1. Project Title: Real-Time Taxi Ride Analytics Pipeline
2. Type of System: Streaming Data Engineering Pipeline
-> The project simulates how a company processes real-time transportation data and generates analytics such as:

- ride counts
- average fares
- busiest pickup zones
- ride demand trends

Instead of processing data in batches, the system processes taxi ride events continuously as they arrive.
3. Logical Problem Statement:
-> Many modern systems (ride-sharing apps, financial transactions, IoT devices) generate continuous streams of data.
-> Traditional batch systems process data after it is stored, which causes delays.
-> Goal of the project:- Build a pipeline that can:

- ingest streaming data
- process it in real time
- simulate real-world streaming systems

-> Example use case: Taxi companies want to analyze ride data as it happens to monitor:

- ride demand
- location trends
- trip duration
- revenue patterns

4. Project Problem Statement:
-> Urban transportation systems generate massive streams of ride data every second.
-> Traditional batch processing systems analyze data hours later, which is not suitable for real-time decision making.
-> Example problems:

- city authorities cannot monitor ride demand in real time
- taxi companies cannot detect busy zones immediately
- analytics dashboards update too slowly.

-> Therefore, we need a real-time data pipeline that can:

- ingest streaming ride events
- process them instantly
- store metrics for analysis
- enable real-time insights

5. Aim of the Project
The aim is to design and implement a real-time data engineering pipeline that:

1. Simulates taxi ride events from the NYC Taxi dataset
2. Streams the data using Kafka
3. Processes the stream using Spark
4. Stores analytics metrics in Cassandra
5. Uses Airflow to orchestrate and manage the pipeline

This project demonstrates core data engineering concepts:

- streaming ingestion
- distributed processing
- scalable storage
- workflow orchestration

6. Dataset:

-> The data is taken from the **NYC Taxi dataset** from the New York City Taxi and Limousine Commission.

-> The dataset contains:

- pickup location
- dropoff location
- passenger count
- trip distance
- fare amount
- Timestamps

We simulate this historical dataset as live streaming events. Instead of sending the whole dataset at once, we send rows one by one with a delay.

7. Project Overview:

-> This project builds a real-time data engineering pipeline that processes taxi ride events and produces analytics that can be visualized on a dashboard. The dataset used is the NYC Taxi Trip dataset, which contains information about taxi rides such as pickup time, drop-off time, distance, fare amount, etc. Instead of processing the dataset as a static batch, the project simulates real-time streaming of taxi rides and processes them continuously to generate insights.

8. Goal of the Project:

-> The aim is to demonstrate how modern data engineering systems handle streaming data pipelines.

-> The pipeline will:

1. Ingest ride events
2. Stream them through a messaging system
3. Process them in real time
4. Store analytics metrics
5. Visualize insights on a dashboard

9. Industrial tools being used:

- Python
- Apache Kafka
- Apache Spark
- Apache Cassandra
- Apache Airflow
- Docker
- Grafana/Power BI

10. Architecture of the Pipeline:

-> The system follows this architecture:

NYC Taxi Dataset
↓
Python Producer
↓
Kafka Topic
↓
Verifying Kafka Topic
↓
Spark Streaming Processing
↓
Bronze Layer (Raw ride events)
↓
Silver Layer (Cleaned & validated ride data)
↓
Gold Layer (Aggregated analytics)
↓
Cassandra Database
↓
Airflow Workflow Management
↓
Power BI Dashboard / Grafana

11. Explanation of Each Stage:

1. Dataset

- Source data containing historical taxi ride records.

2. Python Producer

- Reads rows from the dataset.
- Converts them to JSON messages.
- Sends them to Kafka one event at a time to simulate real-time streaming.

3. Python:Writing pipeline logic

- Simulate taxi ride events
- Send data to kafka
- Write spark streaming jobs
- Interact with Cassandra
- Write Airflow DAGs
- Used because it works good with spark, karka, airflow

4. Kafka: Real-time streaming platform / real time data ingestion

- Kafka acts as a distributed event streaming platform.
- Acts as a message broker (data pipeline highway)
- Stores incoming ride events in a topic called taxi_rides.
- Instead of sending data directly to Spark, we push it into Kafka first.
- Kafka stores data as topics. Every taxi ride becomes a message.
- Allows downstream systems to consume the stream.

- Used as it is the industry standard of event pipelines, handles high throughput streaming, decouples producers and consumers, scalable and fault-tolerant, also used by netflix and uber.

5. Zookeeper: Kafka cluster coordination

- ZooKeeper was used in Kafka to manage broker metadata, leader election, cluster coordination, and topic configuration. Modern Kafka versions are moving to KRaft mode to remove the ZooKeeper dependency.
- ZooKeeper manages broker metadata, leader election, cluster synchronization

6. Spark Streaming: Real time data processing

- Spark Structured Streaming reads events from Kafka and processes them in micro-batches.
- Read messages from Kafka in real time.
- Processes the ride events.
- Computes analytics metrics such as trip counts, averages, or revenue trends.
- Supports large scale data processing, structured streaming, and machine learning

7. Data Lakehouse Architecture : Bronze–Silver–Gold Layers

Bronze, Silver, and Gold layers represent different levels of data processing in modern data engineering systems.

-> **Bronze Layer – Raw Data**

- Stores raw taxi ride events exactly as received from Kafka.
- No transformation is applied.
- Acts as the raw historical record of the data stream.

->**Silver Layer – Cleaned Data**

- Raw events are cleaned and validated.
- Removes null values, invalid distances, or corrupt records.
- Converts timestamps into structured formats.

->**Gold Layer – Aggregated Analytics**

- Business-level analytics are generated.
- Example metrics:
  - average fare
  - average trip distance
  - ride counts
  - revenue trends
- These aggregated metrics are stored in Cassandra for fast querying.

This architecture is commonly used in modern data platforms such as Databricks Lakehouse architectures.
Bronze and Gold layers are persisted in Cassandra.
Silver is an intermediate transformation layer in Spark

8. Cassandra: Data storage for analytics

- Storage layer for processed metrics
- Spark writes different levels of processed data into the system: Bronze Layer – raw ride events | Silver Layer – cleaned and validated ride data | Gold Layer – aggregated

analytics metrics. Cassandra stores the final Gold layer analytics results so that they can be queried efficiently by dashboards such as Power BI or Grafana.

- Stores processed analytics results.
- Optimized for fast reads and writes for streaming workloads.
- Cassandra is used for High speed, Good for streaming data, and has a Distributed NoSQL database.

9. Apache Airflow: Pipeline Orchestration

- Airflow schedules and manages workflows.
- Instead of manually running scripts, Airflow runs them automatically.
- Airflow uses DAGs (Directed Acyclic Graphs).
- Airflow is used to automate pipelines, schedule jobs, monitor pipelines, and is the industry standard.

10. Docker: Infrastructure management

- Run all tools easily
- Installing Kafka, Spark, Cassandra manually is difficult.
- Docker lets us run them as containers.
- Benefits include easy steps, no dependency conflicts, same environment everywhere.

11. Power BI

- Connects to the processed data.
- Displays analytics through dashboards and visualizations.

STEP I

1. Made the project folder ‘taxistream’
2. Inside the folder taxistream, made the folders:

```
>airflow
>data
>docker
>producer
    >producer.py
>spark
    >stream_processor.py
>docker-compose.yml
>Readme,md
>test_data.py
>requirements.txt
```

STEP II: ENVIRONMENT SETUP

1. In Anaconda Terminal we create a new environment:

```
>conda create -n taxic python=3.10
>conda activate taxic
```

2. We install Basic libraries:

```
>pip install kafka-python pyspark cassandra-driver pandas pyarrow apache-airflow
```

This will be used for Python Producer → Kafka, Spark Streaming, Cassandra connection.

- Pyarrow is used for .parquet files.

- kafka-python is used to connect Python producer to Kafka, and send taxi events to Kafka topics.
- pyspark is used to implement Spark streaming jobs, and process data from Kafka
- pandas is used to read and preprocess the NYC dataset, and simulate streaming data
- cassandra-driver is used to connect Spark/Python to Cassandra database, and insert processed analytics
- apache-airflow is used to define workflow DAGs, and schedule pipeline tasks

3. We save the requirements in the taxistream folder as a txt file:

>conda list >requirements.txt

STEP III: DATASET DOWNLOAD

1. Go to the link: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

(This dataset has yellow taxi trip records, green taxi trip records, for each month and year)

2. Download the “2025, yellow taxi trip records (parquet)” and put that inside >data folder.
3. So now the >data folder has >yellow_tripdata_2025-01.parquet.

