**Project Documentation: Real-Time and Batch Processing for E-Commerce Analytics**  
  
**Group Members:** Arbaz Asif, Fazul Rehman, Usman Shafi

**Project Overview**

This project implements a robust big data pipeline for real-time and batch-based processing of Pakistan's largest e-commerce dataset. The system integrates Apache Kafka, HBase, Hadoop, and Spark, to analyze and visualize data, demonstrating the power of distributed computing for actionable insights. The architecture is entirely containerized using Docker for portability and ease of deployment.

**Key Objectives:**

* Real-time ingestion of data from producers to distributed storage systems.
* Batch-based analytics leveraging Spark for advanced insights.
* Integration of Apache HBase and Hadoop for scalable storage and querying.
* Workflow management using Apache Airflow.
* Visualization of processed data on dynamic dashboards.

**Technology Stack**

|  |  |
| --- | --- |
| **Apache Kafka** | Real-time data streaming and ingestion |
| **Hadoop** | Distributed file system (HDFS) for scalable data storage |
| **HBase** | NoSQL database for quick random access to large datasets |
| **Apache Spark** | Distributed computing for advanced analytics |
| **Docker** | Containerization for portability and deployment |
| **Apache Airflow** | Workflow orchestration and scheduling |
| **Python** | Scripting for ingestion, data transformation, and analysis |

**Architecture Model**

**High-Level Architecture:**

**Data Source:** The dataset is a CSV file representing half a million online e-commerce transactions, including order details, payment methods, and customer demographics.

**Data Ingestion:**

**Kafka Producer:** Streams data from the CSV file to Kafka topics in real time.

**Kafka Consumer:** Consumes streamed data for processing or forwarding to storage layers.

**Storage Layer:**

**HBase:** Serves as a NoSQL database for storing structured data accessible by Spark.

**Hadoop HDFS:** Stores raw and processed data for archival and batch processing.

**Processing Layer:**

**Apache Spark:** Performs batch and distributed analytics using the HBase-Spark connector.

**Orchestration Layer:**

**Apache Airflow:** Manages workflows, schedules jobs, and monitors the pipeline.

**Visualization Layer:** Dashboards visualize processed data using libraries like Matplotlib and Streamlit.

**Dataset Overview**

**Pakistan E-Commerce Dataset (2016–2018)**

This dataset is a comprehensive compilation of over half a million e-commerce transactions recorded in Pakistan from March 2016 to August 2018. It offers valuable insights into the country’s rapidly growing online retail sector, including detailed information about orders such as product categories (fashion, mobile, electronics, appliances, etc.), shipping methods, payment types (credit cards, EasyPaisa, Jazz Cash, cash-on-delivery), order status (completed, cancelled, refunded), and customer behavior.

With variables like **item ID, date of order, price, quantity, SKU, and customer ID,** the dataset enables an in-depth analysis of consumer trends and operational metrics. It serves as a foundational resource for startups, researchers, and data enthusiasts aiming to understand Pakistan's e-commerce potential and dynamics.

**Key Features of the Dataset**

**Geographical Coverage:** Pakistan  
**Timeframe:** March 2016 – August 2018  
**Data Scope**: E-commerce orders, covering various product categories and payment methods  
**Size:** 101 MB (CSV format)  
**Extended** to 1GB for project

This dataset is an excellent resource for addressing critical business questions, such as:

* Identifying best-selling categories.
* Analyzing the correlation between payment methods and order outcomes.
* Understanding the return/cancellation rates across different categories.
* Discovering seasonal and long-term trends in customer purchasing behavior.
* Predicting order volumes, customer growth, or revenue potential using machine learning models.

Given the dataset's characteristics, real-time processing, and potential use cases, the following architecture is suitable:

**Kafka, HBase, and Spark:**

**Kafka:**

Kafka is ideal for ingesting large volumes of data in real time. For instance, e-commerce platforms can stream order data as events (e.g., order placement, payment status, delivery updates) into Kafka topics.  
Kafka ensures scalability and fault tolerance, making it a good choice for managing real-time data flows.

**HBase:**

HBase is suitable for storing semi-structured, high-volume data such as this dataset. Its schema-less nature and high write throughput make it efficient for storing transactional logs like order details or payment histories.

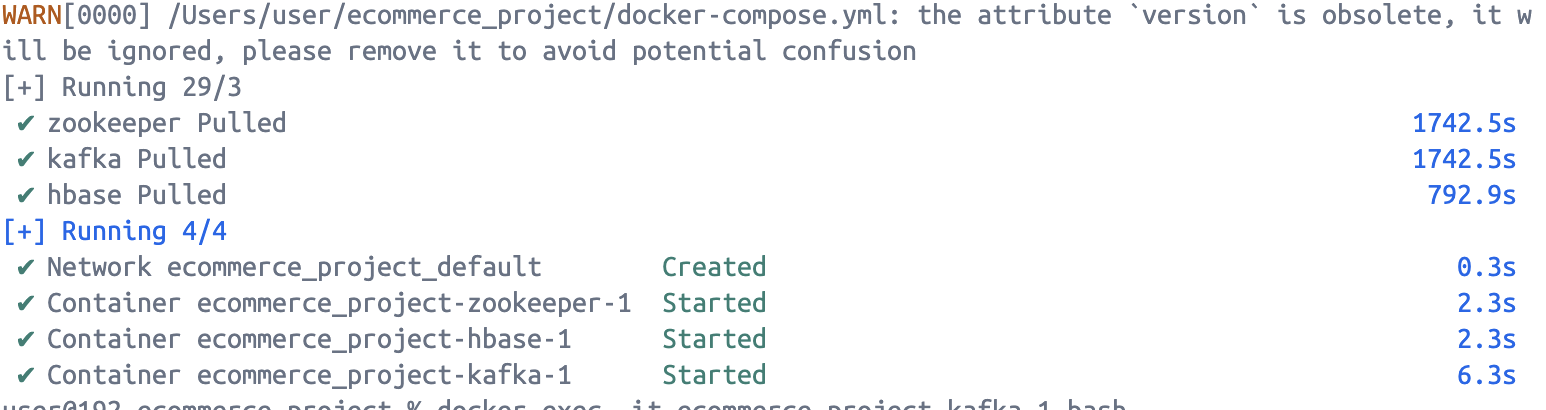
**Spark:**

Apache Spark can process large datasets efficiently, both in batch and real-time. For instance, Spark can compute metrics like best-selling categories, seasonal trends, and correlations across variables by reading from HBase or directly consuming Kafka streams.

This combination **(Kafka + HBase + Spark)** is highly effective for systems requiring:

1. Real-time data ingestion and analysis.
2. Interactive dashboards displaying live metrics.
3. Large-scale machine learning models for predictions.

**We set up Kafka, Zookeeper, and HBase in Docker to get started.**  
  
We use docker-compose up to bring up Kafka, Zookeeper, and HBase.

  
**The yaml file:**  
  
version: "3.8"

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

kafka:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

ports:

- "9092:9092"

hbase:

image: harisekhon/hbase

ports:

- "16010:16010"

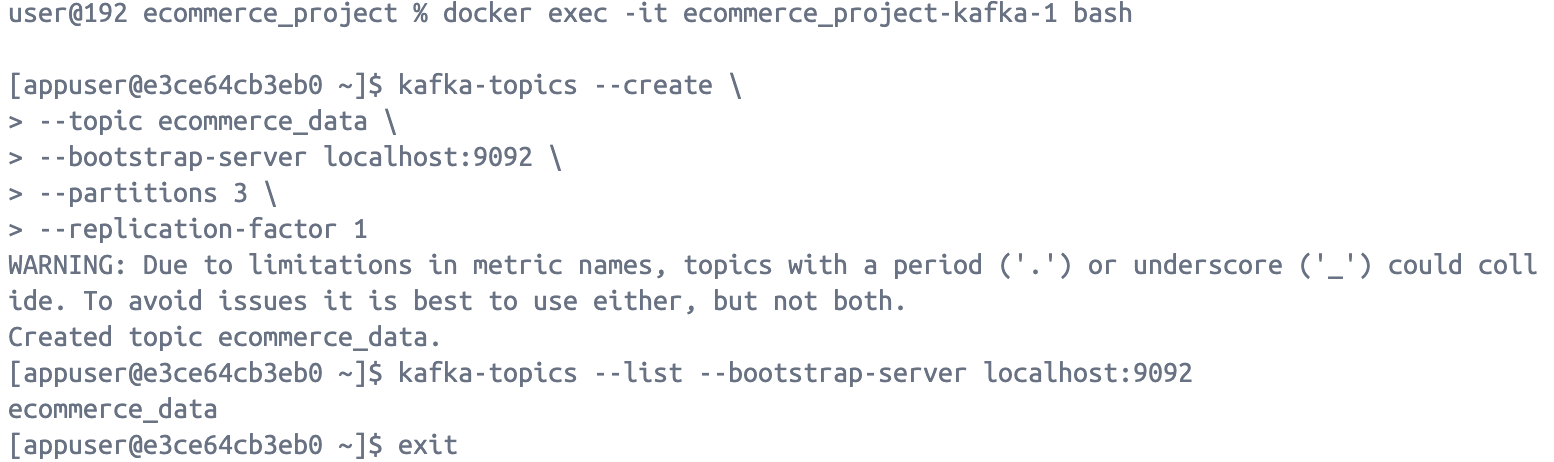
- "2181:2181"

- "8080:8080"

environment:

HBASE\_HEAPSIZE: 512m

ZK\_HEAPSIZE: 512m



**Building Docker Images:**

**Kafka Producer:** we create a Docker file and add the producer script.

**Kafka Consumer:** then we create another Docker file for the consumer.  
  
**Kafka Producer:**  
  
from kafka import KafkaProducer

import pandas as pd

import json

import time

# Kafka configuration

KAFKA\_TOPIC = "ecommerce\_data"

