Name: Omkar Rajesh Desai
Email: omkarbkishor@gmail.com

Real-Time Advertisement Data Aggregation Documentation

Tools Used:

1. Python3 – PyCharm Community Edition
2. Docker
3. Kafka
4. Zookeeper
5. Spark
6. Jupyter
7. Cassandra

Files Attached:

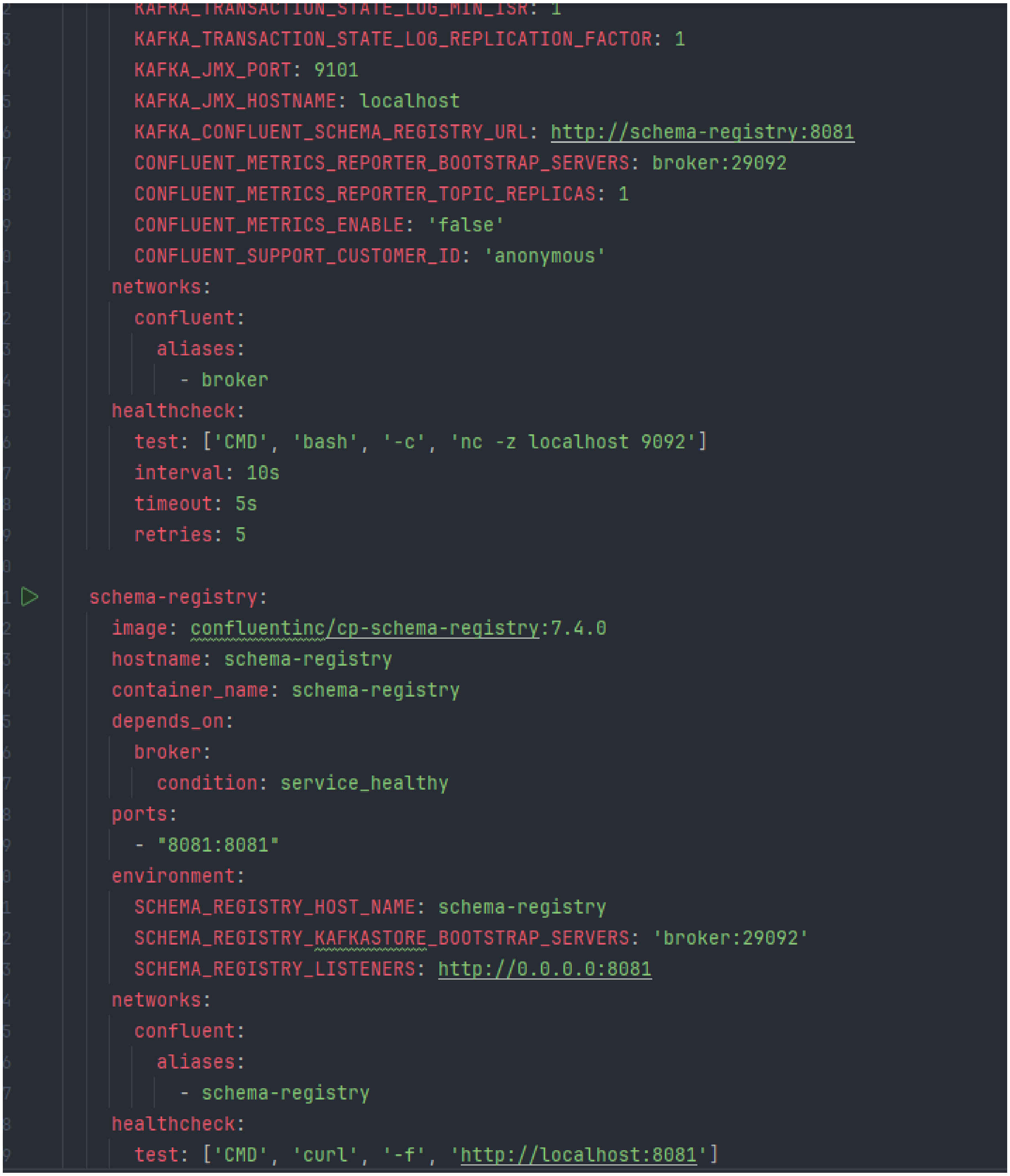
1. Documentation.pdf – This file
2. producer.py – file for producing ads_data to kafka topic(i.e ads_data).
3. Docker-compose.yml – For setting up the environment for Kafka, zookeeper, spark, Cassandra.
4. streaming_code.py – which contain streaming code.
5. schema.avsc – which contain Avro-schema used.

providing **GitHub** link for more information

Process and File Descriptions:

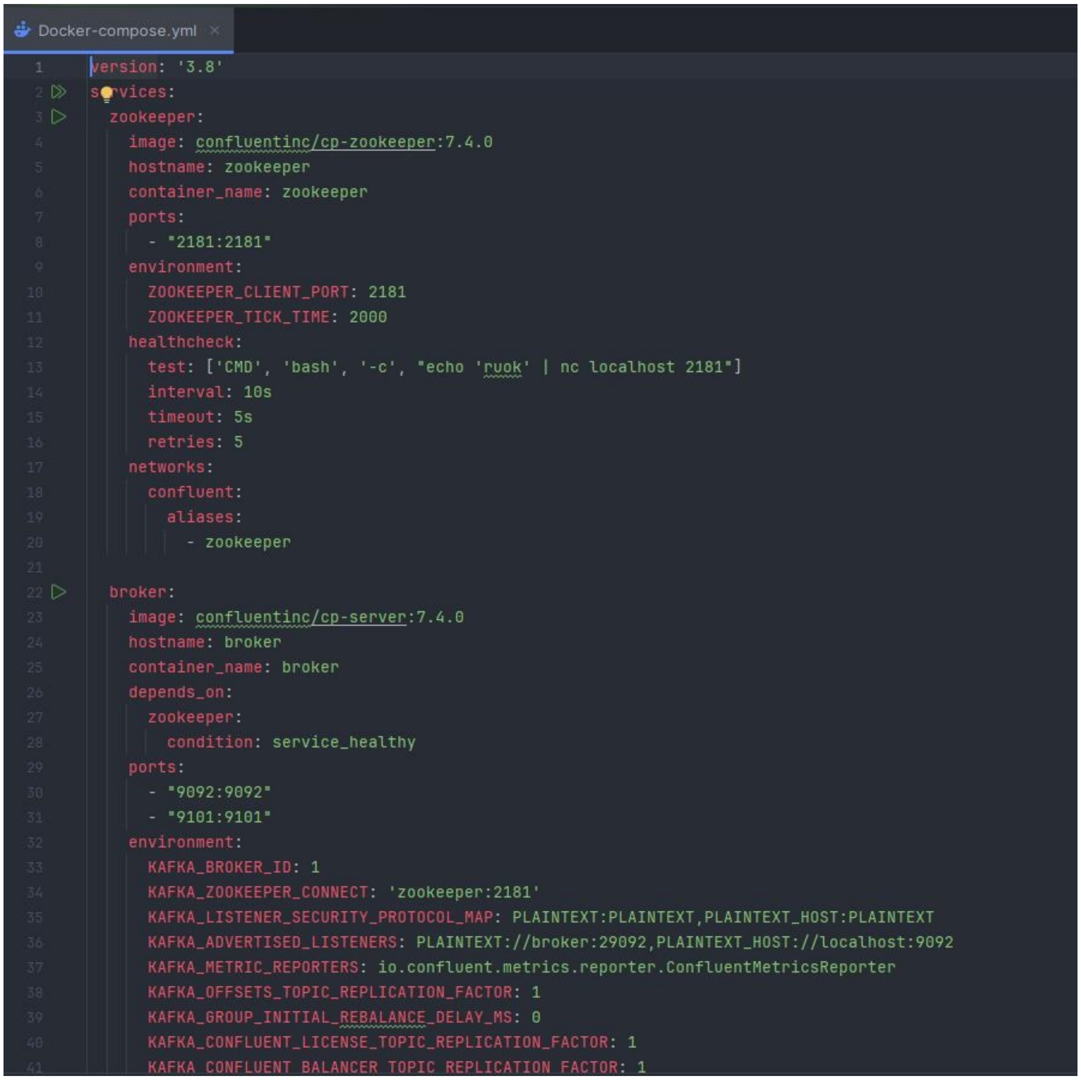
Step 1:

I first created a docker-compose file for setting up the environment for project by considering all the required services for the project and make sure all the required ports for each service are exposed and make sure to keep all the containers connected to the same network. I also created the service for control center and schema registry control center which will provide web UI for the schema registry for our Avro schema which is depend on schema registry. Schema registry help us to register our avro_schema.



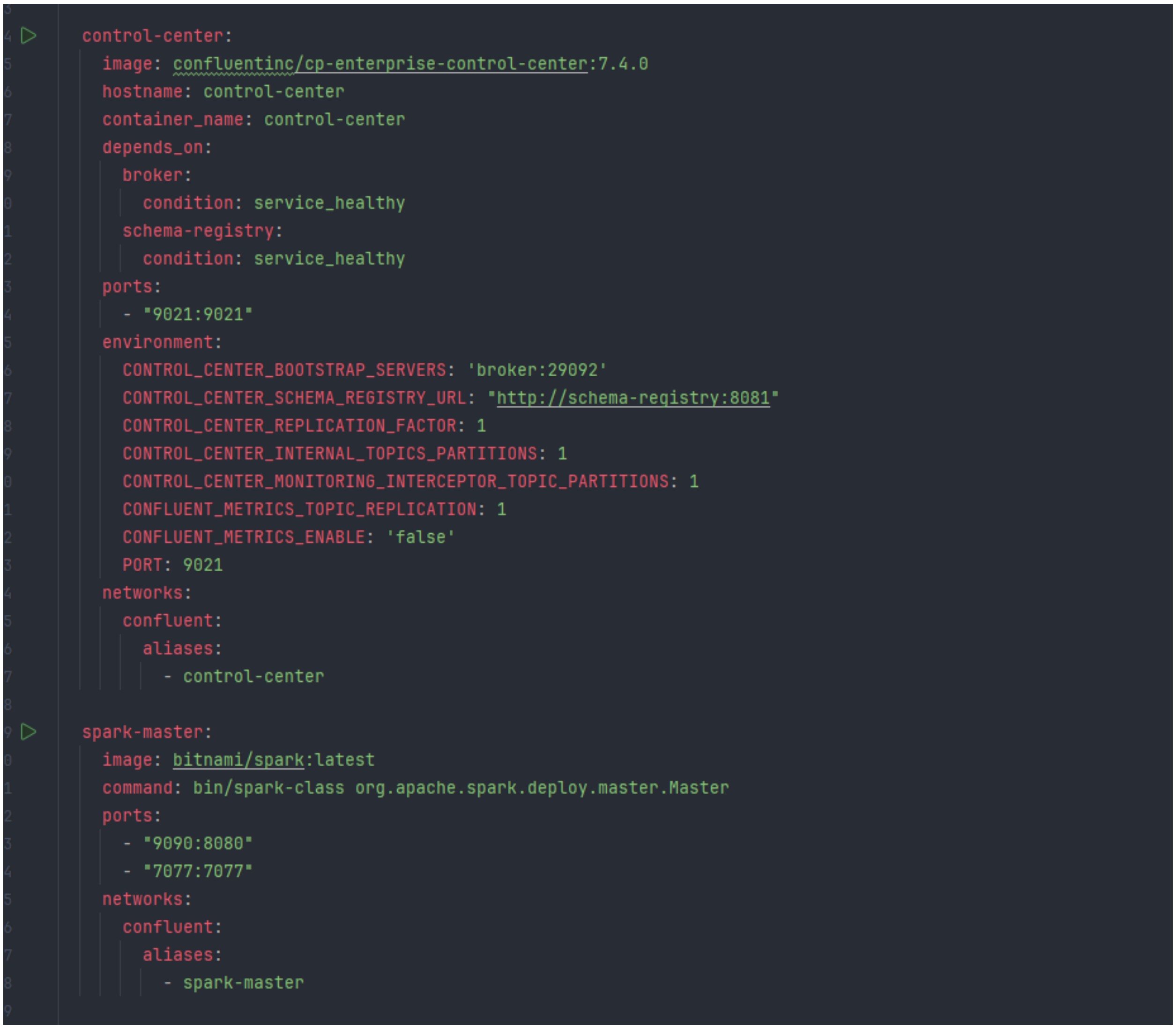
```
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost
      KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry:8081
      CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092
      CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
      CONFLUENT_METRICS_ENABLE: 'false'
      CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'
    networks:
      confluent:
        aliases:
          - broker
    healthcheck:
      test: ['CMD', 'bash', '-c', 'nc -z localhost 9092']
      interval: 10s
      timeout: 5s
      retries: 5

  schema-registry:
    image: confluentinc/cp-schema-registry:7.4.0
    hostname: schema-registry
    container_name: schema-registry
    depends_on:
      broker:
        condition: service_healthy
    ports:
      - "8081:8081"
    environment:
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: 'broker:29092'
      SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081
    networks:
      confluent:
        aliases:
          - schema-registry
    healthcheck:
      test: ['CMD', 'curl', '-f', 'http://localhost:8081']
```