(.parquet is a columnar data format used in big data systems like Spark, hive, hadoop, so it actually makes the project look realistic)

…….

Testing:

1. We open the terminal:

>activate taxic

>cd “C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

2. We made the >test_data.py under the taxistream folder by using the terminal command:

>type nul > test_data.py

3. Inside it, we write the code:

```
import pandas as pd
df = pd.read_parquet("data/yellow_tripdata_2025-01.parquet")
print(df.head())
print("Total rows:", len(df))
```

4. We run this code in the terminal:

>python test_data.py

```
   VendorID tpep_pickup_datetime tpep_dropoff_datetime  ...  congestion_surcharge  Airport_fee  cbd_congestion_fee
0         1  2025-01-01 00:18:38   2025-01-01 00:26:59  ...                   2.5          0.0                 0.0
1         1  2025-01-01 00:32:40   2025-01-01 00:35:13  ...                   2.5          0.0                 0.0
2         1  2025-01-01 00:44:04   2025-01-01 00:46:01  ...                   2.5          0.0                 0.0
3         2  2025-01-01 00:14:27   2025-01-01 00:20:01  ...                   0.0          0.0                 0.0
4         2  2025-01-01 00:21:34   2025-01-01 00:25:06  ...                   0.0          0.0                 0.0

[5 rows x 20 columns]
Total rows: 3475226
```

5. Milestone completed: Dataset -> Python

STEP IV: DOCKER INFRASTRUCTURE

(To set up docker containers for heavy tools such as: Zookeeper | kafka | Spark | Cassandra | Airflow. All these will be controlled by Docker Compose)

1. We installed Docker Desktop, Windows ARM64 from the link : https://www.docker.com/products/docker-desktop/ .

(Without Docker you would need to manually install and configure: Apache Kafka | Apache ZooKeeper | Apache Spark | Apache Cassandra | Apache Airflow. Each of those installations can take hours and break easily.
With Docker we run them using one command:

>docker compose up)

2. We can then check Docker version:

>docker –version

Docker version 29.2.1, build a5c7197

3. The >docker-compose.yml file will be used to start Apache Kafka and Apache Zookeeper inside docker containers. Hence, **Docker is the infrastructure layer** that runs Kafka, Spark, Cassandra, etc.
4. While the conda env manages python and its libraries(pandas, pyarrow, pyspark, kafka-spark), Docker on the other hand runs external services your Python code connects to, such as: Apache Kafka, Apache ZooKeeper, Apache Spark, Apache Cassandra, Apache Airflow.
How they connect?:
Example pipeline:

Python Producer (local, Conda env)
↓
Kafka (Docker container)
↓
Verify Kafka Topic
↓
Spark Streaming (local, using spark-submit.)
↓
Cassandra (Docker container)

They communicate through ports like: localhost:9092 → Kafka
So, in short, Python scripts use the tools to send and process data, while Docker runs the data infrastructure. Python and Docker work together using localhost connections.

5. After installing, we create the file >docker-compose.yml under the taxistream folder.
(This file will start the services your pipeline needs. For now we will start with Kafka infrastructure: Apache Kafka and Apache ZooKeeper.
Later we will add: Apache Cassandra, Apache Spark, Apache Airflow
6. In >docker-compose.yml, we write the code:

```
services:

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
```

```
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

7. In the Anaconda prompt, we execute:

>docker compose up

It will ask for : View in Docker Desktop | View Config | Enable Watch | Detach

We execute:

>docker compose up -d

Docker will download images for Apache ZooKeeper and Apache Kafka (because the code has it only for zookeeper and kafka)

-> We can check the containers by:

>docker ps

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED       STATUS       PORTS
                    NAMES
fc372b32f9c9   confluentinc/cp-kafka:7.5.0        "/etc/confluent/dock…"   3 days ago    Up 2 days    0.0.0.0:9092->9092/tc
p, [::]:9092->9092/tcp   kafka
1b42226a5712   confluentinc/cp-zookeeper:7.5.0    "/etc/confluent/dock…"   3 days ago    Up 2 days    0.0.0.0:2181->2181/tc
p, [::]:2181->2181/tcp   zookeeper
```

So, Apache ZooKeeper is running on port 2181

Apache Kafka is running on port 9092

So our streaming infrastructure is now alive.

STEP V: PYTHON KAFKA PRODUCER

(Python Kafka producer that sends taxi rides from the dataset into Kafka.)

1. We create file ‘producer.py’ under >producer folder in the taxistream folder.
2. In producer.py, we run the code:

```
from kafka import KafkaProducer
import pandas as pd
import json
import time

# Kafka connection
```

```python
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Load dataset
df = pd.read_parquet('data/yellow_tripdata_2025-01.parquet')

# Limit rows for testing
df = df.head(50)

topic = "taxi_rides"

for index, row in df.iterrows():

    data = row.to_dict()

    # Convert timestamps to string
    for key, value in data.items():
        if hasattr(value, "isoformat"):
            data[key] = value.isoformat()

    producer.send(topic, value=data)

    print(f"Sent ride {index}")

    time.sleep(0.5) # simulate real-time streaming
```

Then we run the command in anaconda prompt:

>python producer/producer.py

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>python producer/producer.py
Sent ride 0
Sent ride 1
Sent ride 2
Sent ride 3
Sent ride 4
Sent ride 5
Sent ride 6
Sent ride 7
Sent ride 8
Sent ride 9
Sent ride 10
Sent ride 11
Sent ride 12
Sent ride 13
Sent ride 14
Sent ride 15
```

3. If for any reason, the keyline is : “NoBrokerAvailable”, that means that Kafka isn’t reachable at localhost:9092 when the producer starts

What to do:

```
>docker compose down
>docker compose up -d    ; and then,
>python producer/producer.py
```

Also, keep docker desktop on.

—----------

BRIEF OF WHAT WE HAVE DONE TILL NOW:

1. We created a conda environment for the project, and installed libraries like: pandas, kafka-python, pyspark, and pyarrow. This environment is where our pipeline code runs.

2. Infrastructure with Docker: We installed Docker and started containers for: Apache ZooKeeper and Apache Kafka. These run as background services.

Kafka is basically a real-time data stream platform.

Analogy:

Think of it like a message pipeline where systems can:

send data → store temporarily → allow other systems to read it

3. Created a Kafka Producer: We wrote `producer.py`, the program:

   A. Reads the NYC taxi dataset
   B. Converts each row to JSON
   C. Sends it to Apache Kafka

4. Hence, so far, the flow is:

Taxi Dataset
↓
Python Producer
↓
Kafka Topic

5. We also added: time.sleep(0.5)

This **simulates real-time streaming** instead of sending everything instantly.

So our program sent:

Sent ride 0

Sent ride 1

…

Sent ride 49

These messages are now inside Kafka.

6. What Kafka is doing right now:

Kafka stores messages inside something called a topic.

Topic : taxi_rides

So internally it looks like:

Kafka Topic: taxi_rides

Ride 0

Ride 1

Ride 2
…
Kafka is waiting for another system to **consume** this stream.
7. To create a kafka topic under docker:
>docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
8. **Before moving forward, check whether kafka is working or not:**
>docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker
exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
taxi_rides

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>
```

—----------

BRIEF OF WHAT HAS TO BE DONE NOW:

1. Need to create a consumer from Apache Spark, which will:
   - Read Messages from Kafka
   - Process them in real time
   - Compute analytics metrics like: rides per minute, avg trip distance, avg fare amount, total revenue trends
2. Spark streaming consumer: Spark will continuously read the kafka stream, parse the incoming ride events, and perform real time processing. It will also compute metrics such as given above.
Cassandra will store analytics results, so that they can be queried efficiently.
Powerbi will visualize analytics such as: trip volume over time, revenue trends, avg trip details, peak ride hours.
3. Final Outcome: The completed system will function as a real-time taxi analytics platform, demonstrating how streaming data pipelines ingest, process, store, and visualize large volumes of event data.