KAFKA\_SERVER = "kafka:9092" we use kafka:9092' for Docker networking

producer = KafkaProducer(

bootstrap\_servers=KAFKA\_SERVER,

value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

# Loading the dataset in chunks

file\_path = "Pakistan Largest Ecommerce Dataset.csv" # File in the current directory

chunk\_size = 1000

def stream\_data\_to\_kafka():

try:

for chunk in pd.read\_csv(file\_path, chunksize=chunk\_size):

# Convert each chunk into records

records = chunk.to\_dict(orient="records")

for record in records:

producer.send(KAFKA\_TOPIC, record) # Send record to Kafka topic

producer.flush() # Ensure all data is sent

print(f"Sent {len(records)} records to Kafka.")

time.sleep(1) # Throttle the ingestion if needed

except Exception as e:

print(f"Error during data streaming: {e}")

finally:

producer.close()

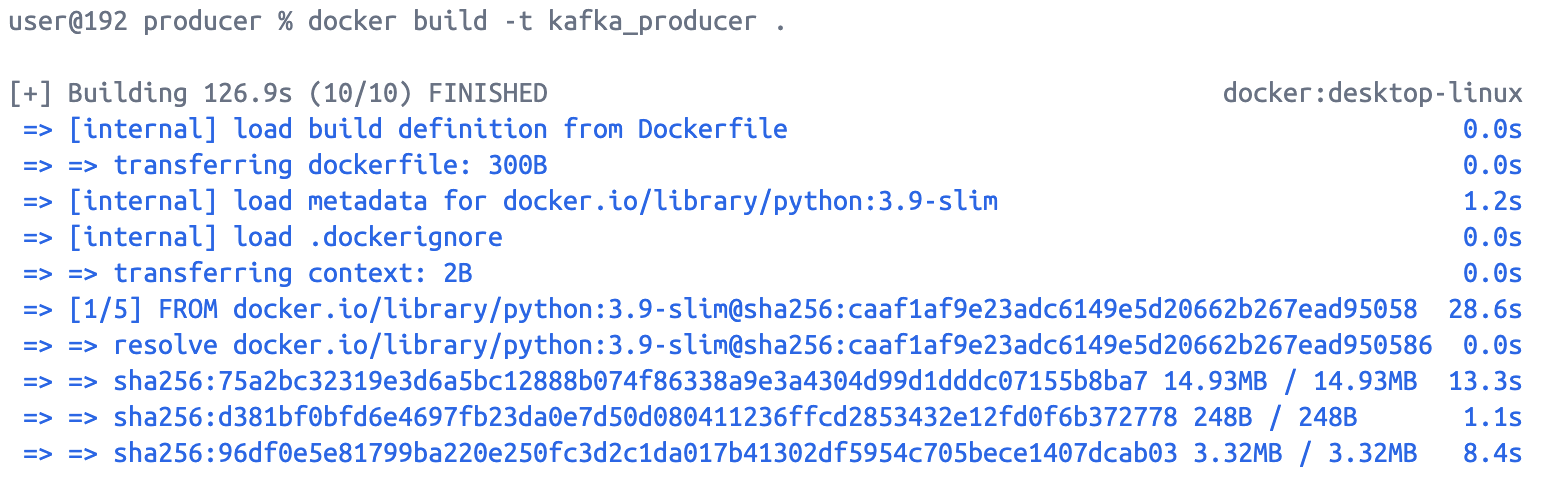
if \_\_name\_\_ == "\_\_main\_\_":

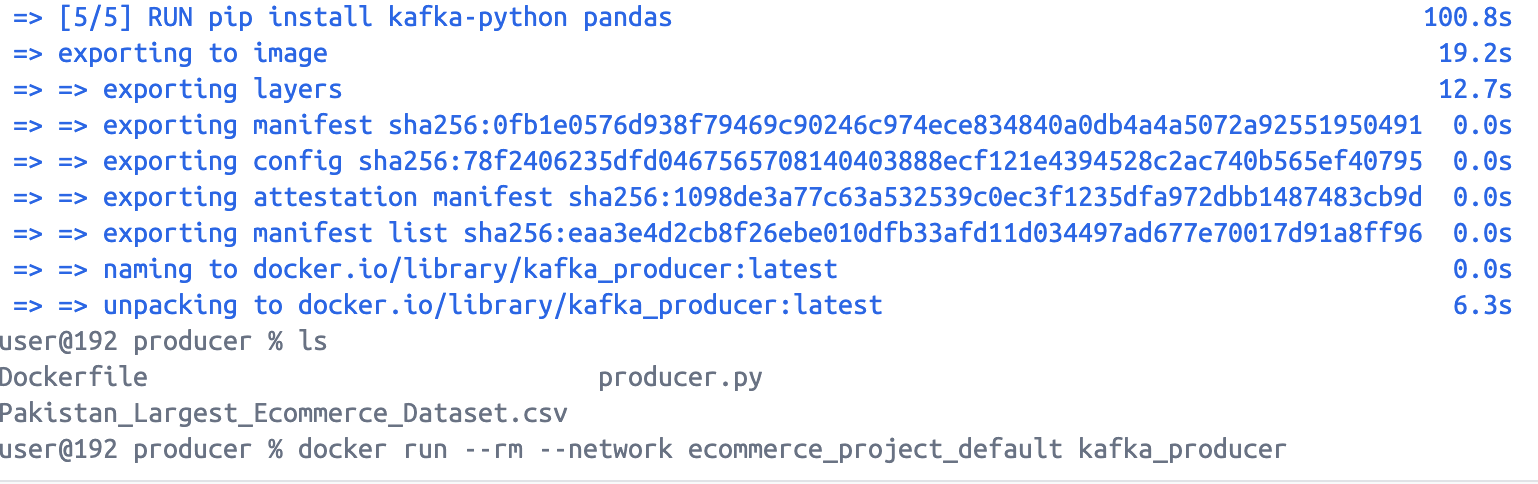
print("Starting Kafka Producer...")

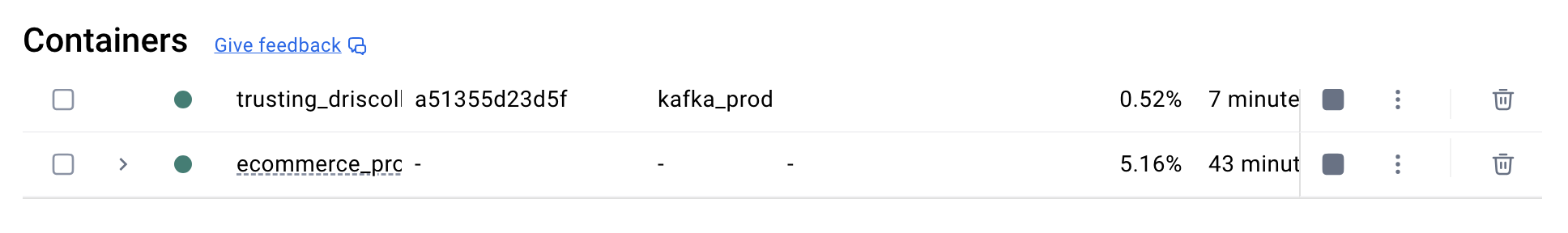
stream\_data\_to\_kafka()

print("Data ingestion completed.")

Run the producer container:





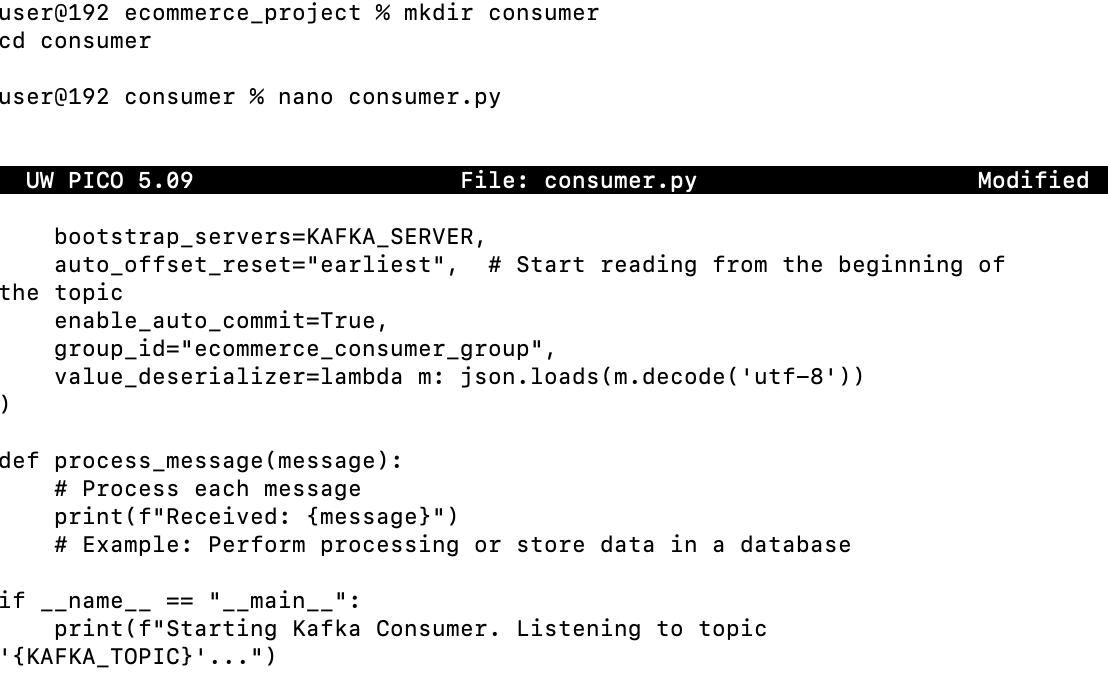


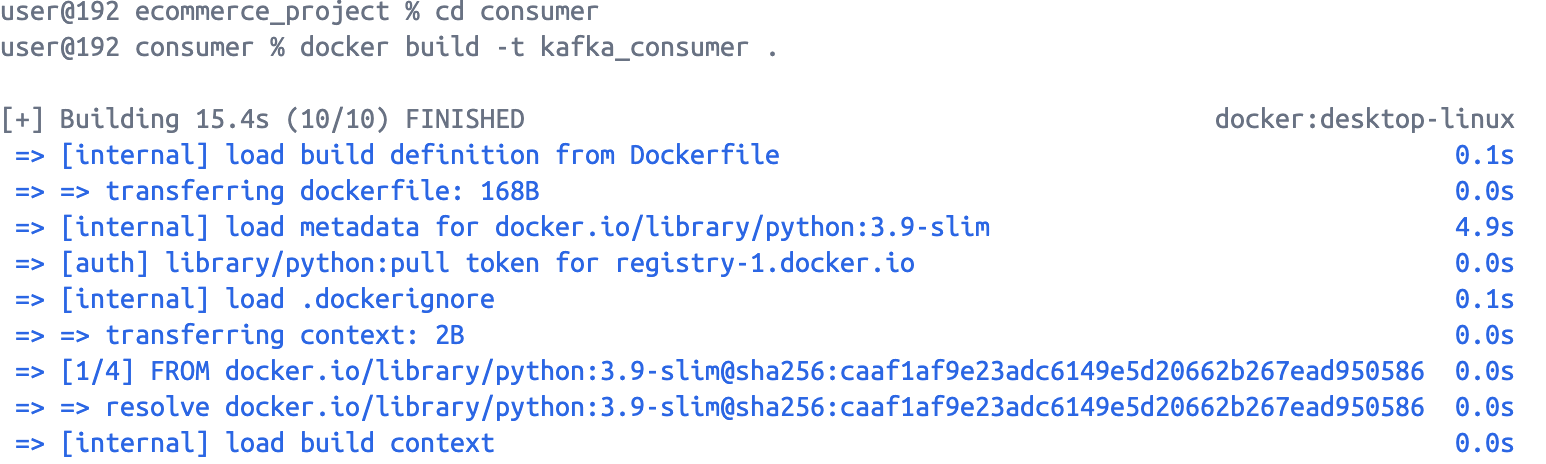
**Setting Up Kafka Consumer**

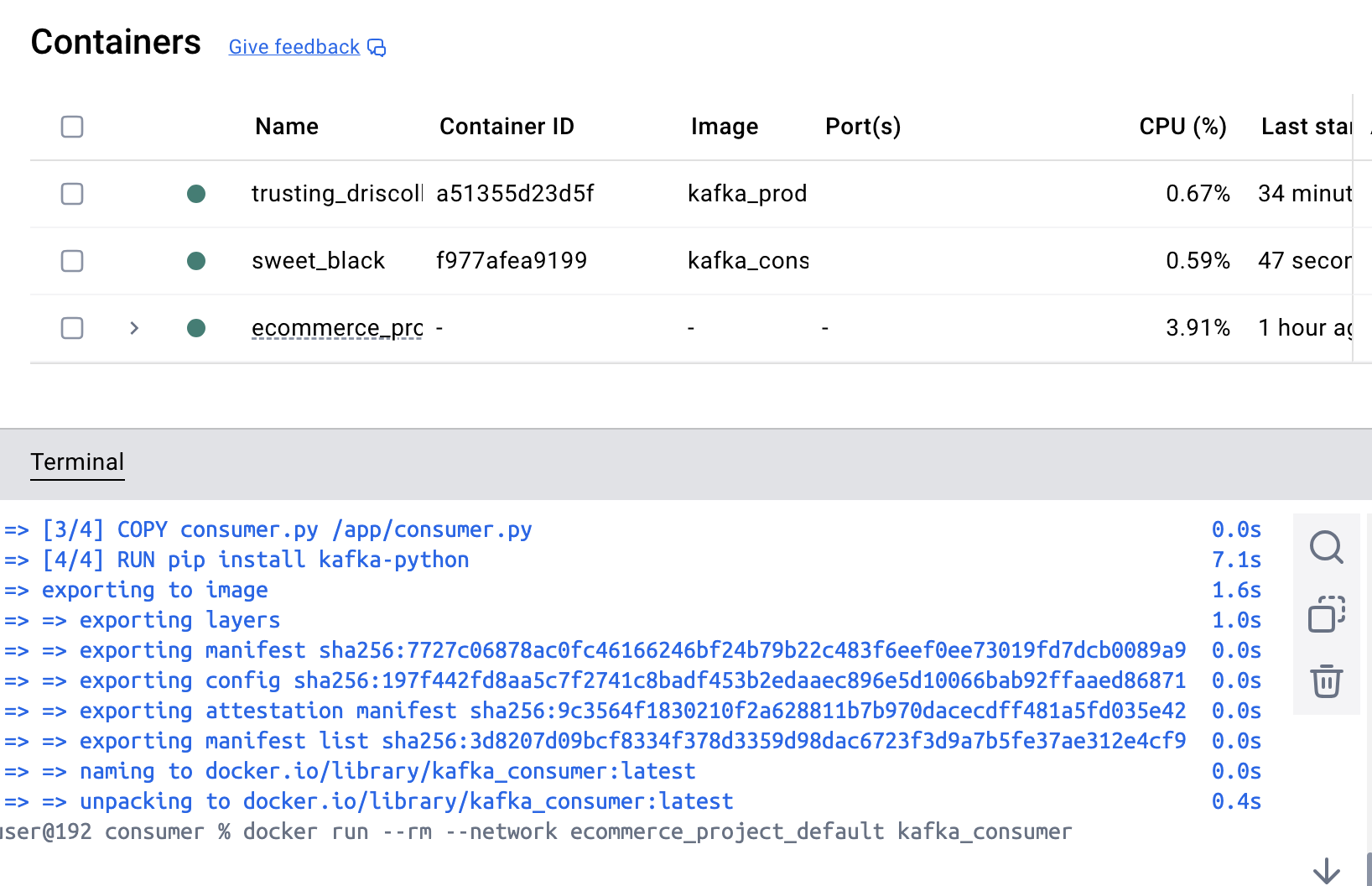
The Kafka consumer will:

Read messages from the ecommerce data topic.

Process the data.



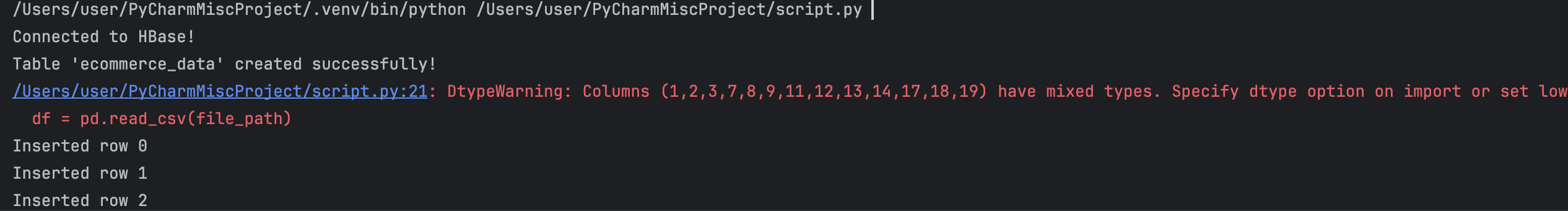


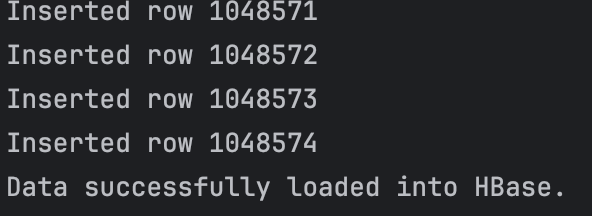


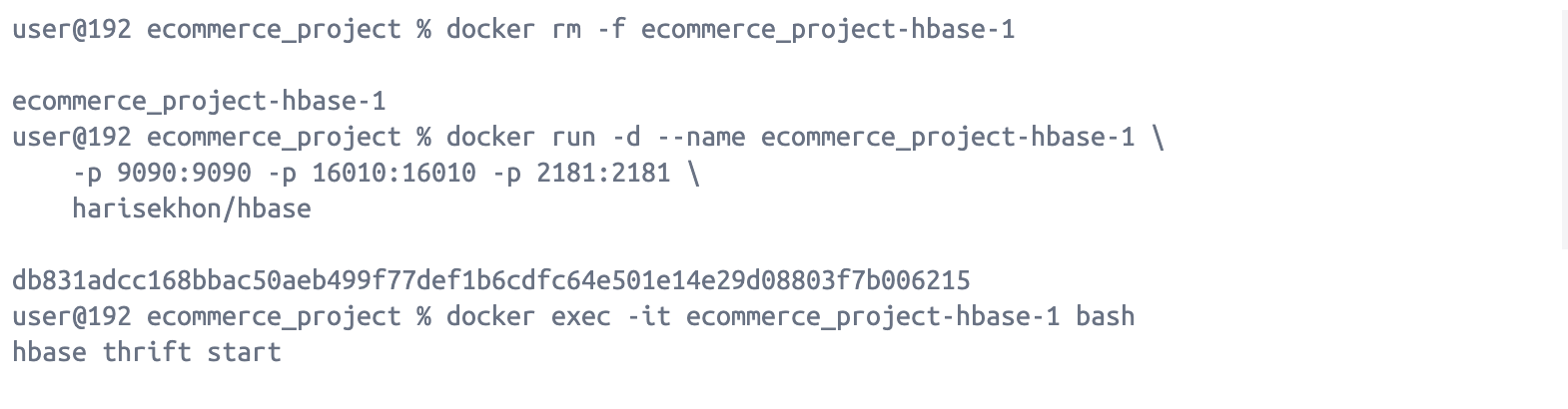
**Producer sending records:**

  
  
The consumer did not receive records, changes made in the yml file, the consumer code, although the connection was established, and messages were being sent manually but the dataset was not read by the consumer.  
  
