

|  |
| --- |
| mkdir kafka-pipeline  cd kafka-pipeline |

3. create a Docker Compose File:

* Go to the project directory that you have created earlier and create a new file with name **docker-compose.yml.** And add the following configuration as shown in the below,

|  |
| --- |
| `version: '2'  services:  zookeeper:  image: confluentinc/cp-zookeeper:latest  environment:  ZOOKEEPER\_CLIENT\_PORT: 2181  ZOOKEEPER\_TICK\_TIME: 2000  ports:  - 22181:2181  networks:  - kafka-network  kafka:  image: confluentinc/cp-kafka:latest  depends\_on:  - zookeeper  ports:  - 9092:9092  - 29092:29092  networks:  - kafka-network  environment:  KAFKA\_BROKER\_ID: 0  KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181  KAFKA\_ADVERTISED\_LISTENERS: LISTENER\_INTERNAL://kafka:9092,LISTENER\_EXTERNAL://localhost:29092  KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: LISTENER\_INTERNAL:PLAINTEXT,LISTENER\_EXTERNAL:PLAINTEXT  KAFKA\_INTER\_BROKER\_LISTENER\_NAME: LISTENER\_INTERNAL  KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1  KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1  my-python-producer:  image: mpradeep954/fetch-de-data-gen  depends\_on:  - kafka  restart: on-failure:10  ports:  - 9093:9093  environment:  BOOTSTRAP\_SERVERS: kafka:9092  KAFKA\_TOPIC: user-login  networks:  - kafka-network  networks:  kafka-network:  driver: bridge |

4. Now start the Docker compose by running the following command in the terminal or command prompt

|  |
| --- |
| docker-compose up -d |

5. Next we have to verify the setup by checking if the containers are running or not. To do that we have to use the below command.

|  |
| --- |
| docker-compose ps |

6. After executing the above command in the project terminal we have to see the following entries “zookeeper”, “Kafka”, and “my-python-producer”

7. Next step is to create a Kafka Topic, we have access to the Kafka container before we create a Kafka topic by using the below command we can achieve that.

|  |
| --- |
| docker exec -it kafka-pipeline\_kafka\_1 /bin/bash |

8. Now we are in the Kafka container we have to create a new topic with name “processed-user-login”. We will use the below command.

|  |
| --- |
| kafka-topics --create --topic processed-user-login --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1 |

9. Now, we have created a new topic to store the processed data and next we have list all the containers that existed.

|  |
| --- |
| Kafka-topics --list --bootstrap-server localhost:9092 |

10.Until now we have setup the docker and Kafka in the project environment, now we have to build Python scripts for Produce and Consumer. Here, producer will generate random data with the parameters that are given by us and consumer has to receive the same data and process that data and has to be stored in the new container that we have created earlier.

11. First create a directory for python scripts,

|  |
| --- |
| mkdir kafka\_pipline  cd kafka\_pipline |

12. Let’s create a “producer.py” script to produce messages to the “user-login” topic

|  |
| --- |
| #producer script  from kafka import KafkaProducer  import json  import time  import random  import uuid  # Kafka Producer Configuration  producer = KafkaProducer(  bootstrap\_servers='localhost:29092',  value\_serializer=lambda x: json.dumps(x).encode('utf-8')  )  # Function to send data  def send\_data(topic, data):  producer.send(topic, value=data)  producer.flush() # Ensure all messages are sent  print(f"Sent data: {data}")  if \_\_name\_\_ == '\_\_main\_\_':  while True:  # sample data  data = {  'user\_id': str(uuid.uuid4()),  'app\_version': f'{random.randint(1, 3)}.{random.randint(0, 9)}.{random.randint(0, 9)}',  'device\_type': random.choice(['android', 'ios']),  'ip': f'{random.randint(0, 255)}.{random.randint(0, 255)}.{random.randint(0, 255)}.{random.randint(0, 255)}',  'locale': random.choice(['RU', 'EN', 'ES', 'FR', 'DE']),  'device\_id': f'{random.randint(100, 999)}-{random.randint(10, 99)}-{random.randint(1000, 9999)}',  'timestamp': str(int(time.time())),  'session\_duration': random.randint(60, 3600), # in seconds  'location': random.choice(['Moscow', 'New York', 'London', 'Berlin', 'Tokyo']),  'network\_type': random.choice(['WiFi', '4G', '5G']),  'app\_activity': random.choice(['login', 'logout', 'purchase', 'browse', 'search']),  'battery\_level': random.randint(1, 100), # in percentage  'screen\_resolution': random.choice(['1080x1920', '720x1280', '1440x2560']),  'os\_version': random.choice(['Android 9', 'Android 10', 'iOS 13', 'iOS 14']),  'carrier': random.choice(['AT&T', 'Verizon', 'T-Mobile', 'Sprint']),  'app\_usage\_type': random.choice(['foreground', 'background'])  }    # Send data to 'user-login' topic  send\_data('user-login', data)    # Sleep for a while before sending the next message  time.sleep(5) # Adjust the sleep time as needed |