—----------

STEP VI: SPARK STREAMING SCRIPT

1. We create a file >stream_processor.py under >spark folder in taxistream.
2. We write the code in stream_processor.py:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StringType, DoubleType

# Create Spark session
spark = SparkSession.builder \
    .appName("TaxiStreamingProcessor") \
    .config("spark.sql.shuffle.partitions", "2") \
    .getOrCreate()
```

```
spark.sparkContext.setLogLevel("WARN")

# Schema for taxi data
schema = StructType() \
    .add("tpep_pickup_datetime", StringType()) \
    .add("tpep_dropoff_datetime", StringType()) \
    .add("trip_distance", DoubleType()) \
    .add("fare_amount", DoubleType())

# Read stream from Kafka
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "taxi_rides") \
    .option("startingOffsets", "latest") \
    .load()

# Convert Kafka value to string
json_df = df.selectExpr("CAST(value AS STRING)")

# Parse JSON
taxi_df = json_df.select(
    from_json(col("value"), schema).alias("data")
).select("data.*")

# Compute analytics
analytics_df = taxi_df.groupBy().avg("fare_amount", "trip_distance")

# Output to console (for testing)
query = analytics_df.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

3. We go to https://adoptium.net/ , and install Java from there. Temurin 17.0.18+8, Windows 64 bit (.MS
4. We see that this java file has been installed here:

C:\Program Files\Eclipse Adoptium\jdk-17.0.18.8-hotspot
Inside this, there should be a bin folder, having java.exe

STEP VII: Setting JAVA_HOME system variable

1. Press Windows Key → Search “Environment Variables” → Edit the system environment variables → Environment Variables.
2. Under System variables, click New.
   - Variable name: JAVA_HOME
   - Variable value: C:\Program Files\Eclipse Adoptium\jdk-17.0.18.8-hotspot (your Java folder path)
   - We then click Ok
3. Add Java to Path:
   - In the same Environment Variables window, select the variable Path under System variables → Edit → New.
   - We add this to the line: %JAVA_HOME%\bin
     We then click ok
4. To verify, we open the terminal and run java -version and echo %JAVA_HOME%

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>java --version
openjdk 17.0.18 2026-01-20
OpenJDK Runtime Environment Temurin-17.0.18+8 (build 17.0.18+8)
OpenJDK 64-Bit Server VM Temurin-17.0.18+8 (build 17.0.18+8, mixed mode, sharing)

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>echo %JAVA_HOME%
C:\Program Files\Eclipse Adoptium\jdk-17.0.18.8-hotspot
```

STEP VIII: SPARK STREAMING

1. We now run this command in the terminal:
   >python spark/stream_processor.py
2. We also run this command, because even with Java installed, Spark still needs the Kafka connector.
   >spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8 spark/stream_processor.py

We will mostly be using this command.
But, this will not work unless the following steps are completed too:

3. We go to the link https://spark.apache.org/downloads.html, and download Spark 3.5.8, Spark 3.5.8
4. A file gets downloaded: spark-3.5.8-bin-hadoop3
5. We Go to MY PC-> Local Disk C (C:\) ->Create a folder ‘spark’, and put the file ‘spark-3.5.8-bin-hadoop3’ inside it.
6. We now copy this file path
7. We go to Windows Search->Environment Variable, and under “System Variable”, we make:
   - Variable name: SPARK_HOME
   - Value: C:\spark\spark-3.5.8-bin-hadoop3
   - Click Ok
8. Under System Variable itself, we find “Path”, we go there, and click Edit->New->and add: %SPARK_HOME%\bin. Then: Ok Ok Ok.
9. In the terminal, we run command:
   >set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe

-> This is needed because Spark doesn't automatically detect the Anaconda Python inside your `taxic` environment.

10. Now, in the Local C, we again create a folder: hadoop, and inside it, the folder: bin
11. We go to the link:
https://github.com/cdarlint/winutils/tree/master/hadoop-3.2.2/bin
12. From there we download both winutils.exe and hadoop.dll.
13. Now we Set Environment Variables: We go to System Variable->New, and then add:
    - Variable name: HADOOP_HOME
    - Value: C:\hadoop
14. In the System Variable itself, we find the "path" option, there we go to edit->new->and add: C:\hadoop\bin, or, %HADOOP_HOME%\bin

STEP IX: TIME TO USE TWO TERMINALS TOGETHER

1. In Terminal 1, run:

```
>set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
>spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8 spark/stream_processor.py
```

It will show as:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+----------------+------------------+
|avg(fare_amount)|avg(trip_distance)|
+----------------+------------------+
|            NULL|              NULL|
+----------------+------------------+
```

But it won't start as of now.

2. In terminal 2:

```
>python producer/producer.py
```

As this goes on:

```
Sent ride 0
Sent ride 1
Sent ride 2
Sent ride 3
Sent ride 4
```

So does this too:

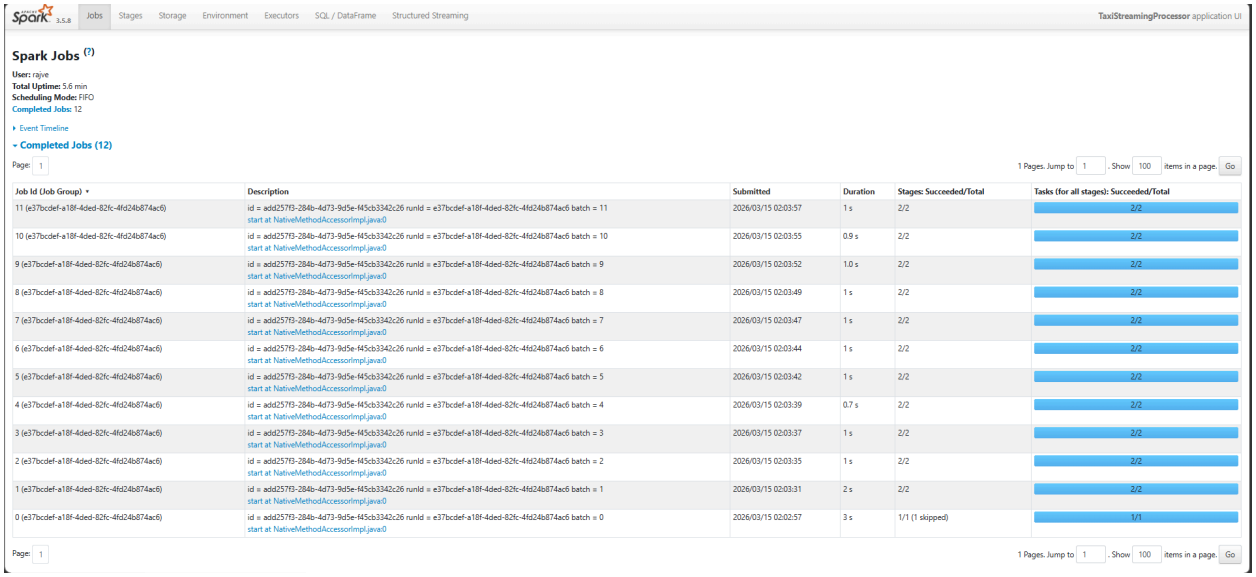
```
Batch: 1
-------------------------------------------
+----------------+------------------+
|avg(fare_amount)|avg(trip_distance)|
+----------------+------------------+
|            10.0|               1.6|
+----------------+------------------+

-------------------------------------------
Batch: 2
-------------------------------------------
+----------------+------------------+
|avg(fare_amount)|avg(trip_distance)|
+----------------+------------------+
|             8.6|           1.06375|
+----------------+------------------+

-------------------------------------------
Batch: 3
-------------------------------------------
+------------------+------------------+
|  avg(fare_amount)|avg(trip_distance)|
+------------------+------------------+
|10.053846153846154|1.3738461538461537|
+------------------+------------------+

-------------------------------------------
Batch: 4
-------------------------------------------
+----------------+------------------+
```

3. We can have a Quick Verification by going on the browser, and typing:
   >http://localhost:4040

-> This is the **Apache Spark UI**. There you can see:

- Jobs
- Stages
- Streaming queries
- Execution DAG
- Processing times

->This is very useful when demonstrating the project.