Fig.: Docker-compose File



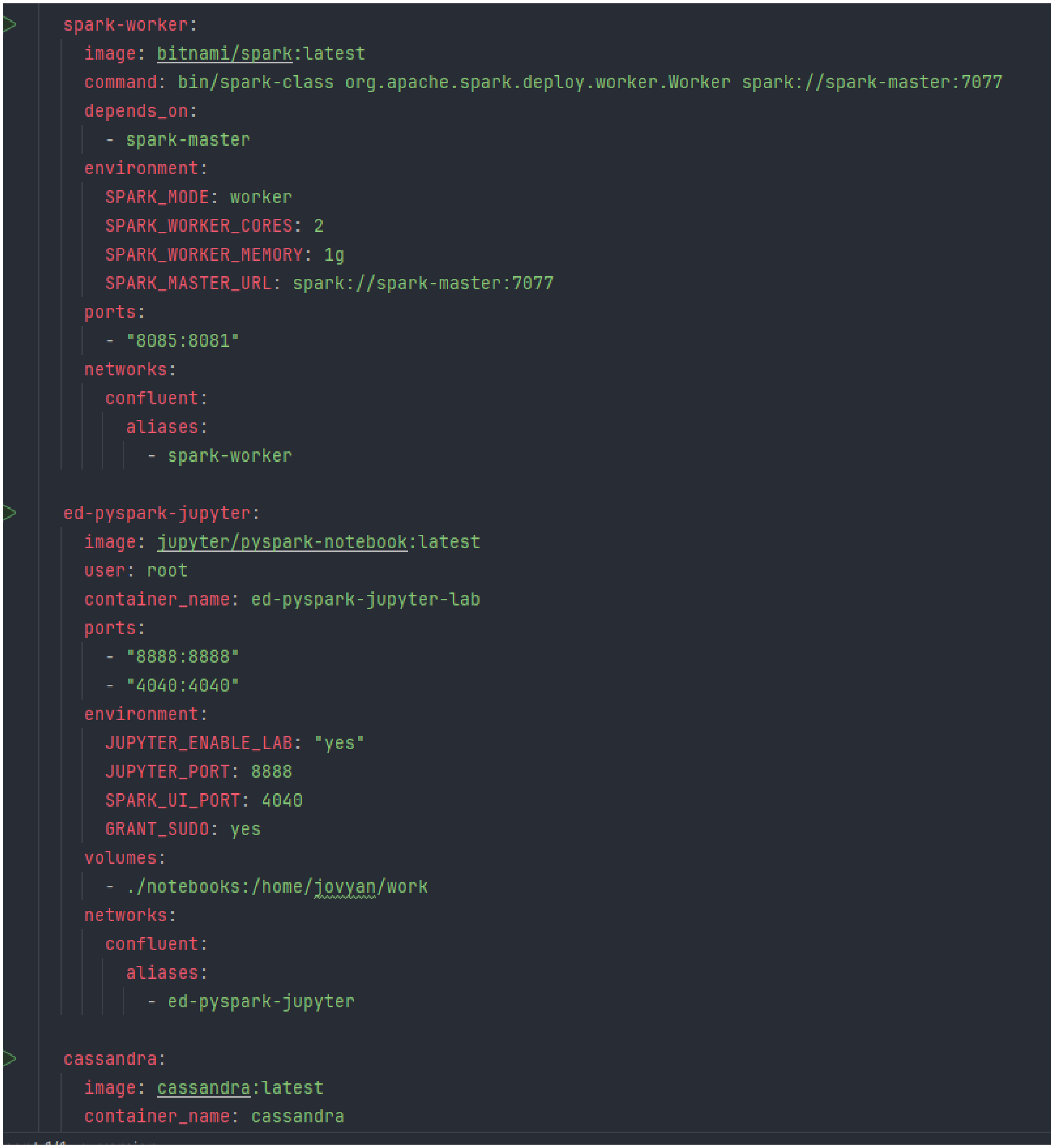
```yaml
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    healthcheck:
      test: ['CMD', 'bash', '-c', "echo 'ruok' | nc localhost 2181"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      confluent:
        aliases:
          - zookeeper

  broker:
    image: confluentinc/cp-server:7.4.0
    hostname: broker
    container_name: broker
    depends_on:
      zookeeper:
        condition: service_healthy
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.ConfluentMetricsReporter
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_CONFLUENT_BALANCER_TOPIC_REPLICATION_FACTOR: 1
```

Fig.: Docker-compose File



```yaml
  control-center:
    image: confluentinc/cp-enterprise-control-center:7.4.0
    hostname: control-center
    container_name: control-center
    depends_on:
      broker:
        condition: service_healthy
      schema-registry:
        condition: service_healthy
    ports:
      - "9021:9021"
    environment:
      CONTROL_CENTER_BOOTSTRAP_SERVERS: 'broker:29092'
      CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
      CONTROL_CENTER_REPLICATION_FACTOR: 1
      CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
      CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
      CONFLUENT_METRICS_TOPIC_REPLICATION: 1
      CONFLUENT_METRICS_ENABLE: 'false'
      PORT: 9021
    networks:
      confluent:
        aliases:
          - control-center

  spark-master:
    image: bitnami/spark:latest
    command: bin/spark-class org.apache.spark.deploy.master.Master
    ports:
      - "9090:8080"
      - "7077:7077"
    networks:
      confluent:
        aliases:
          - spark-master
```

