**CA – 3**

**Infrastructure for Big data**

**BY – Mitul Srivastava (C00313606)**

**Big Data Architecture and Workflow with Docker, kafka, PostgreSQL, Power BI and OneDrive.**

**Contents**

[**1.** **System Overview** 2](#_Toc184915825)

[**2.** **Components of the Architecture** 3](#_Toc184915826)

[**1.** **Data Generation** 3](#_Toc184915827)

[**2.** **Message Broker** 3](#_Toc184915828)

[**3.** **Data Storage** 3](#_Toc184915829)

[**4.** **Visualization** 3](#_Toc184915830)

[**5.** **Backup** 3](#_Toc184915831)

[**3.** **Docker Containers Creation** 3](#_Toc184915832)

[**Step 1: Create a docker-compose.yaml File** 3](#_Toc184915833)

[**Step 2: Run the Containers** 4](#_Toc184915834)

[**Explanation of Docker Containers** 4](#_Toc184915835)

[**4.** **Python files creation for data ingestion and storage** 5](#_Toc184915836)

[**1.** **Create a kafka\_producer.py file for data generation.** 5](#_Toc184915837)

[**2.** **Create a DBandTable.py file for data generation.** 6](#_Toc184915838)

[**3.** **Create a kafka\_to\_postgres.py file for data generation** 7](#_Toc184915839)

[**5.** **Workflow Steps** 8](#_Toc184915840)

[**Step 1: Data Generation** 8](#_Toc184915841)

[**Step 2: Message Brokering** 8](#_Toc184915842)

[**Step 3: Database and Table creation in Postgres** 8](#_Toc184915843)

[**Step 4: Data Processing and Storage** 8](#_Toc184915844)

[**Step 5: Visualization** 8](#_Toc184915845)

[**Step 6: Backup** 9](#_Toc184915846)

[**6.** **Tools and Software Used** 9](#_Toc184915847)

[**7.** **Innovation and Optimization** 9](#_Toc184915848)

[**7.1 Innovations** 9](#_Toc184915849)

[**1.** **Integrated Data Workflow with Diverse Tools** 9](#_Toc184915850)

[**2.** **Synthetic Data Generation with faker** 9](#_Toc184915851)

[**3.** **Real-Time Processing Capability** 9](#_Toc184915852)

[**4.** **Backup Strategy with Cloud Integration** 10](#_Toc184915853)

[**7.2 Optimizations** 10](#_Toc184915854)

[**1. Containerized Architecture** 10](#_Toc184915855)

[**2.** **Efficient Data Flow Design** 10](#_Toc184915856)

[**3.** **Lightweight Python Scripts** 10](#_Toc184915857)

[**4.** **Use of Power BI for Interactive Dashboards** 10](#_Toc184915858)

[**5.** **Comprehensive Backup Command** 10](#_Toc184915859)

[**8.** **Issues faced and resolution** 10](#_Toc184915860)

[**Error 1: Docker Containers Failing to Start** 10](#_Toc184915861)

[**Error 2: Kafka Producer Unable to Connect to Kafka Broker** 10](#_Toc184915862)

[**Error 3: Database Connection Refused** 11](#_Toc184915863)

[**Error 4: Data Loss in Kafka Topics** 11](#_Toc184915864)

[**Error 5: Power BI Unable to Connect to PostgreSQL** 11](#_Toc184915865)

[**Error 6: kafka-python Consumer Timeout** 11](#_Toc184915866)

[**Error 7: Backup Script Permissions Denied** 11](#_Toc184915867)

[**Error 8: Data Format Mismatch Between Kafka and PostgreSQL** 11](#_Toc184915868)

[**9.** **References** 11](#_Toc184915869)

# **System Overview**

This Big data architecture project utilizes Docker desktop for containerization. Utilizing kafka as the message broker and PostgresSQL for data storage. Python scripts have been utilized to automate and make the process of data generation and storage easy and efficient. The architecture also contains Power BI for data visualization and Onedrive to backup postgres data.