BRIEF OF WHAT WE HAVE DONE TILL NOW:

1. We now have a real-time data engineering pipeline:

Taxi Dataset

↓

Python Producer

↓

Kafka topic Stream

↓

Spark Structured Streaming

↓

Real-time processing in micro-batches

Using:

- Apache Kafka
- Apache Spark
- Apache Spark Structured Streaming

2. Now, in short,

For Terminal 1:

```
>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>docker ps
>docker compose up -d
>set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
>spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8 spark/stream_processor.py
```

And for Terminal 2:
>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>python producer/producer.py
3. Completed stages:
    1. Environment setup
    2. Spark installation
    3. Hadoop utilities setup
    4. Kafka services started with Docker
    5. Producer script sending taxi ride events
    6. Spark streaming job consuming Kafka data
    7. Micro-batch processing working

The **end-to-end pipeline is functional**.
4. Till here, this project demonstrates a modern real-time data engineering pipeline that;
- simulates live data streams
- transports data via Kafka
- processes data using Spark streaming
- operates in micro-batch streaming mode

->It showcases the fundamental architecture used in large-scale data systems such as ride-sharing platforms, financial trading systems, and IoT analytics.
5. Skills that have been demonstrated till yet in this project:
- Streaming Data Engineering
- Kafka Event Pipelines
- Spark Structured Streaming
- Dockerized Infrastructure
- Real-time Data Processing
- Data Pipeline Architecture

—----------

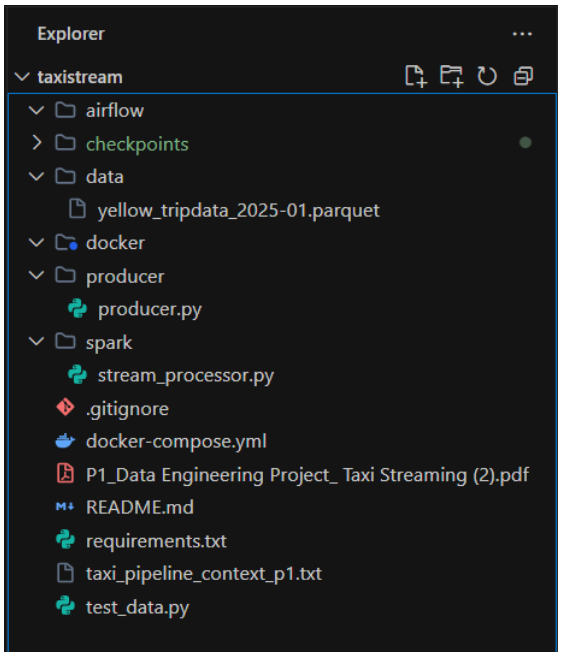
BRIEF OF WHAT HAS TO BE DONE NOW:

1. Data storage in apache Cassandra
2. Workflow orchestration in Apache Airflow for scheduling pipelines.
3. Create analytics dashboards using Power BI or Grafana
4. Continuous streaming: To Modify the producer so it continuously streams events instead of stopping at 50 records.

—----------

STEP X: THE CASSANDRA BUILDING STAGE : PART 1 : UPDATING CODES

1. Before starting, this is how our project file looks like yet:



Where;

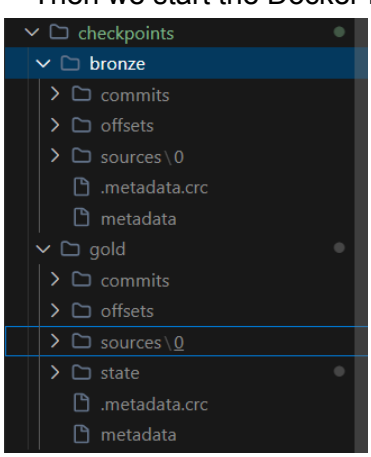
i) The Airflow folder will be used for workflow orchestration. This folder will contain Apache Airflow DAGs. It will be used to: Schedule the pipeline, Start Spark jobs, Run batch tasks, Monitor pipelines. Right now it may be empty, but it is meant for automation.

ii) The Checkpoints folder is for Spark Structured Streaming state storage. Spark uses this folder to store: Kafka offsets, Streaming progress, Metadata, State store data. Why does it exist? Without checkpoints, Spark cannot know: which Kafka messages were already processed, or, where to resume after restart. So this folder enables fault tolerance and recovery. If deleted, we need to follow these steps:

-> Restart Docker Desktop: Open task manager, and end these if they exist:

- Docker Desktop
- Docker Desktop Backend
- Com.docker.backend

Then we start the Docker Desktop again.



iii) The Data folder stores the NYC dataset: yellow_tripdata_2025-01.parquet
iv) The Producer folder is one of the main folders, and it is used for Kafka data ingestion. The producer.py file reads the taxi datasets, converts rows into JSON events, and sends events to kafka. So this script simulates real-time taxi ride events.
v) The Spark folder is one of the main folders used, and acts as the Streaming processing layer. The file: stream_processor.py is the main streaming engine. It:

- Reads events from Kafka
- Parses JSON
- Processes data
- Writes to Cassandra

-> Pipeline inside this script:

Kafka
↓
Spark Streaming
↓
Bronze Table (raw)
↓
Silver Table (cleaned)
↓
Gold Table (aggregated)

-> This implements the Medallion Architecture.
vi) The .gitignore file prevents unnecessary files from being uploaded such as:

- checkpoints/
- __pycache__/
- .env
- *.log

vii) The docker-compose.yml file is one of the main files, which starts the whole infrastructure. Till this stage of the project, it launches Kafka, Zookeeper, and Cassandra. The command: docker compose up -d creates the local data platform.
viii) The test_data.py is basically used for testing / experimentation to inspect dataset, test transformations, and debug Kafka events. It isn't part of the main project.
2. These are the codes in every main file being used. Though only the stream_processor.py file has been updated for the cassandra stage, just for the sake of documentation, here are the codes, as they should for the working of this stage:
2.1) **producer.py :**

```
from kafka import KafkaProducer
import pandas as pd
import json
import time

# Kafka connection
producer = KafkaProducer(
```

```
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Load dataset
df = pd.read_parquet('data/yellow_tripdata_2025-01.parquet')

# Limit rows for testing
df = df.head(50)

topic = "taxi_rides"

for index, row in df.iterrows():

    data = row.to_dict()

    # Convert timestamps to string
    for key, value in data.items():
        if hasattr(value, "isoformat"):
            data[key] = value.isoformat()

    producer.send(topic, value=data)

    print(f"Sent ride {index}")

    time.sleep(0.5) # simulate real-time streaming
```

Only 49 rides are being sent for the time being

2.2) **docker-compose.yml** :

```
services:

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
```

```yaml
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  cassandra:
    image: cassandra:4.1
    container_name: cassandra
    ports:
      - "9042:9042"
    environment:
      - CASSANDRA_CLUSTER_NAME=TaxiCluster
    volumes:
      - cassandra_data:/var/lib/cassandra

volumes:
  cassandra_data:
```

2.3) **stream_processor.py** : This is where real change has been made:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date
from pyspark.sql.functions import col, from_json, current_timestamp, expr,
avg, count, to_timestamp
from pyspark.sql.types import StructType, StringType, DoubleType,
IntegerType
from pyspark.sql.functions import window

# -------------------- Spark Session --------------------
spark = SparkSession.builder \
    .appName("TaxiStreamingProcessor") \
    .config("spark.sql.shuffle.partitions", "2") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .config("spark.driver.host", "127.0.0.1") \
    .config("spark.sql.streaming.stateStore.providerClass",
```

```
"org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvid
er") \
    .getOrCreate()

spark.catalog.clearCache()
spark.sparkContext.setLogLevel("WARN")

# -------------------- Full schema (all fields from your Kafka JSON)
--------------------
schema = StructType() \
    .add("VendorID", IntegerType()) \
    .add("tpep_pickup_datetime", StringType()) \
    .add("tpep_dropoff_datetime", StringType()) \
    .add("passenger_count", DoubleType()) \
    .add("trip_distance", DoubleType()) \
    .add("RatecodeID", DoubleType()) \
    .add("store_and_fwd_flag", StringType()) \
    .add("PULocationID", IntegerType()) \
    .add("DOLocationID", IntegerType()) \
    .add("payment_type", IntegerType()) \
    .add("fare_amount", DoubleType()) \
    .add("extra", DoubleType()) \
    .add("mta_tax", DoubleType()) \
    .add("tip_amount", DoubleType()) \
    .add("tolls_amount", DoubleType()) \
    .add("improvement_surcharge", DoubleType()) \
    .add("total_amount", DoubleType()) \
    .add("congestion_surcharge", DoubleType()) \
    .add("Airport_fee", DoubleType()) \
    .add("cbd_congestion_fee", DoubleType())

# -------------------- Read Kafka Stream --------------------
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "taxi_rides") \
    .option("startingOffsets", "earliest") \
    .load()
```