Fig.: Docker-compose File

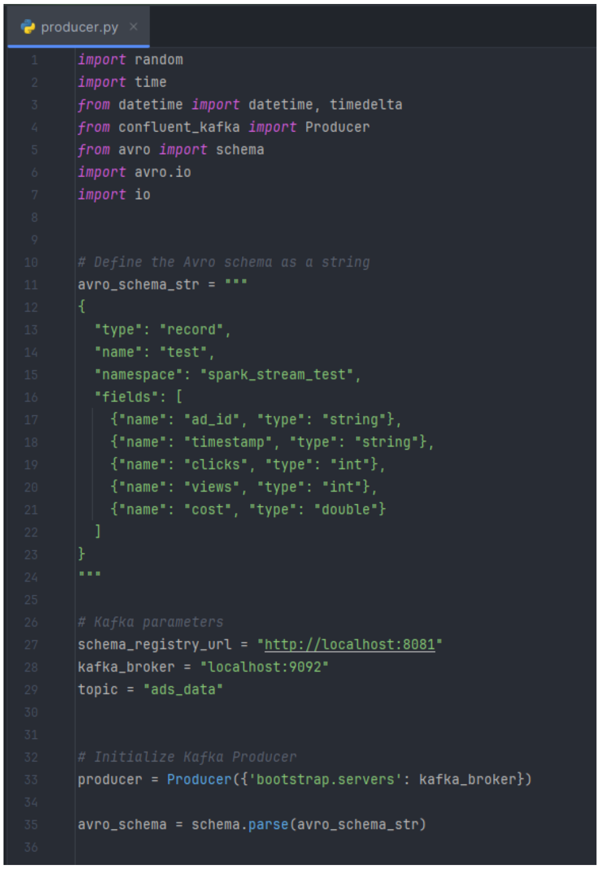


```
  spark-worker:
    image: bitnami/spark:latest
    command: bin/spark-class org.apache.spark.deploy.worker.Worker spark://spark-master:7077
    depends_on:
      - spark-master
    environment:
      SPARK_MODE: worker
      SPARK_WORKER_CORES: 2
      SPARK_WORKER_MEMORY: 1g
      SPARK_MASTER_URL: spark://spark-master:7077
    ports:
      - "8085:8081"
    networks:
      confluent:
        aliases:
          - spark-worker

  ed-pyspark-jupyter:
    image: jupyter/pyspark-notebook:latest
    user: root
    container_name: ed-pyspark-jupyter-lab
    ports:
      - "8888:8888"
      - "4040:4040"
    environment:
      JUPYTER_ENABLE_LAB: "yes"
      JUPYTER_PORT: 8888
      SPARK_UI_PORT: 4040
      GRANT_SUDO: yes
    volumes:
      - ./notebooks:/home/jovyan/work
    networks:
      confluent:
        aliases:
          - ed-pyspark-jupyter

  cassandra:
    image: cassandra:latest
    container_name: cassandra
```

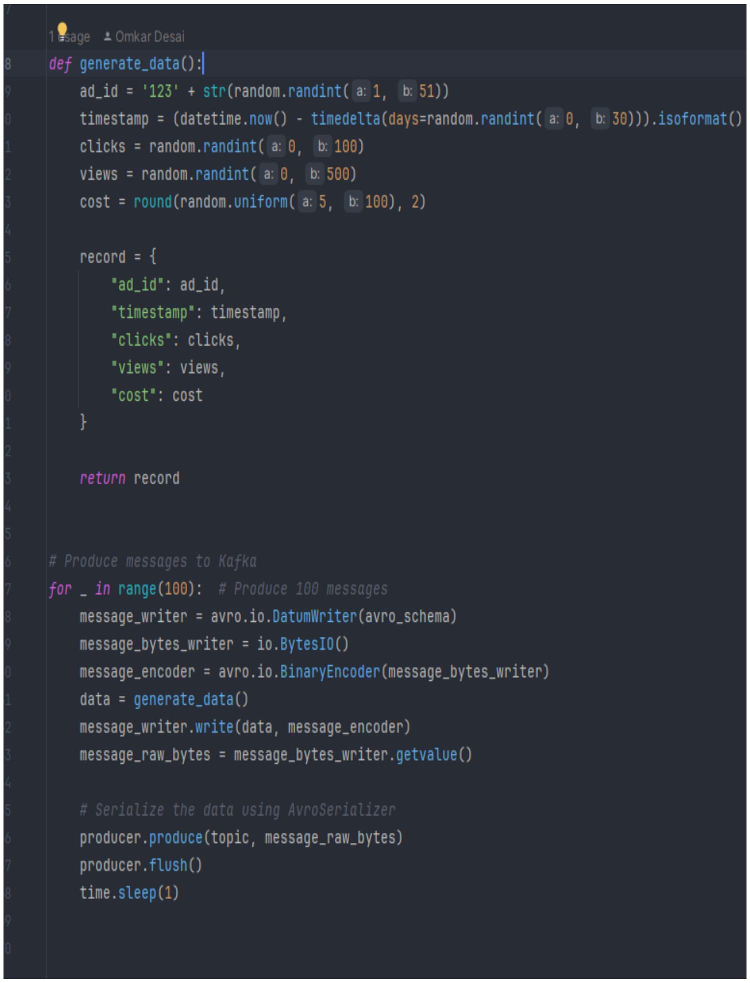
Fig.: Docker-compose File

Step 2:

After setting up the containers checked all the contianers are up and running then create the **ads_data** topic with 2 partitions on kafka using control center web UI after that create Avro schema file for schema registry then write logic to produce messages on kafka topic.



```python
import random
import time
from datetime import datetime, timedelta
from confluent_kafka import Producer
from avro import schema
import avro.io
import io


# Define the Avro schema as a string
avro_schema_str = """
{
  "type": "record",
  "name": "test",
  "namespace": "spark_stream_test",
  "fields": [
    {"name": "ad_id", "type": "string"},
    {"name": "timestamp", "type": "string"},
    {"name": "clicks", "type": "int"},
    {"name": "views", "type": "int"},
    {"name": "cost", "type": "double"}
  ]
}
"""


# Kafka parameters
schema_registry_url = "http://localhost:8081"
kafka_broker = "localhost:9092"
topic = "ads_data"


# Initialize Kafka Producer
producer = Producer({'bootstrap.servers': kafka_broker})

avro_schema = schema.parse(avro_schema_str)