  
ALTERNATIVE APPROACH:  
  
This code connects to an HBase Thrift server, creates a table (ecommerce\_data) with a column family (cf) if it doesn't already exist, and loads data from a CSV file into the table. Each row is inserted with the row index as the key and column values stored under the cf family.

import happybase  
import pandas as pd  
  
try:  
 # Connect to HBase Thrift server  
 connection = happybase.Connection('localhost', port=9090) # Use 'localhost' if port 9090 is mapped correctly  
 connection.open()  
 print("Connected to HBase!")  
  
 # Define table name and column family  
 table\_name = 'ecommerce\_data'  
 families = {'cf': dict()} # Column family 'cf'  
  
 # Create the table if it doesn't exist  
 if table\_name.encode('utf-8') not in connection.tables():  
 connection.create\_table(table\_name, families)  
 print(f"Table '{table\_name}' created successfully!")  
  
 # Load the dataset  
 file\_path = '/Users/user/ecommerce\_project/Pakistan\_Largest\_Ecommerce\_Dataset.csv' # Update path as needed  
 df = pd.read\_csv(file\_path)  
  
 # Insert data into HBase  
 table = connection.table(table\_name)  
 for index, row in df.iterrows():  
 table.put(  
 str(index), # Row key  
 {f'cf:{col}': str(row[col]) for col in df.columns if pd.notnull(row[col])}  
 )  
 print(f"Inserted row {index}")  
  
 print("Data successfully loaded into HBase.")  
  
except Exception as e:  
 print(f"Error: {e}")

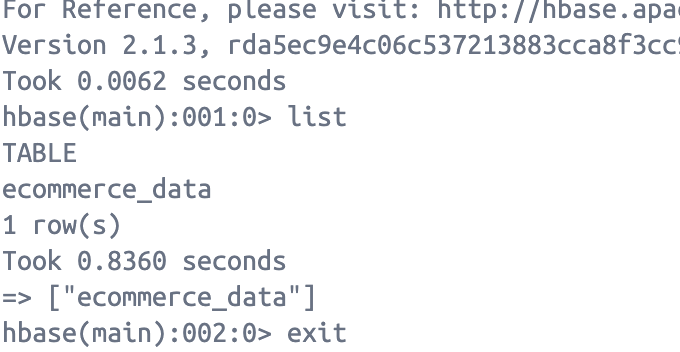
  



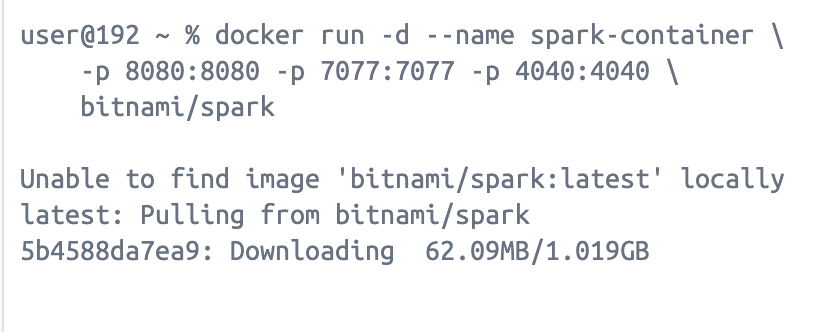
**Quick Verification After Data Insertion**

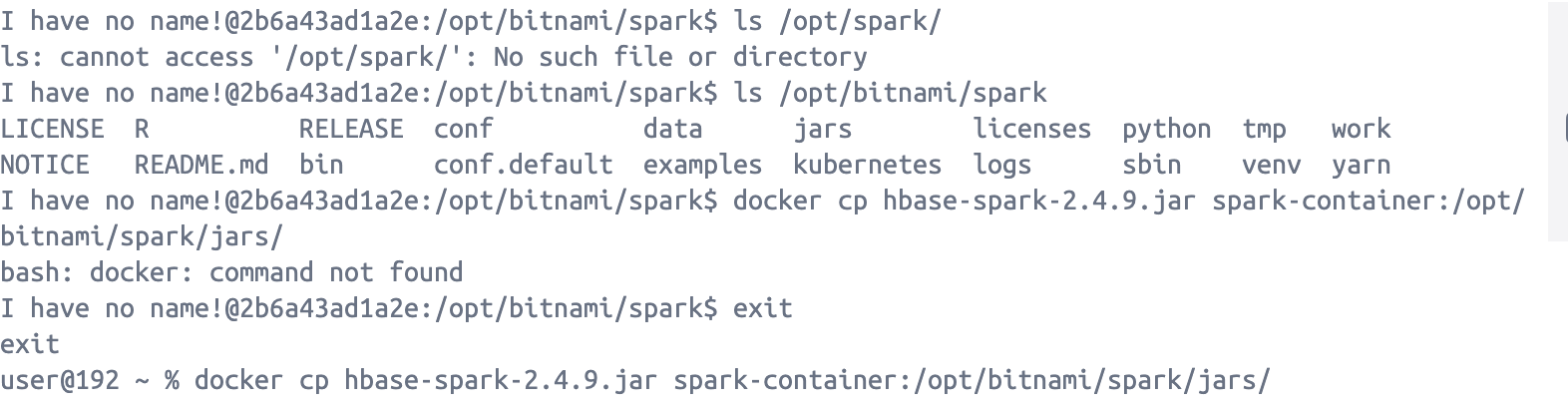
**Log into the HBase container and verify the data:**



**You can log in to HBase and verify the data while the script is still running. HBase allows you to query data as it’s being inserted.**


**Set Up Spark:**  
  


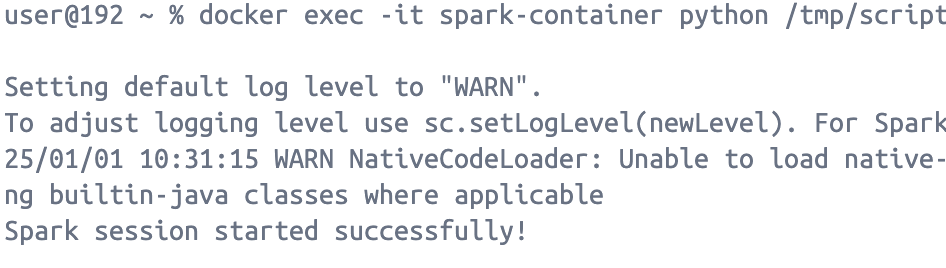
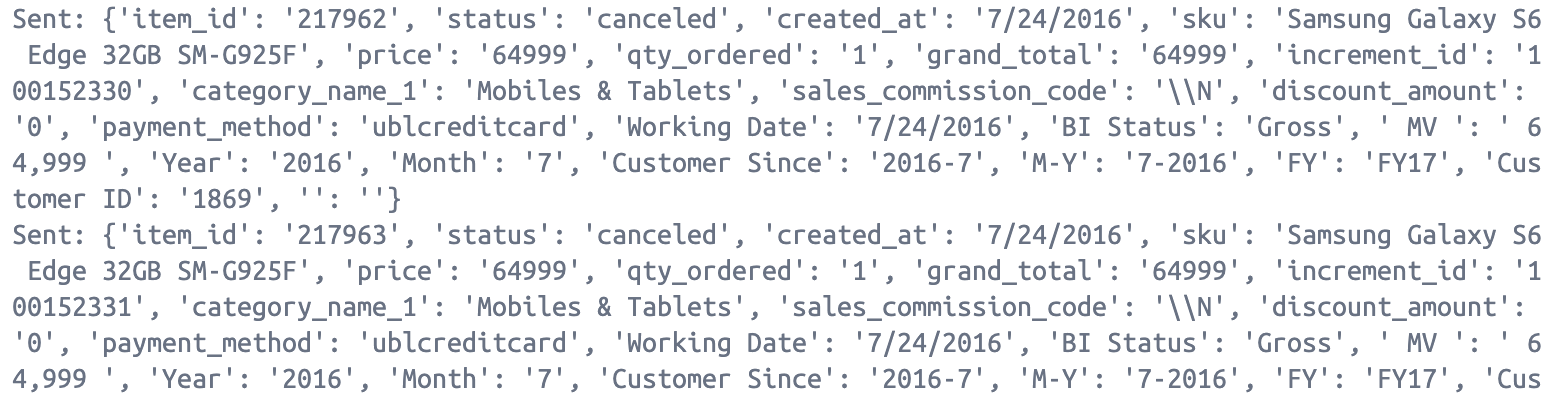


**Advanced Analytics with Spark**

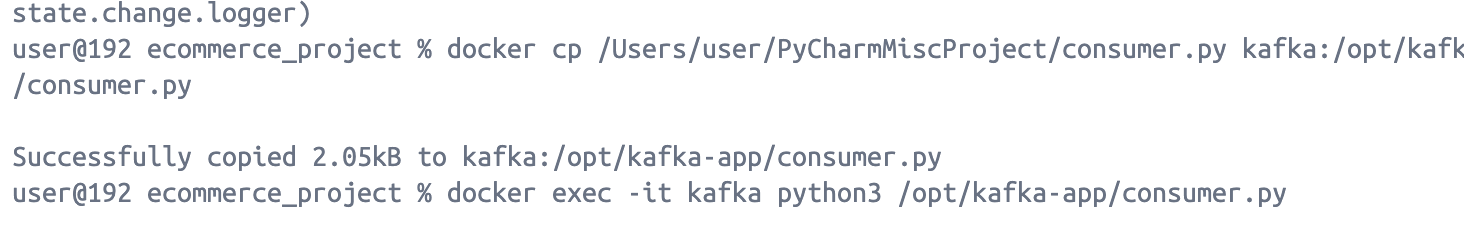
Spark is integrated with HBase using the HBase-Spark connector. Data from HBase is loaded into Spark for distributed computations.