```
json_df = df.selectExpr("CAST(value AS STRING) AS raw_event") \
            .withColumn("ride_id", expr("uuid()"))
# -------------------- BRONZE --------------------
bronze_df = json_df.withColumn("created_at", current_timestamp())
bronze_query = bronze_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="bronze_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/bronze") \
    .outputMode("append") \
    .start()

# -------------------- SILVER --------------------
silver_df = json_df.select(
    from_json(col("raw_event"), schema).alias("data"),
    col("ride_id")
).select("ride_id", "data.*")

silver_df = silver_df.select(
    col("ride_id"),
    col("VendorID").alias("vendorid"),
    col("tpep_pickup_datetime"),
    col("tpep_dropoff_datetime"),
    col("passenger_count"),
    col("trip_distance"),
    col("RatecodeID").alias("ratecodeid"),
    col("store_and_fwd_flag"),
    col("PULocationID").alias("pulocationid"),
    col("DOLocationID").alias("dolocationid"),
    col("payment_type"),
    col("fare_amount"),
    col("extra"),
    col("mta_tax"),
    col("tip_amount"),
    col("tolls_amount"),
    col("improvement_surcharge"),
    col("total_amount"),
```

```
    col("congestion_surcharge"),
    col("Airport_fee").alias("airport_fee"),
    col("cbd_congestion_fee"),
    to_timestamp("tpep_pickup_datetime").alias("pickup_datetime"),
    to_timestamp("tpep_dropoff_datetime").alias("dropoff_datetime")
)
# -------------------- GOLD --------------------
gold_df = silver_df \
    .withColumn("pickup_time", to_timestamp("tpep_pickup_datetime")) \
    .withWatermark("pickup_time", "10 minutes") \
    .groupBy(window("pickup_time", "1 minute")) \
    .agg(
        avg("fare_amount").alias("avg_fare"),
        avg("trip_distance").alias("avg_trip_distance"),
        count("*").alias("ride_count")
    ) \
    .withColumn("agg_time", current_timestamp()) \
    .withColumn("agg_date", to_date("agg_time")) \
    .drop("window")

gold_query = gold_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="gold_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/gold") \
    .outputMode("update") \
    .start()

# -------------------- Await Termination --------------------
spark.streams.awaitAnyTermination()
```

2.4) **requirements.txt** :

```
# packages in environment at C:\Users\rajve\anaconda3\envs\taxic:
#
# Name                    Version          Build            Channel
bzip2                     1.0.8            h2bbff1b_6
```

```
ca-certificates           2025.12.2        haa95532_0
cassandra-driver          3.29.3           pypi_0           pypi
click                     8.3.1            pypi_0           pypi
colorama                  0.4.6            pypi_0           pypi
expat                     2.7.4            hd7fb8db_0
findspark                 2.0.1            pypi_0           pypi
geomet                    1.1.0            pypi_0           pypi
kafka-python              2.3.0            pypi_0           pypi
libexpat                  2.7.4            hd7fb8db_0
libffi                    3.4.4            hd77b12b_1
libzlib                   1.3.1            h02ab6af_0
numpy                     2.2.6            pypi_0           pypi
openssl                   3.5.5            hbb43b14_0
packaging                 25.0             py310haa95532_1
pandas                    2.3.3            pypi_0           pypi
pip                       26.0.1           pyhc872135_0
py4j                      0.10.9.9         pypi_0           pypi
pyarrow                   23.0.1           pypi_0           pypi
pyspark                   4.1.1            pypi_0           pypi
python                    3.10.19          h981015d_0
python-dateutil           2.9.0.post0      pypi_0           pypi
pytz                      2026.1.post1     pypi_0           pypi
setuptools                80.10.2          py310haa95532_0
six                       1.17.0           pypi_0           pypi
sqlite                    3.51.1           hee5a0db_1
tk                        8.6.15           hf199647_0
tzdata                    2025.3           pypi_0           pypi
ucrt                      10.0.22621.0     haa95532_0
vc                        14.3             h2df5915_10
vc14_runtime              14.44.35208      h4927774_10
vs2015_runtime            14.44.35208      ha6b5a95_10
wheel                     0.46.3           py310haa95532_0
xz                        5.8.2            h53af0af_0
zlib                      1.3.1            h02ab6af_0
```

STEP XI: THE CASSANDRA BUILDING STAGE : PART 2: USING CASSANDRA SHELL

We follow the required steps to create tables for bronze, silver, and gold levels. (Bronze and Gold are stored in Cassandra, Silver exists only in Spark):

1. We start docker: kafka, zookeeper, cassandra, in the terminal:
   > docker compose up -d
2. We open the cassandra shell:
   >docker exec -it cassandra cqlsh
3. We create a database for the tables in this project:
   CREATE KEYSPACE taxi_streaming WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'} AND durable_writes = true;
4. We check the whether the database has been made or not:
   cqlsh> DESCRIBE KEYSPACES;

```
cqlsh> DESCRIBE KEYSPACES;

my_keyspace  system_distributed  system_views           taxi_streaming
system       system_schema       system_virtual_schema
system_auth  system_traces       taf
```

5. We use this database:
   cqlsh>USE taxi_streaming:
   cqlsh:taxi_streaming>
6. We create the tables inside it:

6.1) **For bronze layer:**

```
cqlsh> CREATE TABLE bronze_rides (
        ride_id text PRIMARY KEY,
        created_at timestamp,
        raw_event text
        );
```

6.2) **For silver layer:**

```
cqlsh>CREATE TABLE silver_rides (
        ride_id text PRIMARY KEY,
        airport_fee double,
        cbd_congestion_fee double,
        congestion_surcharge double,
        dolocationid int,
        dropoff_datetime timestamp,
        extra double,
        fare_amount double,
        improvement_surcharge double,
        mta_tax double,
        passenger_count double,
        payment_type int,
        pickup_datetime timestamp,
        pulocationid int,
```

```
ratecodeid double,
store_and_fwd_flag text,
 tip_amount double,
 tolls_amount double,
 total_amount double,
 tpep_dropoff_datetime text,
 tpep_pickup_datetime text,
 trip_distance double,
 vendorid int
);
```

6.3) **For Gold Layer:**

```
cqlsh>CREATE TABLE gold_rides (
agg_date date,
agg_time timestamp,
avg_fare double,
avg_trip_distance double,
ride_count bigint,
PRIMARY KEY (agg_date, agg_time)
) WITH CLUSTERING ORDER BY (agg_time DESC);
```

-> The Bronze Layer is used to Store raw Kafka messages, involves no cleaning, and is basically just ingestion of data

-> The Silver Layer parses JSON schema, and cleans and structures taxi ride fields

-> The Gold Layer performs streaming aggregation analytics, and computes queries involving avg fare, avg trip distance, or, ride count per time window.

->The Gold table is partitioned by agg_date and clustered by agg_time so that analytics queries can efficiently retrieve recent ride metrics by day.

-> The Gold layer also uses windowing for real-time analysis and time-based aggregation (per minute)

-> Hence the architecture represents the **Medallion Architecture,** used in modern data engineering pipelines.

```
                Taxi Dataset
                     ↓
            Producer (Python)
                     ↓
          Kafka Topic (taxi_rides)
                     ↓
        Spark Structured Streaming
          ↓          ↓          ↓
       Bronze     Silver     Gold
       (raw)      (clean)    (analytics
```

NOTE:

Bronze and Gold layers are persisted in Cassandra.

Silver layer exists only inside Spark for transformations.

6.4) To check the tables and its structure, we run the command:

cqlsh>DESCRIBE taxi_streaming;

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it cassandra cqlsh
Connected to TaxiCluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.10 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> DESCRIBE taxi_streaming;

CREATE KEYSPACE taxi_streaming WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}  AND durable_writes = true;

CREATE TABLE taxi_streaming.bronze_rides (
    ride_id text PRIMARY KEY,
    created_at timestamp,
    raw_event text
) WITH additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';

CREATE TABLE taxi_streaming.gold_rides (
    agg_date date,
    agg_time timestamp,
    avg_fare double,
    avg_trip_distance double,
    ride_count bigint,
    PRIMARY KEY (agg_date, agg_time)
) WITH CLUSTERING ORDER BY (agg_time DESC)
    AND additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
```

```
CREATE TABLE taxi_streaming.silver_rides (
    ride_id text PRIMARY KEY,
    airport_fee double,
    cbd_congestion_fee double,
    congestion_surcharge double,
    dolocationid int,
    dropoff_datetime timestamp,
    extra double,
    fare_amount double,
    improvement_surcharge double,
    mta_tax double,
    passenger_count double,
    payment_type int,
    pickup_datetime timestamp,
    pulocationid int,
    ratecodeid double,
    store_and_fwd_flag text,
    tip_amount double,
    tolls_amount double,
    total_amount double,
    tpep_dropoff_datetime text,
    tpep_pickup_datetime text,
    trip_distance double,
    vendorid int
) WITH additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
cqlsh>
```

NOTE: We ensure that the columns made here in the cqlsh database, and the ones written in the stream_processor.py file are the same. Otherwise Spark may try writing into columns, which Cassandra might not have.