```

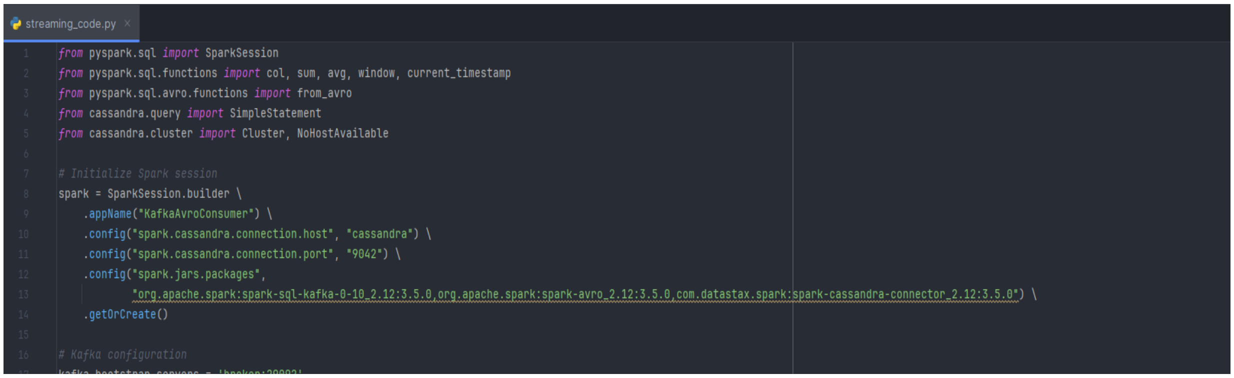


```python
def generate_data():
    ad_id = '123' + str(random.randint(1, 51))
    timestamp = (datetime.now() - timedelta(days=random.randint(0, 30))).isoformat()
    clicks = random.randint(0, 100)
    views = random.randint(0, 500)
    cost = round(random.uniform(5, 100), 2)

    record = {
        "ad_id": ad_id,
        "timestamp": timestamp,
        "clicks": clicks,
        "views": views,
        "cost": cost
    }

    return record


# Produce messages to Kafka
for _ in range(100):  # Produce 100 messages
    message_writer = avro.io.DatumWriter(avro_schema)
    message_bytes_writer = io.BytesIO()
    message_encoder = avro.io.BinaryEncoder(message_bytes_writer)
    data = generate_data()
    message_writer.write(data, message_encoder)
    message_raw_bytes = message_bytes_writer.getvalue()

    # Serialize the data using AvroSerializer
    producer.produce(topic, message_raw_bytes)
    producer.flush()
    time.sleep(1)
```

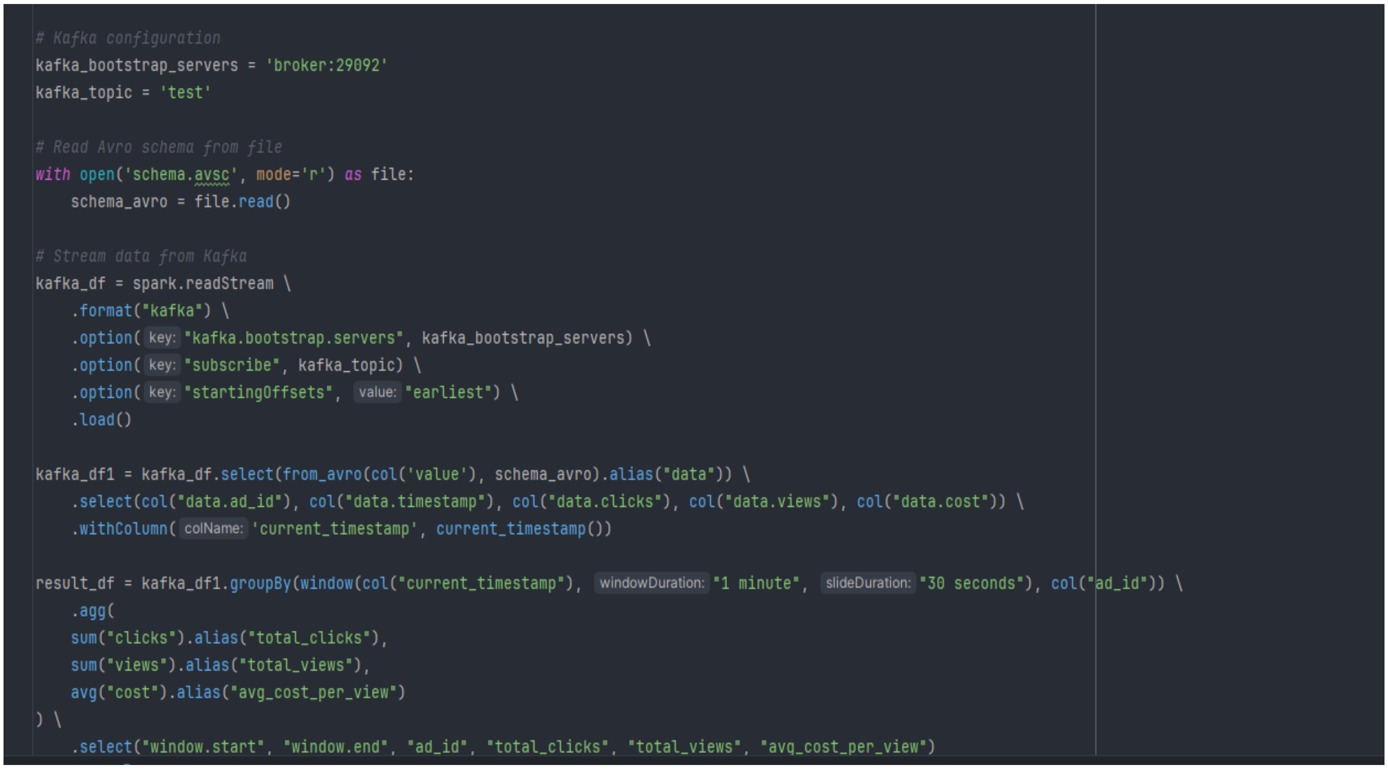
Fig.: Producer Code

Step 3:

I then went ahead and performed the necessary transformations/queries on produce data using spark structured streaming import all the necessary modules by first install them in container using pip command after that mention all the required jar files in config of spark session such as spark-sql-kafka, spark-avro, spark-cassandra then in code done window_based aggregation with window duration of 1 minute and sliding interval of 30 seconds. Then create function for processing each row's data with existing data in cassandra table then used forEachBatch function to apply this function for each batch and for deserializing messages I used from_avro function.