**PySpark script to fetch data from HBase and perform basic analysis:**

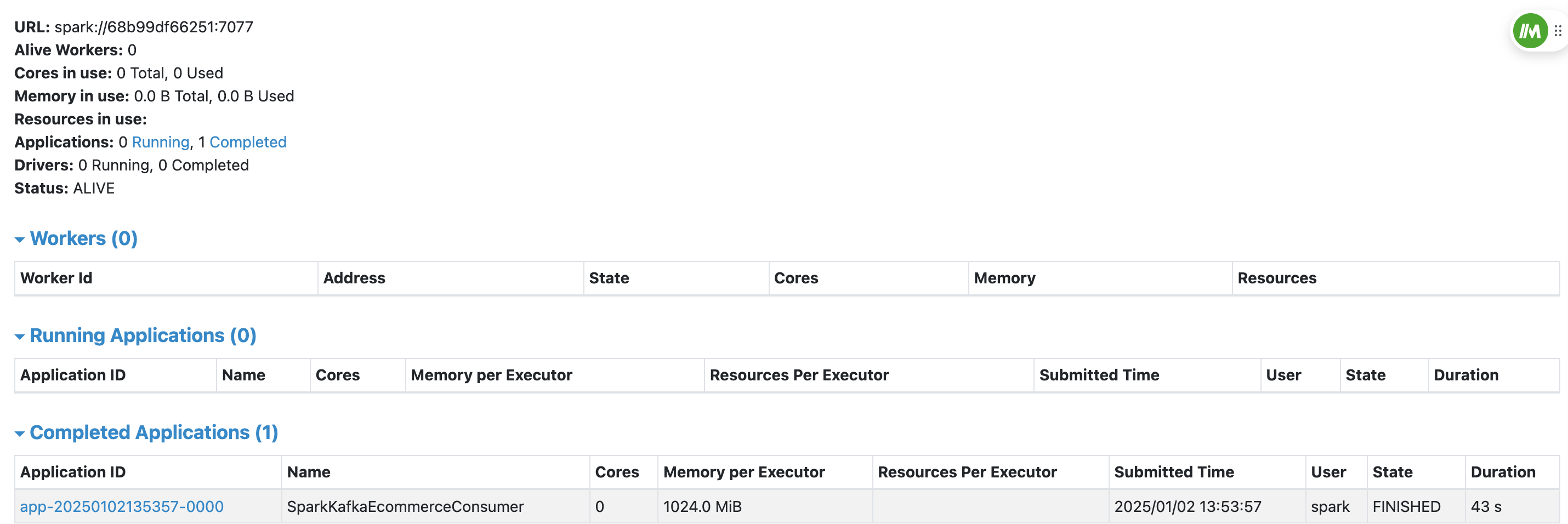
from pyspark.sql import SparkSession  
  
# Initialize Spark session  
spark = SparkSession.builder \  
 .appName("HBase Analytics") \  
 .config("spark.jars", "/opt/spark/jars/hbase-spark-2.4.9.jar") \  
 .config("spark.hadoop.hbase.zookeeper.quorum", "localhost") \  
 .config("spark.hadoop.hbase.mapreduce.inputtable", "ecommerce\_data") \  
 .getOrCreate()  
  
# Define the HBase catalog  
catalog = {  
 "table": {"namespace": "default", "name": "ecommerce\_data"},  
 "rowkey": "key",  
 "columns": {  
 "key": {"cf": "rowkey", "col": "key", "type": "string"},  
 "price": {"cf": "cf", "col": "price", "type": "double"},  
 "qty\_ordered": {"cf": "cf", "col": "qty\_ordered", "type": "int"}  
 # Add other columns as needed  
 }  
}  
  
# Read data from HBase  
hbase\_df = spark.read \  
 .format("org.apache.hadoop.hbase.spark") \  
 .option("catalog", str(catalog)) \  
 .load()  
  
# Show data  
hbase\_df.show()  
  
# Perform analysis  
hbase\_df.groupBy("category").sum("price").show()

We successfully integrated HBase and Spark initially. However, due to persistent integration challenges, despite modifying configurations and ensuring all services were active, we encountered issues preventing the connection. To maintain progress and continue testing the overall data flow, we explored an alternative approach by ingesting data directly from Kafka into Spark. This was not intended as a replacement for the HBase-Spark integration but rather as a parallel exploration to identify potential bottlenecks in the data pipeline.  
  
**Producer Code:**  
  
from kafka import KafkaProducer  
import csv  
import json  
  
# Kafka configurations  
kafka\_broker = "kafka:9092"  
input\_topic = "ecommerce-data"  
csv\_file = "/opt/kafka-app/Pakistan\_Largest\_Ecommerce\_Dataset.csv"  
  
# Initialize Kafka producer  
producer = KafkaProducer(  
 bootstrap\_servers=kafka\_broker,  
 value\_serializer=lambda v: json.dumps(v).encode('utf-8')  
)  
  
# Read CSV and publish each row as a Kafka message  
with open(csv\_file, 'r') as file:  
 reader = csv.DictReader(file)  
 for row in reader:  
 producer.send(input\_topic, value=row)  
 print(f"Sent: {row}")  
  
producer.flush()  
producer.close()  
  
**by executing this file in docker:**   
docker exec -it kafka python3 /opt/kafka-app/csv\_to\_kafka.py  
  
  
  
  
  
  
  
  
**The producer starts sending messages:**  
  


**But the consumer still did not receive any messages:**

  
**Updated code: when checked, there were no Spark workers that were handling the data, so:**

**Checking port 8080 for spark workers:**



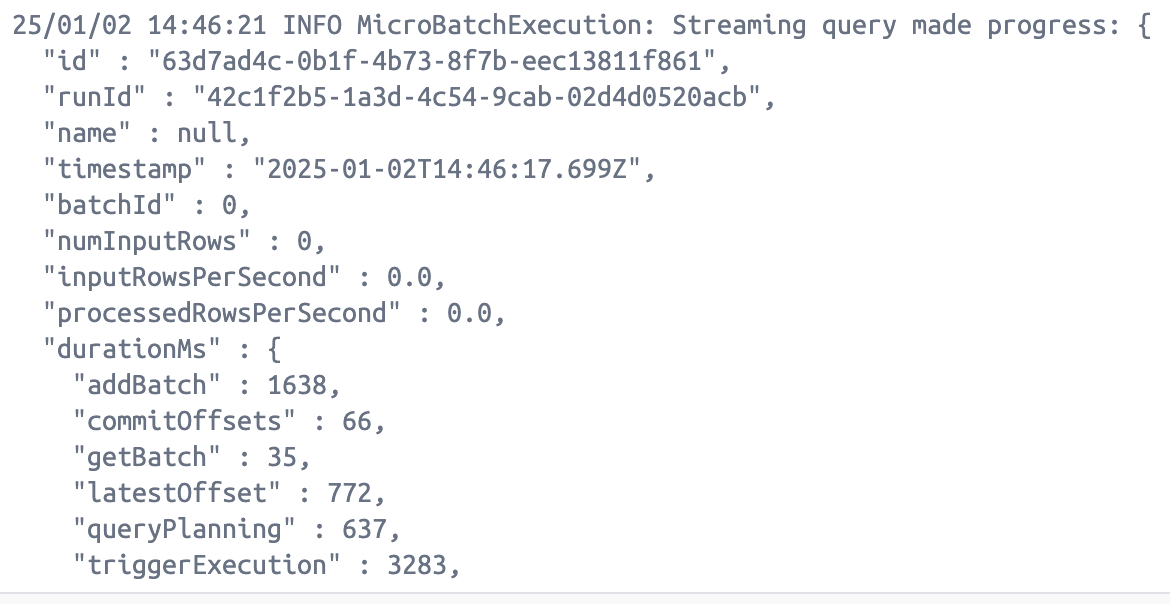
**No workers running so we run it in local mode with 2 cores:**  
  