STEP XII: USING CASSANDRA

We assume that Docker Desktop is down, and no 'checkpoints' file is made yet. This is actually how this stage of the project started. But anyhow just for demonstration, we may delete the checkpoints folder, and also turn off the docker engine, to show how to start the pipeline from the very beginning (because we can't always keep our docker on)

Also, remember:

1. If we delete the Checkpoint folder, we again have to restart Docker, and kafka containers.
2. If we had turned off the docker down, we would have to delete the checkpoint folder.

This happens because Spark stores Kafka offsets inside the checkpoint folder. Example:
Checkpoint says: 150

But if we run docker compose down, Kafka restarts and the topic starts again at:offset: 0. Now spark sees: expected offset: 150, and actual offset: 0.Spark thinks data was lost, so it throws the offset error we saw.

Anyway, we start with how to proceed with the project, when no checkpoint folder is there yet, or if we delete the checkpoint folder:

CASE I : WHEN THERE IS NO CHECKPOINT FOLDER YET:

1. We delete the checkpoint folder from the main project file taxistream, if it's already present.
2. We go to Task Manager, and end these tasks if they exist:
   - Docker Desktop
   - Docker Desktop Backend
   - com.docker.backend

   Then we start Docker Desktop again, just by clicking on the Docker Desktop icon.
3. Now, in Terminal 1, we run the follow commands:

   ```
   >activate taxic
   >cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
   >docker compose down
   >docker compose up -d
   >docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
   ```

   (to create a kafka topic, has to be done everytime the docker has been up, after down)

   ```
   >docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
   ```

   (to verify whether docker is running or not)

```
(base) C:\Users\rajve>activate taxic

(taxic) C:\Users\rajve>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose down
[+] down 4/4
 ✔Container cassandra        Removed                                                                    0.1s
 ✔Container kafka            Removed                                                                    0.1s
 ✔Container zookeeper        Removed                                                                    0.1s
 ✔Network taxistream_default Removed                                                                    0.3s

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose up -d
[+] up 4/4
 ✔Network taxistream_default Created                                                                    0.1s
 ✔Container cassandra        Started                                                                    1.3s
 ✔Container zookeeper        Started                                                                    1.3s
 ✔Container kafka            Started                                                                    1.4s

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --list --boot
strap-server localhost:9092


(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --create --to
pic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issu
es it is best to use either, but not both.
Created topic taxi_rides.
```

>set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe

>spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py

And then basically it'll go up to somewhere like:

```
26/03/16 23:19:58 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 127.0.0.1, 62535, None)
26/03/16 23:19:58 INFO SharedState: Setting hive.metastore.warehouse.dir ('null') to the value of spark.sql.warehouse.di
r.
26/03/16 23:19:58 INFO SharedState: Warehouse path is 'file:/C:/Users/rajve/Desktop/SkillIssues/Personal%20Projects/taxi
stream/spark-warehouse'.
26/03/16 23:20:03 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Dataset
s and will be disabled.
26/03/16 23:20:04 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commi
t, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
26/03/16 23:20:04 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Dataset
s and will be disabled.
26/03/16 23:20:04 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commi
t, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
```

……..

4. Meanwhile in Terminal 2:

>activate taxic

>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

>python producer/producer.py

It'll send rides:

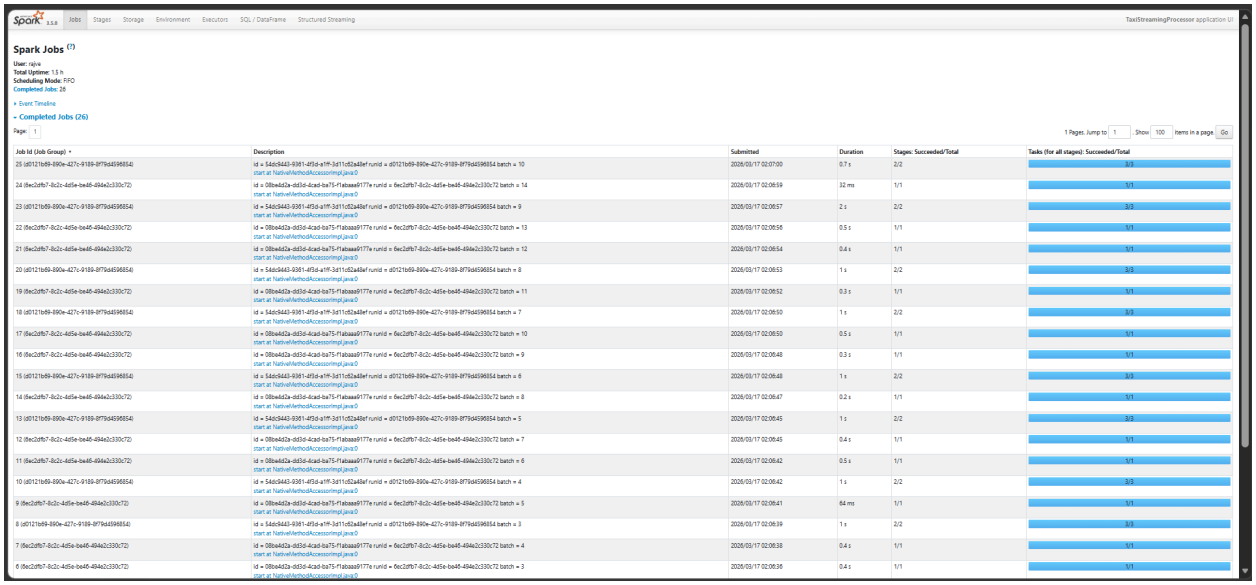
```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>python producer/producer.py
Sent ride 0
Sent ride 1
Sent ride 2
Sent ride 3
Sent ride 4
Sent ride 5
Sent ride 6
Sent ride 7
Sent ride 8
```

……

To Verify, We go to local host:

http://localhost:4040/jobs/ ([TaxiStreamingProcessor - Spark Jobs]())

Here, we'll be able to see the Spark UI:



For the first time, 20,22,26 jobs are expected, for 49 rides being sent.

We can also check whether Cassandra is giving results or not, by going to the cqlsh shell. This is also to be done in the terminal 2, as terminal 1 is still running spark.

```
>docker exec -t cassandra cqlsh
cqlsh>USE taxi_streaming;
cqlsh> SELECT * FROM taxi_streaming.bronze_rides LIMIT 5;
cqlsh> SELECT * FROM taxi_streaming.silver_rides LIMIT 5;
cqlsh>SELECT * FROM taxi_streaming.gold_rides WHERE agg_date = '2026-03-16' LIMIT 10;
```

And notice that calling the bronze table only gives raw data, calling the silver layer gives the cleaned structure, and the gold layer gives proper analysis.

```
cqlsh> SELECT * FROM taxi_streaming.bronze_rides LIMIT 5;

 ride_id                              | created_at                      | raw_event
--------------------------------------+---------------------------------+-------------------------------------
```

```
cqlsh> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;

 agg_date   | agg_time                        | avg_fare | avg_trip_distance | ride_count
------------+---------------------------------+----------+-------------------+------------
 2026-03-17 | 2026-03-16 20:36:55.983000+0000 |      9.3 |              1.63 |          1
 2026-03-17 | 2026-03-16 20:36:52.852000+0000 |     27.5 |              4.26 |          1
 2026-03-17 | 2026-03-16 20:36:49.998000+0000 |    12.45 |             1.725 |          3
 2026-03-17 | 2026-03-16 20:36:47.372000+0000 |    13.15 |             1.665 |          2
 2026-03-17 | 2026-03-16 20:36:44.428000+0000 |        0 |              0.69 |          2

(5 rows)
```