13. Now we have successfully created producer file now we have to create consumer file. This consumer file script is to consume messages from the “user-login” topic, process them and produce them to the “processed-user-login” topic.

|  |
| --- |
| #consumer script  from kafka import KafkaConsumer, KafkaProducer  import json  import logging  import time  # Set up logging  logging.basicConfig(level=logging.INFO)  logger = logging.getLogger(\_\_name\_\_)  # Kafka Consumer Configuration  consumer = KafkaConsumer(  'user-login',  bootstrap\_servers='localhost:29092',  auto\_offset\_reset='earliest',  enable\_auto\_commit=True,  group\_id='my-group',  value\_deserializer=lambda x: json.loads(x.decode('utf-8'))  )  # Kafka Producer Configuration  producer = KafkaProducer(  bootstrap\_servers='localhost:29092',  value\_serializer=lambda x: json.dumps(x).encode('utf-8')  )  # Processing Messages  def process\_message(data):  try:  # Perform some basic processing  processed\_data = {  'user\_id': data['user\_id'],  'app\_version': data['app\_version'],  'device\_type': data['device\_type'],  'ip': data['ip'],  'locale': data['locale'],  'device\_id': data['device\_id'],  'timestamp': data['timestamp'],  'session\_duration': data['session\_duration'],  'location': data['location'],  'network\_type': data['network\_type'],  'app\_activity': data['app\_activity'],  'battery\_level': data['battery\_level'],  'screen\_resolution': data['screen\_resolution'],  'os\_version': data['os\_version'],  'carrier': data['carrier'],  'app\_usage\_type': data['app\_usage\_type'],  'processed\_time': int(time.time())  }  # Send processed data to another topic  producer.send('processed-user-login', value=processed\_data)  logger.info(f"Processed data: {processed\_data}")  except KeyError as e:  logger.error(f"Missing field in data: {e}")  except Exception as e:  logger.error(f"Error processing message: {e}")  def consume\_messages():  try:  for message in consumer:  data = message.value  process\_message(data)  except Exception as e:  logger.error(f"Error in consumer loop: {e}")  if \_\_name\_\_ == '\_\_main\_\_':  logger.info("Starting Kafka consumer...")  consume\_messages() |

14. Now we are having both the producer script and consumer script, next we have to run the producer script and consumer script, the below are following commands to run the scripts in the terminal.

|  |
| --- |
| producer.py  consumer.py |

15. After running the scripts, we have to verify the processed data in the new Kafka topic, so first we have see the running containers by the following command

|  |
| --- |
| docker ps |

16. Next from the container choose the container name that has new data and to access the Kafka container use the following command as we have used initially

|  |
| --- |
| docker exec -it kafka-pipeline-kafka-1 /bin/bash |

16. Now, finally we have to view the consumer data that has been processed and stored into the new container.

|  |
| --- |
| kafka-console-consumer --bootstrap-server localhost:9092 --topic processed-user-login --from-beginning |

17. By following these steps, you will set up a complete Kafka-based project with Docker, including Kafka, Zookeeper, a producer, and a consumer. You will also be able to produce and consume messages to verify the setup

**Data Ingestion**

The Kafka producer generates synthetic user activity data and sends it to the Kafka topic **user-login.**

**Data Transformation and Processing**

The Kafka consumer.py reads data from the user-login topic, processes it by transforming, aggregating, and filtering the data, and sends the processed data to the **processed-user-login** topic.

**Data Storage**

Processed data is stored in the **processed-user-login** topic, making it available for further analysis or consumption by other services.

**Handling Errors and Missing Data**

**Error Handling Strategies**

1. **Logging**: Keeps track of operational messages and errors to facilitate monitoring and debugging.
2. **Exception Handling**: Catches and handles exceptions that occur during message processing, ensuring the pipeline remains operational even when individual messages cause errors.

**Handling Missing Fields**

I have developed a logic to handle missing fields in real-time data streams using a dimension table. This dimension table tracks columns and their last update times, ensuring that any missing fields in the incoming data are filled with a default value (e.g., "Null"). This helps maintain data completeness and consistency over time.

**Key Components**

1. **Dimension Table:**
   * A dictionary (**dimension\_table**) where each key is a column name, and the value is the timestamp of the last update. This table tracks which columns have been seen in the incoming data and when they were last updated.
2. **Dimension Update Threshold**:
   * A time threshold (**dimension\_update\_threshold**) defined in seconds. This determines how long a column should be retained in the dimension table if it hasn't been updated.
   * This value can be updated based on the requirements.