# **Components of the Architecture**

## **Data Generation**

* **Tool:** Kafka Producer (Python).
* **Description:** Simulates realistic message data using faker.
* **Purpose:** Publishes messages to Kafka topics.

## **Message Broker**

* **Tool:** Apache Kafka (Containerized with Zookeeper).
* **Description:** Acts as a buffer for real-time data streams between the producer and consumer.

## **Data Storage**

* **Tool:** PostgreSQL (Containerized).
* **Description:** Stores processed data from Kafka for analysis and visualization.

## **Visualization**

* **Tool:** Power BI.
* **Description:** Visualizes PostgreSQL-stored data through interactive dashboards.

## **Backup**

* **Tool:** OneDrive.
* **Description:** Ensures secure and accessible PostgreSQL backups.

# **Docker Containers Creation**

## **Step 1: Create a docker-compose.yaml File**

The docker-compose.yaml file defines the services needed for the system. Here is an overview of the file:

A screenshot of a computer program

Description automatically generated

## **Step 2: Creating the Containers in Docker**

Start all the defined services using Docker Compose:

**Command: docker-compose up -d**

### **Explanation of Docker Containers**

1. **Zookeeper Container**:
   * Coordinates distributed Kafka brokers.
   * Listens on port 2181.
   * Essential for managing Kafka's metadata and cluster state.
2. **Kafka Container**:
   * Hosts the Kafka broker that handles messages.
   * Listens on port 9092.
   * Communicates with producers and consumers.
3. **PostgreSQL Container**:
   * Runs the PostgreSQL database instance.
   * Stores data consumed from Kafka.

# **Python files creation for data ingestion and storage**

## **Create a kafka\_producer.py file for data generation.**

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

## **Create a DBandTable.py file for data generation.**

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

## **Create a kafka\_to\_postgres.py file for data generation**

**A screen shot of a computer

Description automatically generated**

**A computer screen shot of a program

Description automatically generated**

# **Workflow Steps**

## **Step 1: Data Generation**

1. Start the Kafka producer:
2. **Command: python kafka\_producer.py**
3. Produces realistic messages and publishes them to Kafka.

## **Step 2: Message Brokering**

1. Kafka receives and queues messages in the **flink\_topic**.

## **Step 3: Database and Table creation in Postgres**

1. **Command: python DBandTable.py**

## **Step 4: Data Processing and Storage**

1. Start the Kafka consumer:
2. **Command: python kafka\_to\_postgres.py**
3. Consumes messages, deserializes JSON, and stores them in PostgreSQL.

## **Step 5: Visualization**

1. Use Power BI to connect to PostgreSQL.
2. Create visualizations to analyze the processed data.

## **Step 6: Backup**

1. Backup PostgreSQL:
2. **Command: pg\_dump -U postgres -h localhost MSdb > C:\Users\Mitul\OneDrive\Bigdata\_backup/MSdb\_backup.sql**
3. Sync to OneDrive:

# **Tools and Software Used**

1. **Python Libraries:** kafka-python, psycopg2, faker.
2. **Docker Desktop:** Containerizes and orchestrates services.
3. **VS CODE:** for .YAML and .PY files creation.
4. **Power BI:** Visualizes PostgreSQL data.
5. **OneDrive:** Secures backups.

# **Innovation and Optimization**

## **7.1 Innovations**

### **Integrated Data Workflow with Diverse Tools**

* + Combining **Kafka**, **PostgreSQL**, **Power BI**, and **OneDrive** in a seamless pipeline ensures data flows from generation to visualization and backup without manual intervention. Using containerized services through **Docker** enhances portability and simplifies deployment, which adds a layer of innovation by making the architecture scalable and reproducible.

### **Synthetic Data Generation with faker**

* + Incorporating the faker library to simulate realistic data provides flexibility to test the architecture with dynamic data without relying on pre-existing datasets. This adaptability is innovative in the context of demonstrating proof-of-concept infrastructure.