Some more examples:

```
cqlsh> SELECT * FROM taxi_streaming.gold_rides WHERE agg_date = '2026-03-16' LIMIT 10;

 agg_date   | agg_time                        | avg_fare | avg_trip_distance | ride_count
------------+---------------------------------+----------+-------------------+------------
 2026-03-16 | 2026-03-16 17:51:09.388000+0000 |      9.3 |              1.63 |          3
 2026-03-16 | 2026-03-16 17:51:06.443000+0000 |     27.5 |              4.26 |          3
 2026-03-16 | 2026-03-16 17:51:02.922000+0000 |    12.45 |             1.725 |          9
 2026-03-16 | 2026-03-16 17:50:56.475000+0000 |    1.275 |            0.6175 |          8
 2026-03-16 | 2026-03-16 17:50:53.470000+0000 |    13.78 |              1.94 |          5
 2026-03-16 | 2026-03-16 17:46:16.832000+0000 |      9.3 |              1.63 |          2
 2026-03-16 | 2026-03-16 17:46:13.225000+0000 |      7.9 |               0.7 |          2
 2026-03-16 | 2026-03-16 17:46:10.076000+0000 |    12.45 |             1.725 |          6
 2026-03-16 | 2026-03-16 17:46:03.043000+0000 |     1.02 |             0.632 |          5
 2026-03-16 | 2026-03-16 17:45:59.479000+0000 | 14.66667 |           2.08333 |          3
```

```
cqlsh> SELECT * FROM taxi_streaming.gold_rides;

 agg_date   | agg_time                        | avg_fare | avg_trip_distance | ride_count
------------+---------------------------------+----------+-------------------+------------
 2026-03-17 | 2026-03-16 20:36:55.983000+0000 |      9.3 |              1.63 |          1
 2026-03-17 | 2026-03-16 20:36:52.852000+0000 |     27.5 |              4.26 |          1
 2026-03-17 | 2026-03-16 20:36:49.998000+0000 |    12.45 |             1.725 |          3
 2026-03-17 | 2026-03-16 20:36:47.372000+0000 |    13.15 |             1.665 |          2
 2026-03-17 | 2026-03-16 20:36:44.428000+0000 |        0 |              0.69 |          2
 2026-03-17 | 2026-03-16 20:36:41.358000+0000 |     16.3 |               1.9 |          1
 2026-03-17 | 2026-03-16 20:36:38.445000+0000 |     14.2 |              2.29 |          1
 2026-03-17 | 2026-03-16 20:36:34.651000+0000 |      5.8 |               0.6 |          2
 2026-03-17 | 2026-03-16 20:36:31.222000+0000 |       10 |               1.6 |          1
 2026-03-16 | 2026-03-16 17:51:09.388000+0000 |      9.3 |              1.63 |          3
 2026-03-16 | 2026-03-16 17:51:06.443000+0000 |     27.5 |              4.26 |          3
 2026-03-16 | 2026-03-16 17:51:02.922000+0000 |    12.45 |             1.725 |          9
 2026-03-16 | 2026-03-16 17:50:56.475000+0000 |    1.275 |            0.6175 |          8
 2026-03-16 | 2026-03-16 17:50:53.470000+0000 |    13.78 |              1.94 |          5
 2026-03-16 | 2026-03-16 17:46:16.832000+0000 |      9.3 |              1.63 |          2
 2026-03-16 | 2026-03-16 17:46:13.225000+0000 |      7.9 |               0.7 |          2
 2026-03-16 | 2026-03-16 17:46:10.076000+0000 |    12.45 |             1.725 |          6
 2026-03-16 | 2026-03-16 17:46:03.043000+0000 |     1.02 |             0.632 |          5
 2026-03-16 | 2026-03-16 17:45:59.479000+0000 | 14.66667 |           2.08333 |          3
 2026-03-16 | 2026-03-16 17:40:02.796000+0000 |      9.3 |              1.63 |          1
 2026-03-16 | 2026-03-16 17:39:58.854000+0000 |     27.5 |              4.26 |          1
 2026-03-16 | 2026-03-16 17:39:54.626000+0000 |      1.7 |          0.593333 |          3
 2026-03-16 | 2026-03-16 17:39:50.785000+0000 |    13.15 |             1.665 |          2
 2026-03-16 | 2026-03-16 17:39:45.283000+0000 |     19.1 |               2.8 |          2
 2026-03-16 | 2026-03-16 17:39:40.819000+0000 |      5.8 |               0.6 |          2
 2026-03-16 | 2026-03-16 17:39:37.052000+0000 |       10 |               1.6 |          1
 2026-03-16 | 2026-03-16 17:07:52.969000+0000 |      9.3 |              1.63 |          1
 2026-03-16 | 2026-03-16 17:07:49.019000+0000 |     27.5 |              4.26 |          1
 2026-03-16 | 2026-03-16 17:07:45.181000+0000 |    12.45 |             1.725 |          3
 2026-03-16 | 2026-03-16 17:07:41.708000+0000 |    13.15 |             1.665 |          2
 2026-03-16 | 2026-03-16 17:07:37.417000+0000 |     16.3 |               1.9 |          2
 2026-03-16 | 2026-03-16 17:07:33.219000+0000 |     19.1 |               2.4 |          2
 2026-03-16 | 2026-03-16 17:07:29.547000+0000 |      5.8 |               0.6 |          2
 2026-03-16 | 2026-03-16 17:07:25.199000+0000 |       10 |               1.6 |          1
 2026-03-16 | 2026-03-16 12:59:10.103000+0000 |     27.5 |              4.26 |          6
 2026-03-16 | 2026-03-16 12:53:31.258000+0000 |    12.45 |             1.725 |         15
 2026-03-16 | 2026-03-16 12:44:35.602000+0000 |     27.5 |              4.26 |          4
 2026-03-16 | 2026-03-16 11:22:21.387000+0000 |      9.3 |              1.63 |          3
 2026-03-16 | 2026-03-16 11:22:18.882000+0000 |      7.9 |               0.7 |          3
 2026-03-16 | 2026-03-16 11:22:16.363000+0000 |     27.5 |              4.26 |          3
 2026-03-16 | 2026-03-16 11:22:13.737000+0000 |      1.7 |          0.593333 |          9
 2026-03-16 | 2026-03-16 11:22:08.831000+0000 |    13.78 |              1.94 |          8
 2026-03-16 | 2026-03-16 11:15:20.592000+0000 |      9.3 |              1.63 |          2
 2026-03-16 | 2026-03-16 11:15:17.863000+0000 |      7.9 |               0.7 |          2
 2026-03-16 | 2026-03-16 11:15:14.966000+0000 |     27.5 |              4.26 |          2
 2026-03-16 | 2026-03-16 11:15:11.727000+0000 |      1.7 |          0.593333 |          6
 2026-03-16 | 2026-03-16 11:15:08.163000+0000 |     1.02 |             0.632 |          5
 2026-03-16 | 2026-03-16 11:14:59.404000+0000 | 14.66667 |           2.08333 |          3
 2026-03-16 | 2026-03-16 11:07:55.842000+0000 |     27.5 |              4.26 |          4

(49 rows)
```

CASE II : WHEN THE CHECKPOINT FOLDER IS ALREADY THERE, AND DOCKER WAS NEVER DONE DOWN:

Suppose after the above process, we never did the docker down, basically it just remained up the whole time. In that case, since the docker was up the whole time, the checkpoints don't need to be deleted, and the kafka containers made are still active, even if the terminals are closed.

This will only work if docker wasn't done down and the checkpoints were not deleted. If any one was even done, we have to start from the beginning, that is, start docker, make new kafka containers, and run stream_processor.py which will create new checkpoints.