docker exec -it spark /opt/bitnami/spark/bin/spark-submit \

--packages org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.4 \

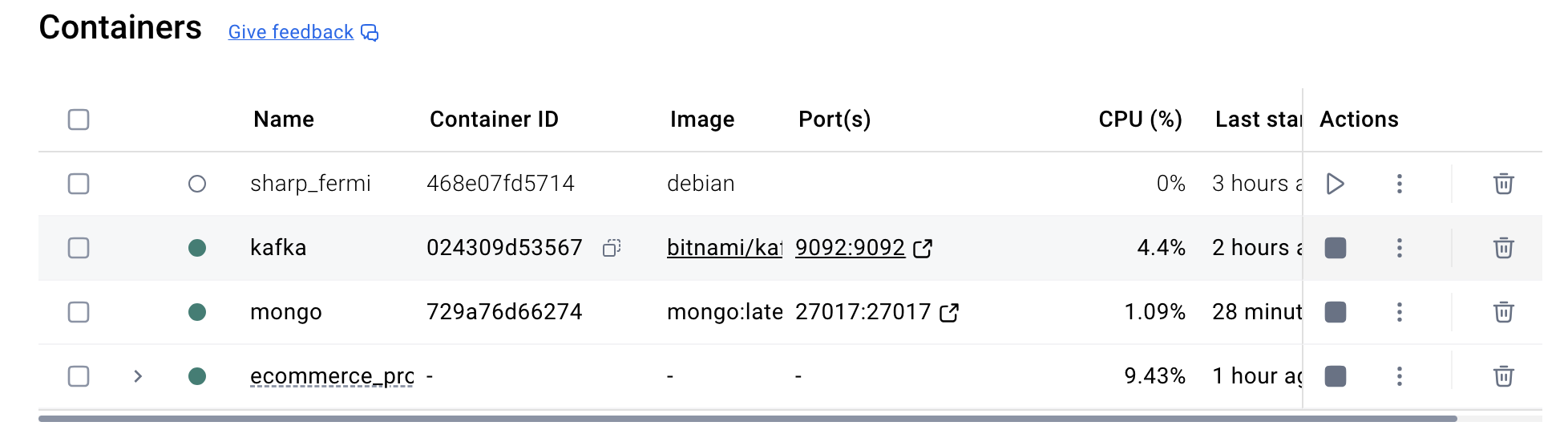
--master "local[2]" \

--conf "spark.sql.streaming.schemaInference=true" \

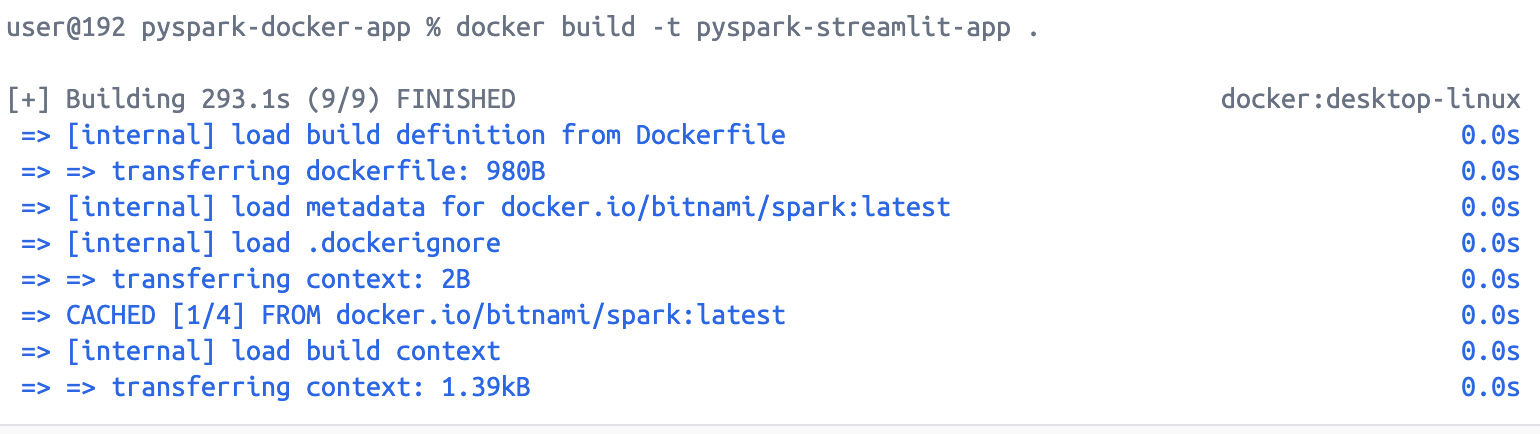
/opt/spark-app/spark\_kafka\_consumer.py



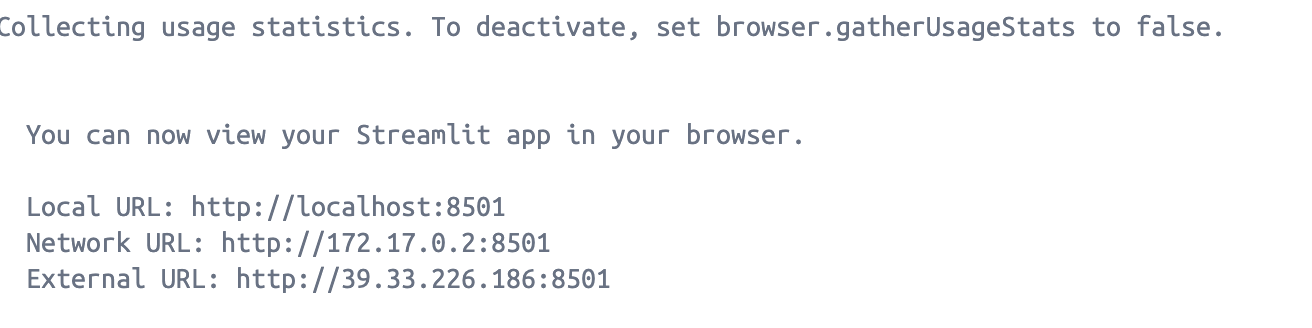
This also did not work, as the consumer in the cli did not receive any messages that were being sent: so, I tried a different approach, to store that data in MongoDB:  
  
Code: To read data from Kafka and write into MongoDB  
  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import \*  
from pyspark.sql.types import StringType  
from pyspark.sql import DataFrame  
from pymongo import MongoClient  
  
# Kafka configurations  
kafka\_bootstrap\_servers = 'kafka:9092'  
kafka\_topic = 'ecommerce-data'  
  
# MongoDB configurations  
mongo\_uri = "mongodb://mongo:27017/"  
mongo\_db = "kafka\_data"  
mongo\_collection = "ecommerce\_orders"  
  
  
# Create a Spark session  
spark = SparkSession.builder \  
 .appName("KafkaMongoConsumer") \  
 .config("spark.sql.streaming.schemaInference", "true") \  
 .getOrCreate()  
  
# Read data from Kafka  
df = spark.readStream.format("kafka") \  
 .option("kafka.bootstrap.servers", kafka\_bootstrap\_servers) \  
 .option("subscribe", kafka\_topic) \  
 .load()  
  
# Kafka message contains value in binary format, so we need to convert it to string  
df = df.selectExpr("CAST(value AS STRING)")  
  
# Parse the JSON data from Kafka message (assuming the data is in JSON format)  
df = df.select(from\_json(col("value"),  
 "order\_id STRING, product STRING, price STRING, quantity STRING, total\_price STRING, order\_date STRING").alias(  
 "data"))  
df = df.select("data.\*")  
  
  
# Write the data to MongoDB  
def write\_to\_mongo(batch\_df: DataFrame, batch\_id: int):  
 # Convert Spark DataFrame to Pandas DataFrame  
 pandas\_df = batch\_df.toPandas()  
  
 # Create MongoDB client and database  
 client = MongoClient(mongo\_uri)  
 db = client[mongo\_db]  
 collection = db[mongo\_collection]  
  
 # Convert DataFrame to dictionary format and insert to MongoDB  
 collection.insert\_many(pandas\_df.to\_dict(orient='records'))  
 print(f"Inserted {len(pandas\_df)} records into MongoDB")  
  
  
# Start the streaming query and write data to MongoDB  
query = df.writeStream \  
 .foreachBatch(write\_to\_mongo) \  
 .outputMode("append") \  
 .start()  
  
query.awaitTermination()