### **Real-Time Processing Capability**

* + Implementing Kafka for message brokering enables real-time processing, which is critical for big data applications. This showcases the ability to handle live data streams, which is a step beyond traditional batch processing.

### **Backup Strategy with Cloud Integration**

* + Utilizing **OneDrive** for automated backups demonstrates a forward-thinking approach to ensuring data security and recovery. Cloud integration for backups adds reliability and modernity to the architecture.

## **7.2 Optimizations**

### **1. Containerized Architecture**

* + Using **Docker Compose** to containerize Kafka, Zookeeper, and PostgreSQL minimizes setup complexity and ensures resource efficiency. This optimization simplifies scaling and deployment across different environments.

### **Efficient Data Flow Design**

* + The pipeline is structured to ensure minimal latency between data ingestion (Kafka producer) and storage (PostgreSQL). This direct and streamlined data flow eliminates unnecessary processing delays.

### **Lightweight Python Scripts**

* + The Python scripts (kafka\_producer.py, DBandTable.py, kafka\_to\_postgres.py) are optimized for modularity and focus on specific tasks. This reduces redundancy and improves maintainability.

### **Use of Power BI for Interactive Dashboards**

* + Power BI's ability to query PostgreSQL directly ensures visualization updates are real-time and efficient. This reduces the overhead of exporting data into intermediate formats.

### **Comprehensive Backup Command**

* + Using **pg\_dump** to create backups ensures a compact and consistent backup process. Syncing it directly to OneDrive avoids the need for additional tools or manual uploads, which saves time and effort.

# **Issues faced and resolution**

## **Error 1: Docker Containers Failing to Start**

* **Issue**: Containers for Kafka and Zookeeper did not start due to port conflicts.
* **Resolution**: Identified conflicting processes using netstat and updated the docker-compose.yaml file to assign non-conflicting ports.

## **Error 2: Kafka Producer Unable to Connect to Kafka Broker**

* **Issue**: The producer script could not establish a connection to the Kafka broker.
* **Resolution**: Verified the Kafka broker hostname and port in the kafka\_producer.py file and ensured they matched the Docker network settings. Added retries to handle connection delays.

## **Error 3: Database Connection Refused**

* **Issue**: PostgreSQL container was inaccessible from Python scripts.
* **Resolution**: Updated PostgreSQL configuration in the docker-compose.yaml to allow external connections. Verified credentials and added a delay in the scripts to allow PostgreSQL to initialize.

## **Error 4: Data Loss in Kafka Topics**

* **Issue**: Messages published to Kafka were getting lost due to improper topic configuration.
* **Resolution**: Configured the Kafka topic with appropriate replication and retention policies in the producer script.

## **Error 5: Power BI Unable to Connect to PostgreSQL**

* **Issue**: Power BI could not establish a connection with the database hosted in Docker.
* **Resolution**: Used the host machine's IP instead of localhost and ensured the PostgreSQL container exposed the correct port.

## **Error 6: kafka-python Consumer Timeout**

* **Issue**: Consumer in kafka\_to\_postgres.py timed out while consuming messages.
* **Resolution**: Increased the timeout parameter in the consumer configuration and optimized topic reading logic.

## **Error 7: Backup Script Permissions Denied**

* **Issue**: The pg\_dump command failed due to insufficient file system permissions.
* **Resolution**: Updated the script to use a directory with proper write permissions and added error handling to log issues.

## **Error 8: Data Format Mismatch Between Kafka and PostgreSQL**

* **Issue**: Data consumed from Kafka did not match the PostgreSQL table schema.
* **Resolution**: Implemented a JSON deserialization step in kafka\_to\_postgres.py to validate and transform data before insertion.

# **References**

* 1. <https://chatgpt.com/c/>
  2. <https://towardsdatascience.com/how-i-dockerized-apache-flink-kafka-and-postgresql-for-real-time-data-streaming-c4ce38598336>
  3. https://www.youtube.com/results?search\_query=docker+and+apache+flink.