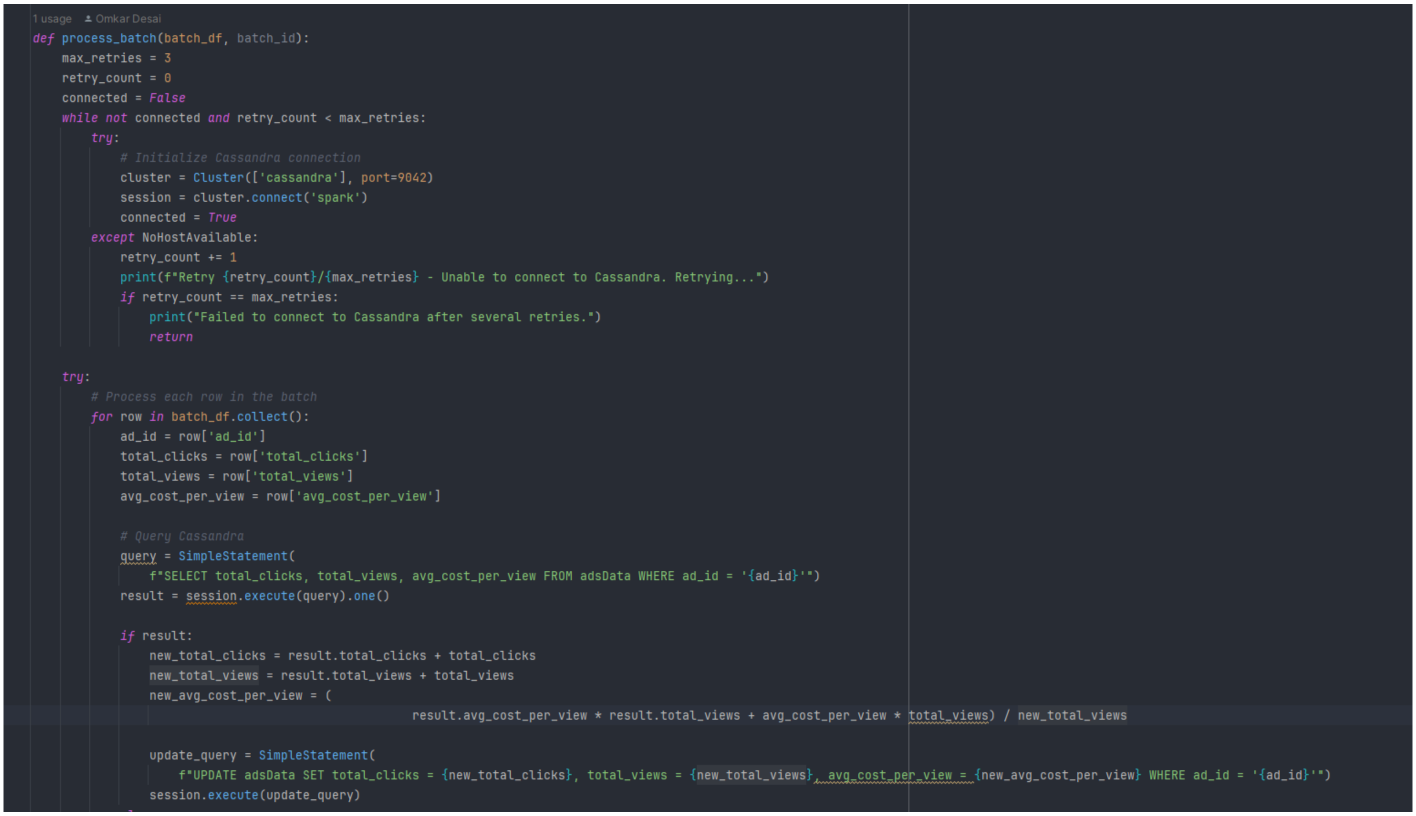


```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum, avg, window, current_timestamp
from pyspark.sql.avro.functions import from_avro
from cassandra.query import SimpleStatement
from cassandra.cluster import Cluster, NoHostAvailable

# Initialize Spark session
spark = SparkSession.builder \
    .appName("KafkaAvroConsumer") \
    .config("spark.cassandra.connection.host", "cassandra") \
    .config("spark.cassandra.connection.port", "9042") \
    .config("spark.jars.packages",
            "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,org.apache.spark:spark-avro_2.12:3.5.0,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0") \
    .getOrCreate()

# Kafka configuration
kafka_bootstrap_servers = 'broker:29092'
```



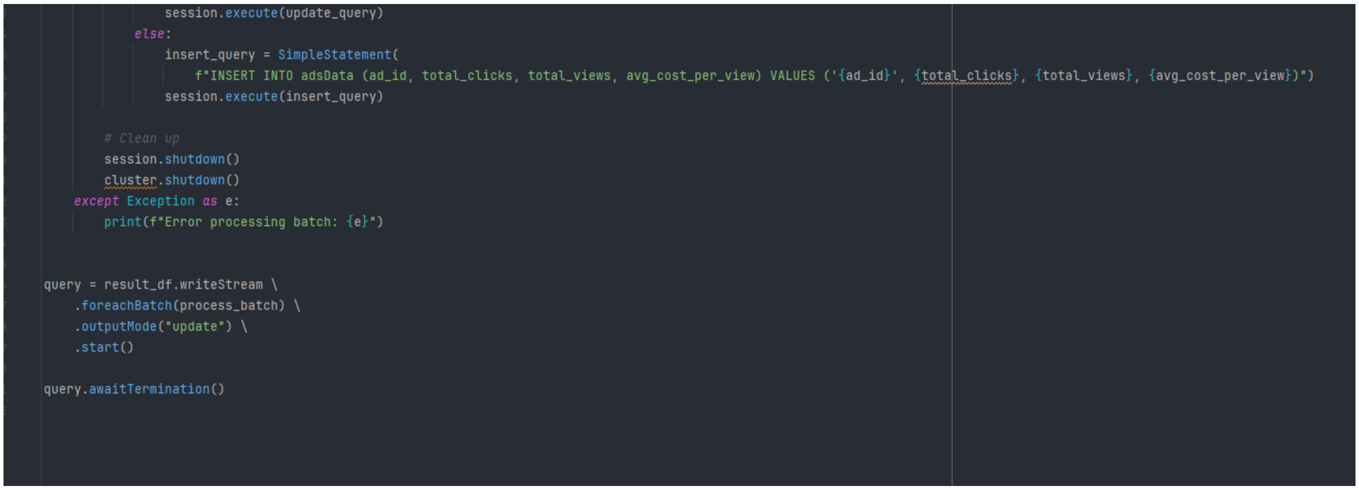
```
# Kafka configuration
kafka_bootstrap_servers = 'broker:29092'
kafka_topic = 'test'

# Read Avro schema from file
with open('schema.avsc', mode='r') as file:
    schema_avro = file.read()

# Stream data from Kafka
kafka_df = spark.readStream \
    .format("kafka") \
    .option( key: "kafka.bootstrap.servers", kafka_bootstrap_servers) \
    .option( key: "subscribe", kafka_topic) \
    .option( key: "startingOffsets",  value: "earliest") \
    .load()

kafka_df1 = kafka_df.select(from_avro(col('value'), schema_avro).alias("data")) \
    .select(col("data.ad_id"), col("data.timestamp"), col("data.clicks"), col("data.views"), col("data.cost")) \
    .withColumn( colName: 'current_timestamp', current_timestamp())

result_df = kafka_df1.groupBy(window(col("current_timestamp"),  windowDuration: "1 minute",  slideDuration: "30 seconds"), col("ad_id")) \
    .agg(
    sum("clicks").alias("total_clicks"),
    sum("views").alias("total_views"),
    avg("cost").alias("avg_cost_per_view")
) \
    .select("window.start", "window.end", "ad_id", "total_clicks", "total_views", "avg_cost_per_view")
```