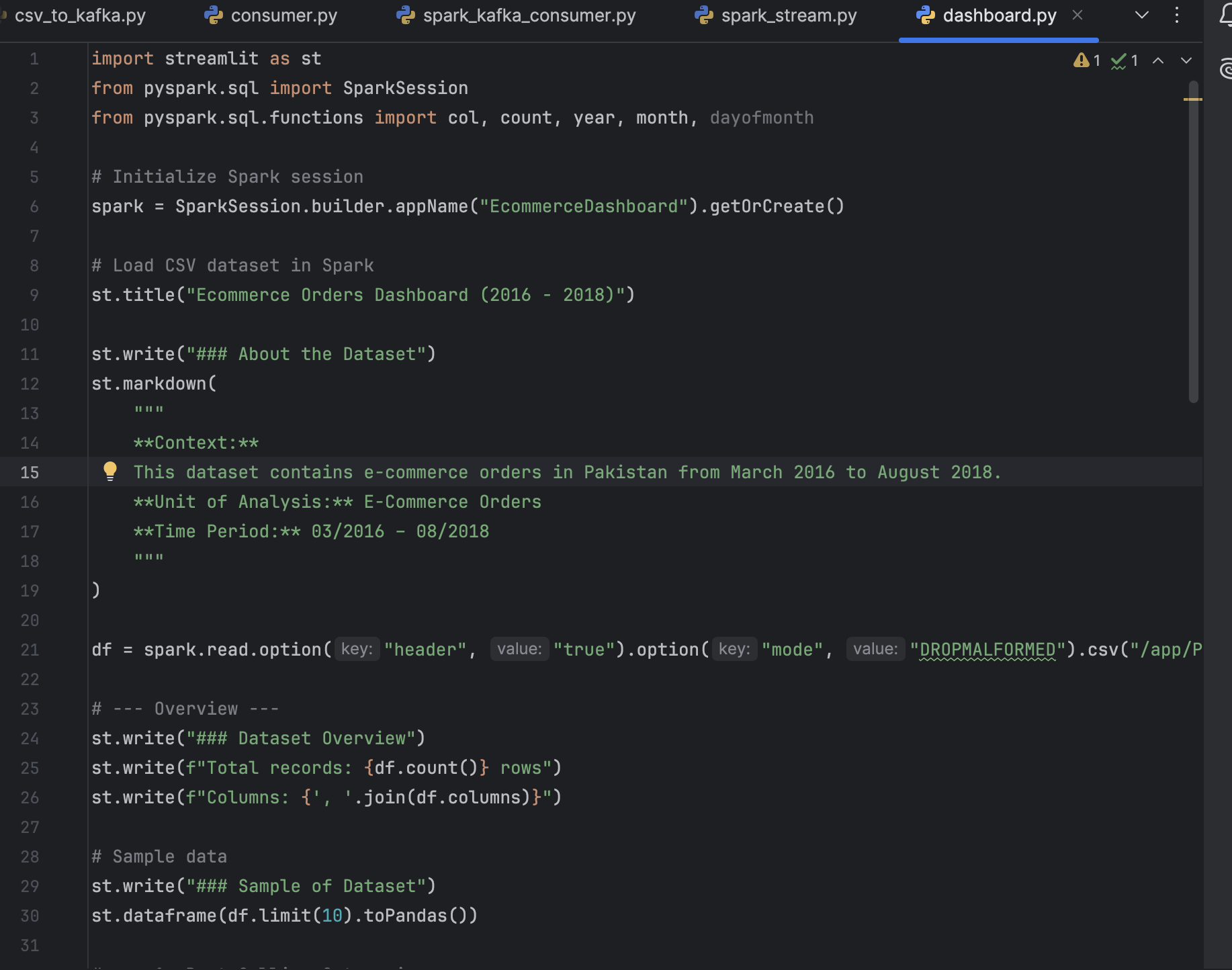
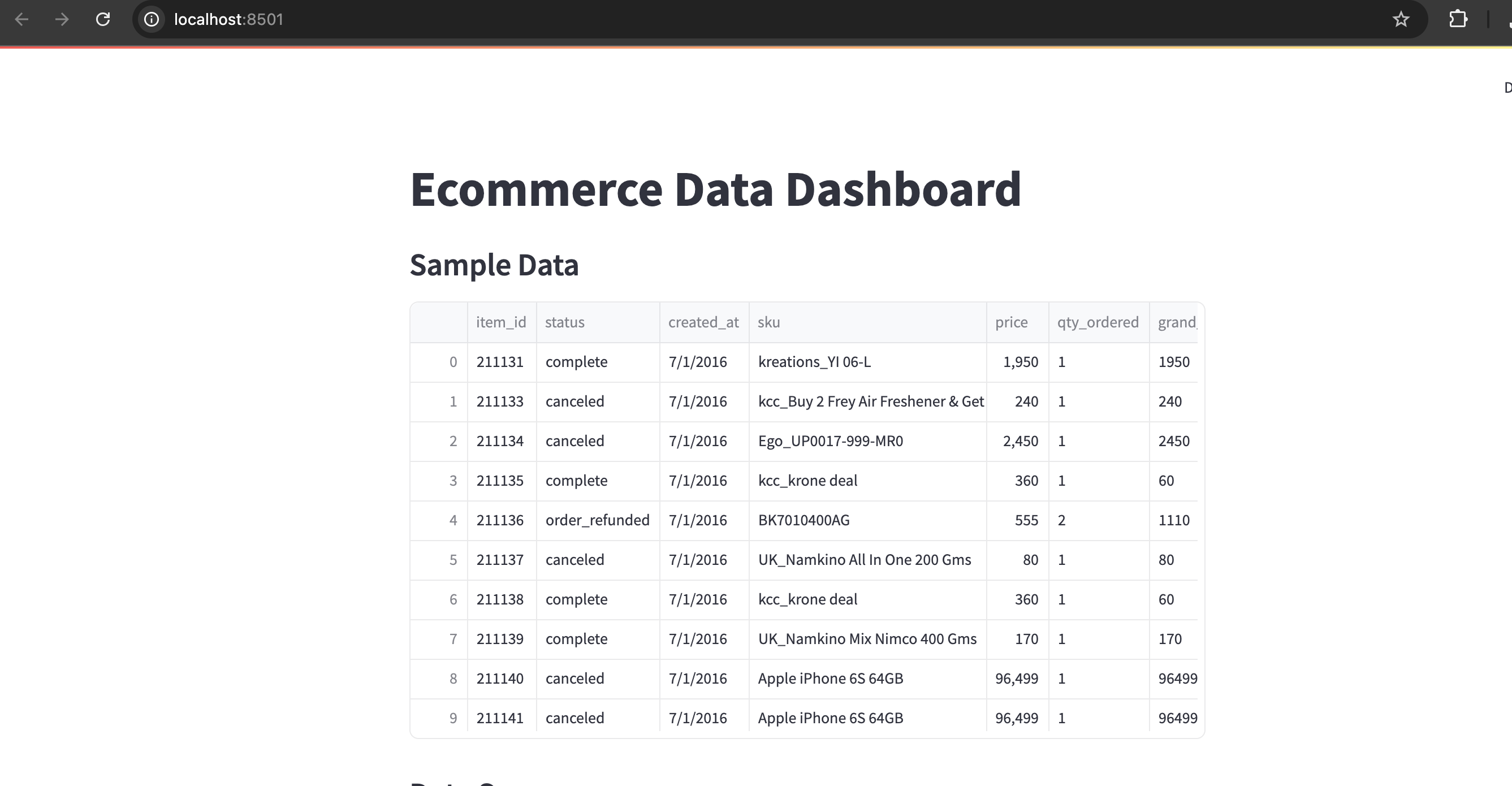


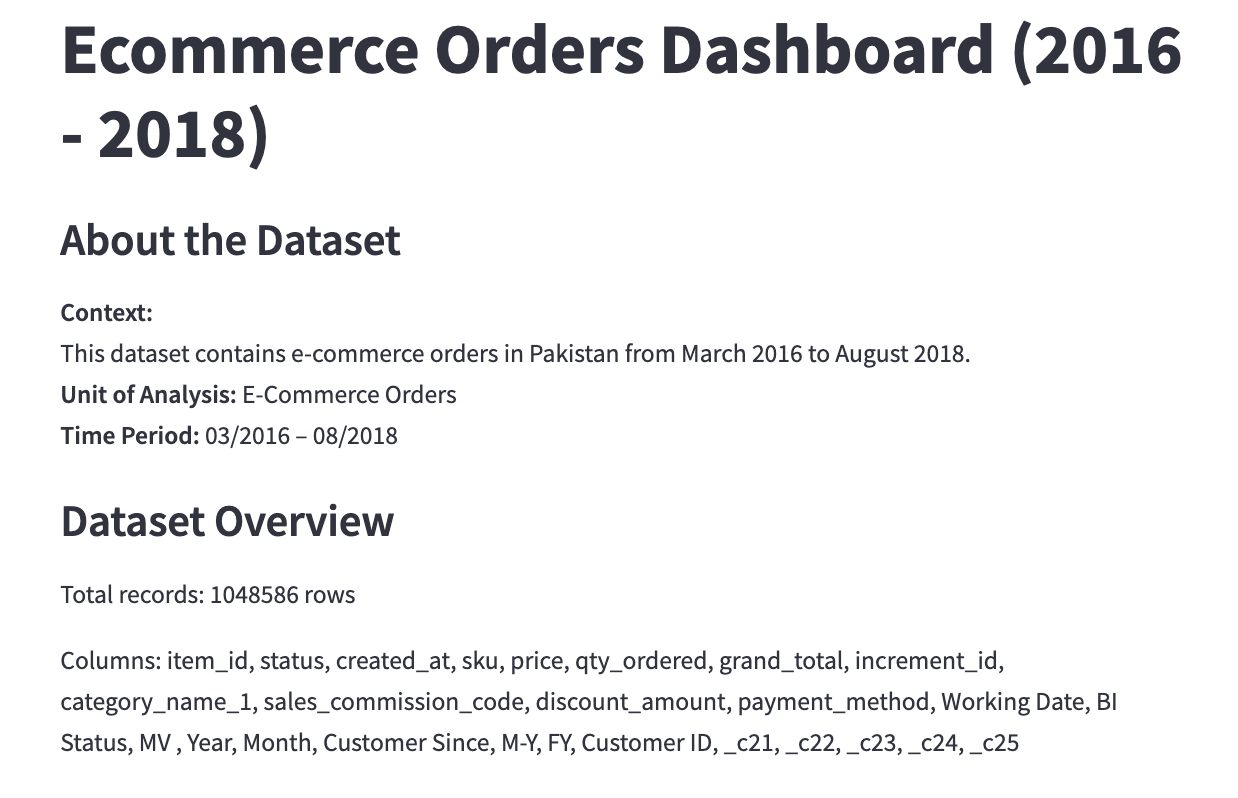
**We are now building a Docker container that packages a PySpark-based data processing application (spark\_stream.py) along with Streamlit for visualization and Kafka-python for data streaming. The Spark application reads the large e-commerce dataset, performs distributed data analytics, and Streamlit can display real-time insights via a web interface. The goal is to create a seamless and scalable data pipeline for big data analytics and visualization using containerized tools.**