**Functions**

1**. update\_dimension\_table**(data)

**Purpose**: This function updates the dimension table with the current columns in the incoming data and removes columns that haven't been updated within the threshold time.

**Steps:**

1. Get Current Time: The function retrieves the current timestamp using int(time.time()).
2. Update Table: For each column (key) in the incoming data, it updates the dimension table with the current timestamp.
3. Remove Outdated Columns: The function identifies columns that haven't been updated within the specified threshold (e.g., 60 seconds) and removes them from the dimension table.

**Example Workflow:**

* If **dimension\_update\_threshold** is set to 60 seconds, any column not updated within the last 60 seconds will be removed from the table.

**2. handle\_missing\_fields(data)**

Purpose: This function ensures that all expected columns are present in the incoming data by checking against the dimension table and adding missing columns with a default value.

**Steps:**

1. Get Current Time: The function retrieves the current timestamp using int(time.time()).
2. Check Missing Columns: For each column name (key) in the dimension table, it checks if the column exists in the incoming data.
3. Add Missing Columns: If a column is missing in the incoming data, it adds the column with the value "Null".

**How the Logic Works Overtime**

1. **Initial State:**
   * The **dimension\_table** starts empty. As new data arrives, the table is populated with column names and their timestamps.
2. **Updating with New Data:**
   * Each time new data is processed, the **update\_dimension\_table** function updates the dimension table with the current columns and their timestamps.
   * If the same columns keep arriving within the threshold time (e.g., every 30 seconds), their timestamps in the dimension table are continuously updated.
3. **Handling Missing Fields:**
   * Before processing, the **handle\_missing\_fields** function ensures that the incoming data contains all the columns listed in the dimension table.
   * If any columns are missing, they are added with the value "Null".
4. **Removing Outdated Columns:**
   * Columns that haven't been updated within the threshold time (e.g., 60 seconds) are removed from the dimension table. This ensures that only relevant columns are tracked.

**Example Scenario**

1. **Time 0s:**
   * Incoming data: {'user\_id': '123', 'app\_version': '1.0'}
   * Dimension table after update: {'user\_id': 0, 'app\_version': 0}
2. **Time 30s:**
   * Incoming data: {'user\_id': '124', 'device\_type': 'android'}
   * Dimension table after update: {'user\_id': 30, 'app\_version': 0, 'device\_type': 30}
   * Processed data: {'user\_id': '124', 'device\_type': 'android', 'app\_version': 'Null'} (fills missing 'app\_version' field)
3. **Time 90s:**
   * Incoming data: {'user\_id': '125', 'app\_version': '1.1'}
   * Dimension table after update: {'user\_id': 90, 'device\_type': 30, 'app\_version': 90}
   * Processed data: {'user\_id': '125', 'app\_version': '1.1', 'device\_type': 'Null'} (fills missing 'device\_type' field)
   * Column **device\_type** is removed from the dimension table as it hasn't been updated in the last 60 seconds this threshold can be modified.

**Scalability**

1. **Horizontal Scaling:**
   * Kafka Brokers: Add more Kafka brokers to distribute the load.
   * Consumers: Increase the number of consumer instances in a consumer group to allow parallel processing.
   * Producers: Scale producers if the data production rate increases.
2. **Partitioning:**
   * Increase Partitions: More partitions allow for better parallel processing.
   * Even Load Distribution: Use keys to distribute data evenly across partitions.
3. **Resource Allocation with Kubernetes:**
   * Deploy Kafka and Zookeeper: Use StatefulSets for Kafka brokers and Zookeeper nodes.
   * Horizontal Pod Autoscaler: Use Kubernetes HPA to automatically scale consumers based on CPU/memory usage or custom metrics like consumer lag.

**Fault Tolerance**

1. **Replication:**
   * Kafka Topics: Set a higher replication factor for Kafka topics to ensure data redundancy.
   * Zookeeper: Deploy multiple Zookeeper nodes for high availability.
2. **Monitoring and Alerts:**
   * Prometheus and Grafana: Monitor Kafka metrics and set up alerts for anomalies.
3. **Consumer Resilience:**
   * Retry Logic: Implement retry logic and use dead-letter topics for messages that fail processing.
   * Idempotent Processing: Ensure consumers can process messages idempotently.
4. **Disaster Recovery:**
   * Backups and Geo-Replication: Regularly back up Kafka data and use tools like MirrorMaker for cross-region replication.

**Conclusion**

This project showcases the power and flexibility of Kafka for real-time data processing and demonstrates best practices for building scalable and efficient data pipelines. The implementation ensures data integrity, fault tolerance, and continuous processing, making it a robust solution for real-time analytics.