Fig.: Streaming Code



```
1 usage  Omkar Desai
def process_batch(batch_df, batch_id):
    max_retries = 3
    retry_count = 0
    connected = False
    while not connected and retry_count < max_retries:
        try:
            # Initialize Cassandra connection
            cluster = Cluster(['cassandra'], port=9042)
            session = cluster.connect('spark')
            connected = True
        except NoHostAvailable:
            retry_count += 1
            print(f"Retry {retry_count}/{max_retries} - Unable to connect to Cassandra. Retrying...")
            if retry_count == max_retries:
                print("Failed to connect to Cassandra after several retries.")
                return

    try:
        # Process each row in the batch
        for row in batch_df.collect():
            ad_id = row['ad_id']
            total_clicks = row['total_clicks']
            total_views = row['total_views']
            avg_cost_per_view = row['avg_cost_per_view']

            # Query Cassandra
            query = SimpleStatement(
                f"SELECT total_clicks, total_views, avg_cost_per_view FROM adsData WHERE ad_id = '{ad_id}'")
            result = session.execute(query).one()

            if result:
                new_total_clicks = result.total_clicks + total_clicks
                new_total_views = result.total_views + total_views
                new_avg_cost_per_view = (
                                                    result.avg_cost_per_view * result.total_views + avg_cost_per_view * total_views) / new_total_views

                update_query = SimpleStatement(
                    f"UPDATE adsData SET total_clicks = {new_total_clicks}, total_views = {new_total_views}, avg_cost_per_view = {new_avg_cost_per_view} WHERE ad_id = '{ad_id}'")
                session.execute(update_query)
```

Fig.: Streaming Code

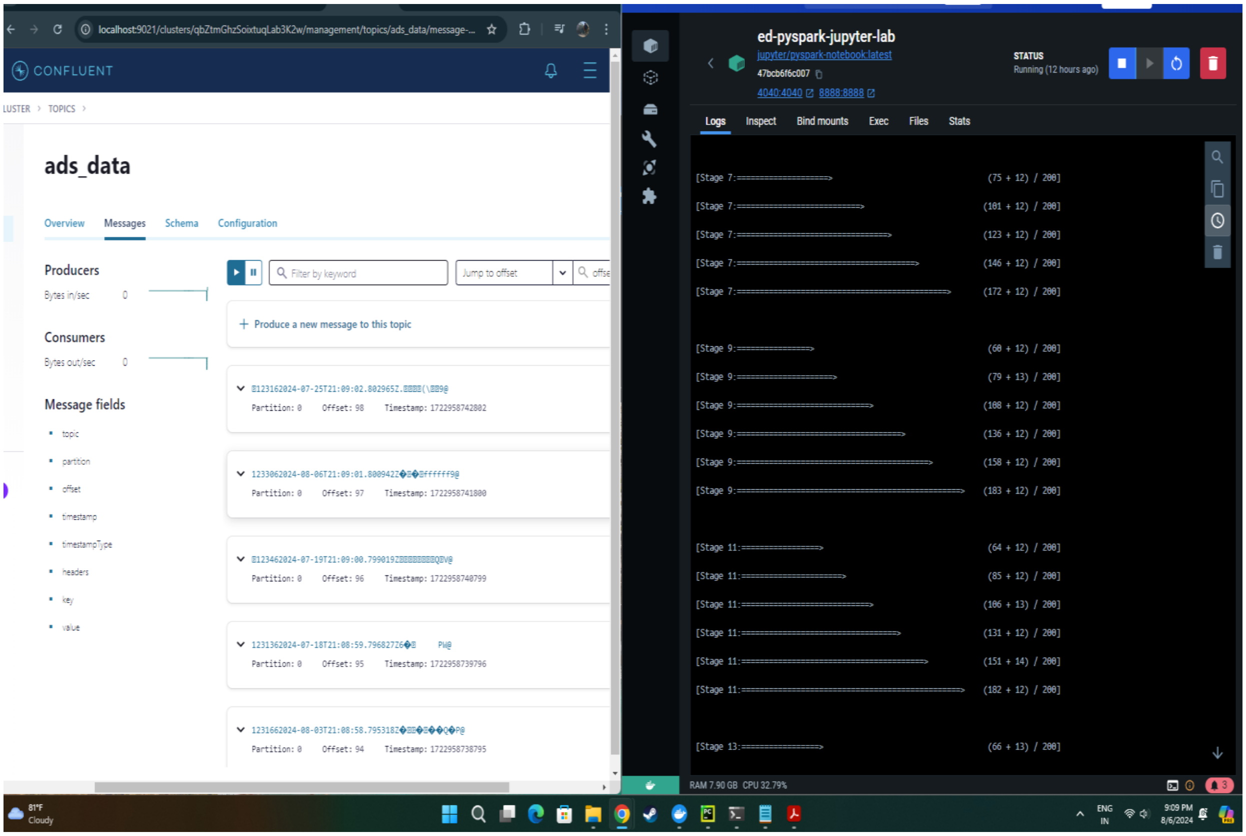


```
                session.execute(update_query)
            else:
                insert_query = SimpleStatement(
                    f"INSERT INTO adsData (ad_id, total_clicks, total_views, avg_cost_per_view) VALUES ('{ad_id}', {total_clicks}, {total_views}, {avg_cost_per_view})")
                session.execute(insert_query)

        # Clean up
        session.shutdown()
        cluster.shutdown()
    except Exception as e:
        print(f"Error processing batch: {e}")


query = result_df.writeStream \
    .foreachBatch(process_batch) \
    .outputMode("update") \
    .start()

query.awaitTermination()
```

Fig.: Streaming Code

Step 4:

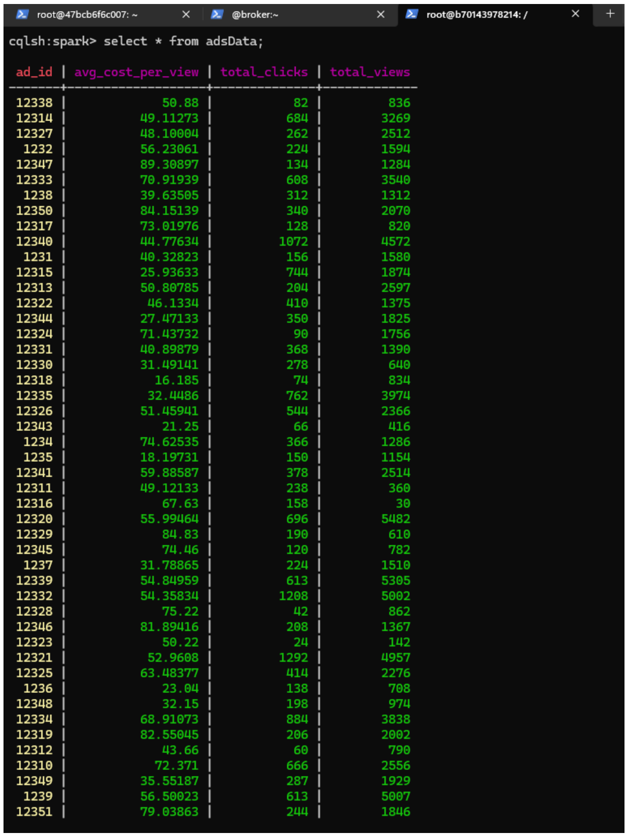
After that using CQL I created keyspace and then create table in which I will be writing data make sure to match the schema of the table with the datatypes. after running the job I checked table for the entry of data and also checked the topic which I was producing messages simultaneously checked the log of the container for running spark application.



```
▯123162024-07-25T21:09:02.802965Z.▯▯▯▯(\▯▯9@
Partition: 0    Offset: 98    Timestamp: 1722958742802

1233062024-08-06T21:09:01.800942Z�▯�▯ffffff9@
Partition: 0    Offset: 97    Timestamp: 1722958741800

▯123462024-07-19T21:09:00.799019Z▯▯▯▯▯▯▯▯Q▯V@
Partition: 0    Offset: 96    Timestamp: 1722958740799

1231362024-07-18T21:08:59.796827Z6�▯      PW@
Partition: 0    Offset: 95    Timestamp: 1722958739796

1231662024-08-03T21:08:58.795318Z�▯▯�▯��Q�P@
Partition: 0    Offset: 94    Timestamp: 1722958738795
```

```
[Stage 7:=================>                                    (75 + 12) / 200]
[Stage 7:=======================>                             (101 + 12) / 200]
[Stage 7:============================>                        (123 + 12) / 200]
[Stage 7:=================================>                   (146 + 12) / 200]
[Stage 7:=======================================>             (172 + 12) / 200]

[Stage 9:==============>                                       (60 + 12) / 200]
[Stage 9:==================>                                   (79 + 13) / 200]
[Stage 9:=========================>                           (108 + 12) / 200]
[Stage 9:===============================>                     (136 + 12) / 200]
[Stage 9:====================================>                (158 + 12) / 200]
[Stage 9:==========================================>          (183 + 12) / 200]

[Stage 11:===============>                                     (64 + 12) / 200]
[Stage 11:===================>                                 (85 + 12) / 200]
[Stage 11:========================>                           (106 + 13) / 200]
[Stage 11:=============================>                      (131 + 12) / 200]
[Stage 11:==================================>                 (151 + 14) / 200]
[Stage 11:=========================================>          (182 + 12) / 200]

[Stage 13:===============>                                     (66 + 13) / 200]
```

Fig.: Message producing in bytes format on confluent with running spark application container.



```
cqlsh:spark> select * from adsData;

 ad_id | avg_cost_per_view | total_clicks | total_views
-------+-------------------+--------------+-------------
 12338 |             50.88 |           82 |         836
 12314 |          49.11273 |          684 |        3269
 12327 |          48.10004 |          262 |        2512
  1232 |          56.23061 |          224 |        1594
 12347 |          89.30897 |          134 |        1284
 12333 |          70.91939 |          608 |        3540
  1238 |          39.63505 |          312 |        1312
 12350 |          84.15139 |          340 |        2070
 12317 |          73.01976 |          128 |         820
 12340 |          44.77634 |         1072 |        4572
  1231 |          40.32823 |          156 |        1580
 12315 |          25.93633 |          744 |        1874
 12313 |          50.80785 |          204 |        2597
 12322 |           46.1334 |          410 |        1375
 12344 |          27.47133 |          350 |        1825
 12324 |          71.43732 |           90 |        1756
 12331 |          40.89879 |          368 |        1390
 12330 |          31.49141 |          278 |         640
 12318 |            16.185 |           74 |         834
 12335 |           32.4486 |          762 |        3974
 12326 |          51.45941 |          544 |        2366
 12343 |             21.25 |           66 |         416
  1234 |          74.62535 |          366 |        1286
  1235 |          18.19731 |          150 |        1154
 12341 |          59.88587 |          378 |        2514
 12311 |          49.12133 |          238 |         360
 12316 |             67.63 |          158 |          30
 12320 |          55.99464 |          696 |        5482
 12329 |             84.83 |          190 |         610
 12345 |             74.46 |          120 |         782
  1237 |          31.78865 |          224 |        1510
 12339 |          54.84959 |          613 |        5305
 12332 |          54.35834 |         1208 |        5002
 12328 |             75.22 |           42 |         862
 12346 |          81.89416 |          208 |        1367
 12323 |             50.22 |           24 |         142
 12321 |           52.9608 |         1292 |        4957
 12325 |          63.48377 |          414 |        2276
  1236 |             23.04 |          138 |         708
 12348 |             32.15 |          198 |         974
 12334 |          68.91073 |          884 |        3838
 12319 |          82.55045 |          206 |        2002
 12312 |             43.66 |           60 |         790
 12310 |            72.371 |          666 |        2556
 12349 |          35.55187 |          287 |        1929
  1239 |          56.50023 |          613 |        5007
 12351 |          79.03863 |          244 |        1846
```

Fig.: Result Table in Cassandra

<u>Challenges:</u>

1. setup the environment using docker.
2. It was not able to connect with spark-avro and spark-sql-kafka (Lots of time spent on trying to get the correct jar/jdbc drivers).
3. Lot of time spent on finding logic to process the data in cassandra for each row.