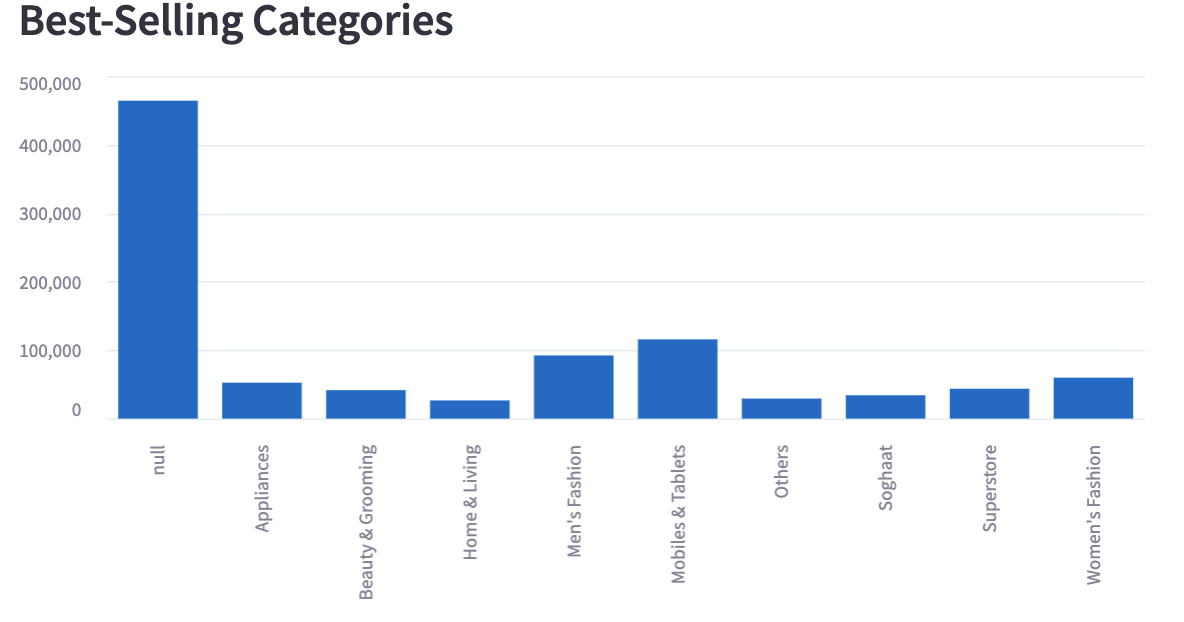




**Exploratory Data Analysis and BI:**  


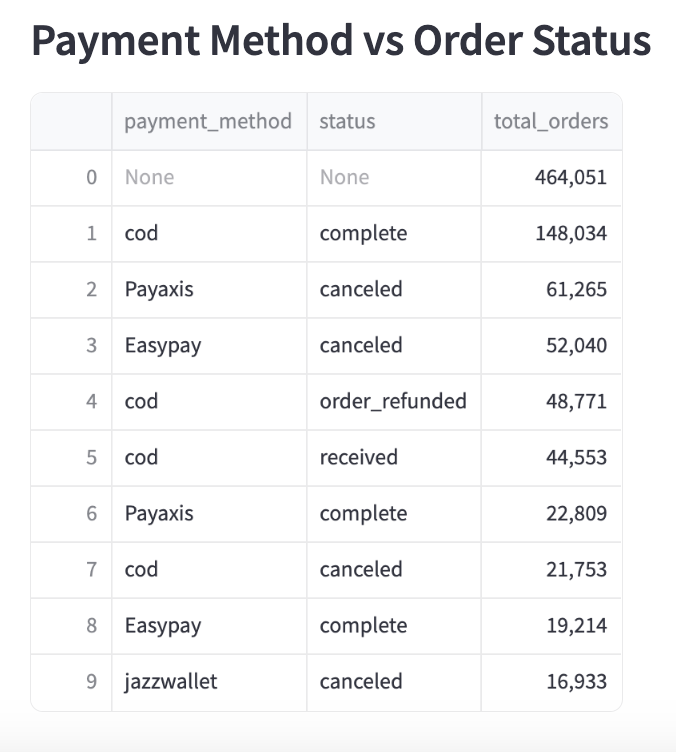
**Best-Selling Categories:**

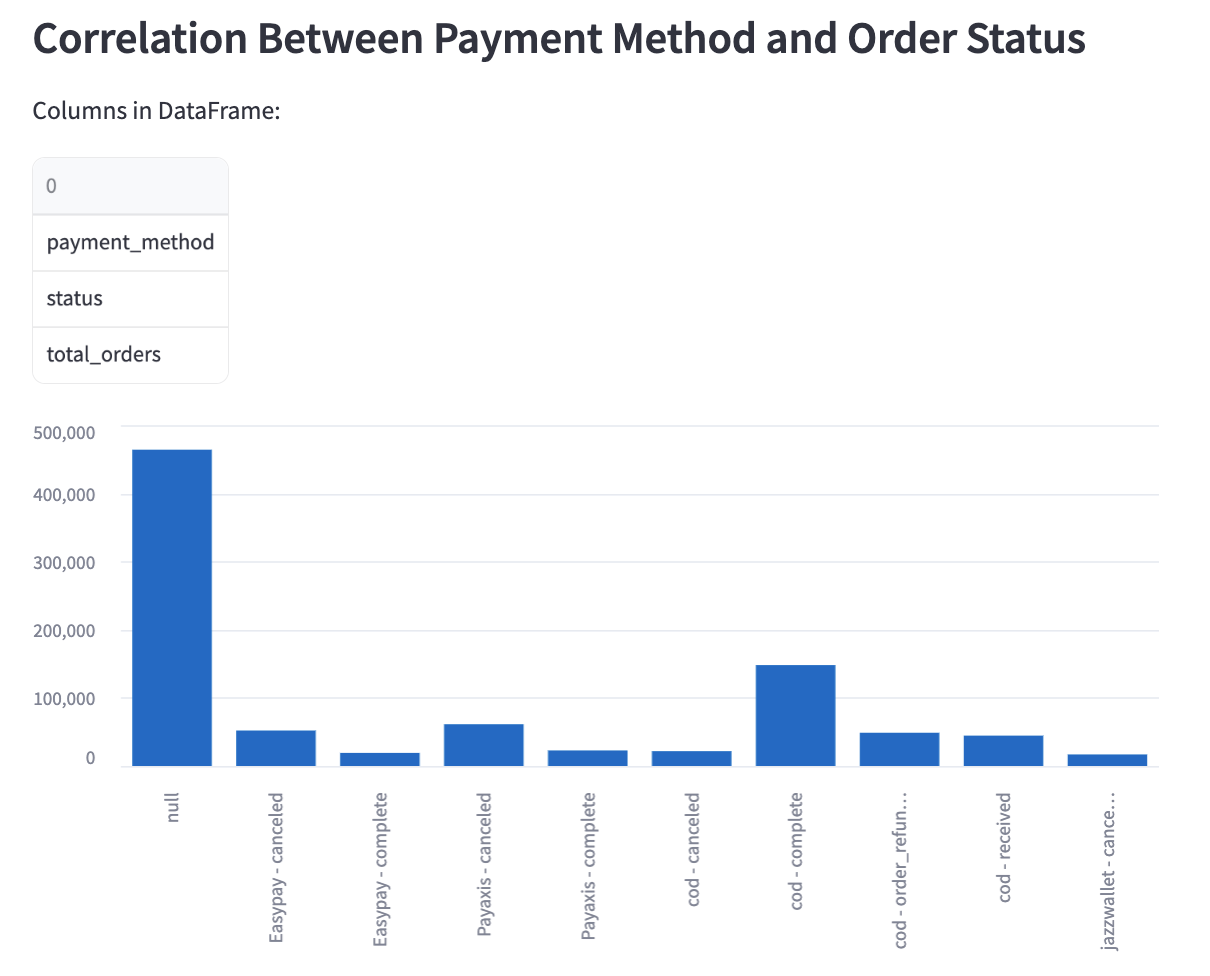
Bar chart of the top 10 product categories based on the number of orders.



**Payment Method vs Order Status:**

Shows the frequency of different payment methods and their corresponding order status (completed, canceled, etc.).



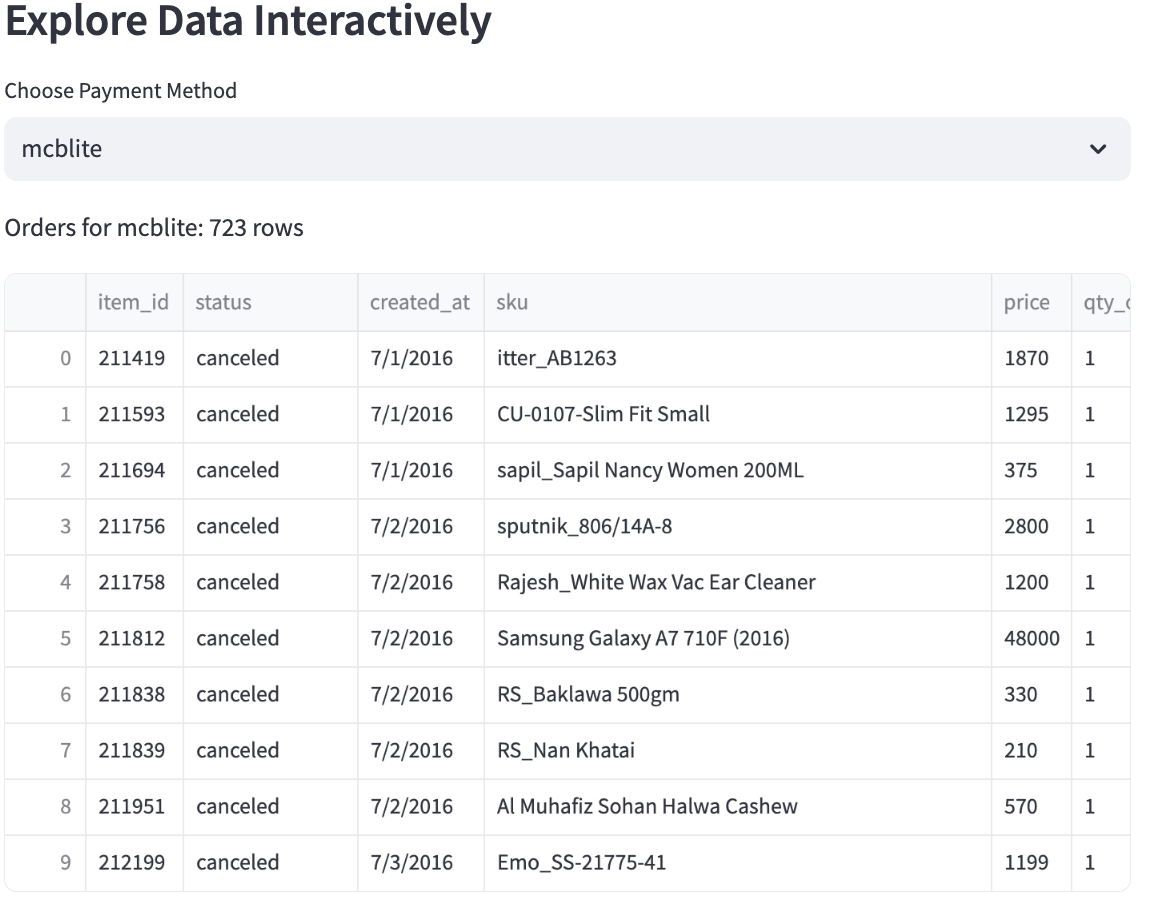


**Hidden Insights:**

Adds interpretation and domain insights.

**Interactive Filter:**

Allows users to filter orders by specific payment methods and display the corresponding records. For example, i explore the payment option mcblite and view its details:



Then I change the payment method to: **internet banking**

Using Streamlit and PySpark this gives us an advantage to make real time data exploration:  
  
  