Anyway, for Case II:

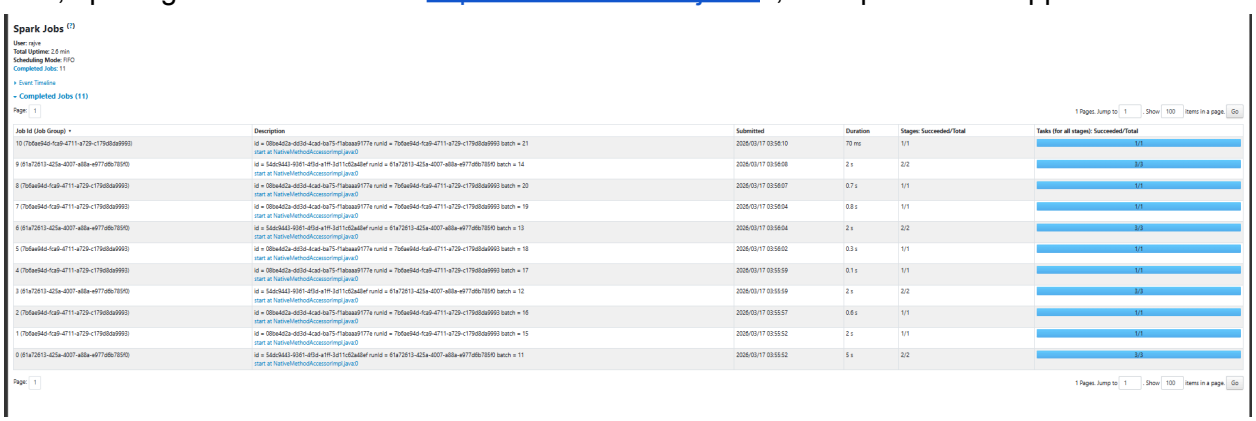
In terminal 1:

```
>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
>spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py
```

Meanwhile, in terminal 2:

```
>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>python producer/producer.py
```

Now, opening the local host link :http://localhost:4040/jobs/ , the Spark UI will appear.



However, notice that this is actually less than what came earlier, and the more we run the pipeline like this, the less it may become.

This is because:

->When you deleted checkpoints, or didn't have them, Spark had to rebuild everything:

- Create state store
- Initialize stream metadata
- Build checkpoint directories
- Start Kafka offsets tracking
- Create Cassandra connection
- Run initial micro-batches

All this creates extra initialization jobs.

->->Meanwhile the second run had fewer jobs. This is because since we did not delete checkpoints, Spark already had:

- Kafka offsets
- State store metadata
- Query progress
- Schema information

—----------

BRIEF OF WHAT WE HAVE DONE TILL NOW:

1. Built a real-time data engineering system:

Taxi Dataset
↓
Python Producer
↓
Kafka Topic (taxi_rides)
↓
Spark Structured Streaming
↓
Bronze Layer (raw events)
↓
Silver Layer (cleaned structured rides)
↓
Gold Layer (aggregated analytics)
↓
Cassandra

2. Have completed the following steps:

✔Streaming ingestion
✔ Schema parsing
✔ Medallion architecture (Bronze/Silver/Gold)
✔ Windowed aggregation
✔ Stateful streaming
✔ Cassandra sink
✔ Checkpointing
✔ Spark monitoring via UI

3. Recent Improvements of what we did in the cassandra stage of the project:

- window aggregation
- removal of unnecessary columns
- state store fix
- stability improvements

4.COMMON ISSUES & RULES

1. If checkpoints are deleted → restart everything
2. If docker is down → Kafka topics are gone
3. If no data in UI → check producer
4. Silver has no checkpoint (not a streaming sink)
5. First run = many jobs, later runs = fewer jobs

—----------

BRIEF OF WHAT HAS TO BE DONE NOW:

1. Real-time dashboard visualization using grafana, where dashboard metrics are:
    - rides per minute
    - average fare
    - average trip distance
2. Workflow Orchestration using Airflow, which can:
    - start Kafka
    - start Spark job
    - run producer
    - monitor pipeline
3. Adding Additional metrics like:
    - rides_by_location
    - rides_by_payment_type

If wanted in the future. These will be separate tables built in cqlsh, like:
gold_rides_by_location
Gold_by_payment_type

—----------

STEP XIII: IMPLEMENTATION OF SURGE, ANOMALY, AND HOTSPOT DETECTION

1. Problem:
Till now, we had:
    - Proper Bronze → Silver → Gold flow
    - Watermark + window aggregation
    - Cassandra integration working
    - Clean schema handling

And our gold_df only did: avg_fare, avg_trip_distance, ride_count;
Which was kind of just descriptive analytics
What was missing was a real-time decision system
So now our goal is to make a real-time decision-making system for ride demand and pricing.
2. Goal:
-> Adding Surge Detection
-> Anomaly Detection
-> Hotspot Detection
All inside the Spark Streaming Layer, and persist results properly in Cassandra (Gold Layer).
2.1) **The Surge Detection:**
Surge detection identifies high-demand periods in real-time using ride density in time windows, enabling dynamic pricing and supply optimization.
“If too many rides are happening in a short time → demand is high”
Ex: when(ride_count > 5, "HIGH_DEMAND")
Basically;
    - Too many people booking rides
    - Drivers may be fewer
    - System needs to react

Companies may use this feature to:

- Increase prices (surge pricing)
- Send more drivers to that area
- Reduce waiting time
- Rush due to Office hours
- Sudden rush due to rain

2.2) **The Anomaly Detection:**

Anomaly detection flags unusual fare patterns in real-time, helping identify pricing errors or abnormal ride conditions.

“If fare suddenly becomes too high → something unusual is happening”

Ex: when(avg_fare > 100, "ABNORMAL")

Basically;

- Pricing bug
- Fraud / overcharging
- Unexpected traffic or route

Companies may use this feature to:

- Detect pricing errors
- Prevent fraud
- Alert system failures

2.3) **The Hotspot Detection:**

Hotspot detection identifies high-demand locations based on ride density, enabling better driver allocation and operational efficiency.

“Which locations have high ride activity?”

Ex: groupBy(window, pulocationid)

Basically;

- Areas with high demand
- Busy zones (airports, offices, events)

Companies may use this feature to:

- Identify high-demand zones
- Send drivers to those locations
- Optimize driver distribution and planning driver positioning
- Reduce idle time

NOTE:

Surge = “WHEN demand is high” (time-based)

Ex: The Code: groupBy(window("pickup_time", "1 minute"))

Asks: “In this time window, how many rides happened?”

So if the ride count between 10:00 and 10:01 is 8, then it has high demand

Hotspot = “WHERE demand is high” (location-based)

Ex: The Code: groupBy(window("pickup_time", "5 minutes"), pulocationid)

Asks: “Which location has the most rides?”

So if the ride of a specific location, like an Airport, is 20, then it's a hotspot.

3. Before we start, we need to understand what changes and updations were taken for this stage of the project. The docker-compose.yml file was not touched. Major changes were taken in:
-> The stream_processor.py file
-> The producer.py file (basically we increased the count of df.count(), for testing more.)
-> The cqlsh, taxi_streaming database (basically addition of the hotspot table, and updation of the gold table)
4. Steps taken:
-> Modified the gold_df block of the stream_processor.py file such that the system can now detect real-time demand surges and pricing anomalies.
-> We created a new hotspot_df , and hotspot_query layer.
-> We altered the existing gold table in cqlsh:

```
>ALTER TABLE taxi_streaming.gold_rides ADD surge_flag text;
>ALTER TABLE taxi_streaming.gold_rides ADD fare_anomaly text;
```

-> We created the new hotspot layer:

```
>CREATE TABLE taxi_streaming.hotspot_rides (
  agg_date date,
  agg_time timestamp,
  pulocationid int,
  ride_count bigint,
  hotspot_flag text,
  PRIMARY KEY (agg_date, agg_time, pulocationid)
  ) WITH CLUSTERING ORDER BY (agg_time DESC);
```

Now in Cassandra gold_rides table, we have-> avg_fare, ride_count, surge_flag, fare_anomaly, window_start, window_end .
And hotspot_rides include-> pulocationid, ride_count, hotspot_flag, indow_start, window_end, agg_time, agg_date .
We now have built a real-time system that detects demand surges, fare anomalies, and geographic hotspots using streaming data.
—----------

STEP XIV : IMPLEMENTATION OF MINOR CHANGES AND UPDATIONS

1. In the file producer.py,
The Code line: df = df.head(250), is just for testing purposes,
For real-time mimic of data streaming, it should be updated later to:

```
while True:
    for index, row in df.iterrows():
```

2. In the stream_processor.py file:
We update the Kafka Offset setting:
The previous code of line, that was: .option("startingOffsets", "earliest")
Had a problem , and that was that with every restart, it used to process ALL data. This was bad for “production” explanation.
So we replaced it with : .option("startingOffsets", "latest")
Now, this line of code reads only new data, and works correctly with checkpointing.

This ensures the system processes only new incoming data and avoids reprocessing old Kafka messages, making it suitable for production-level streaming systems.
3. The current code has the follow thresholds:

- Surge : ride_count > 5
- Hotspot: ride_count > 10
- Anomaly: avg_fare > 100

Which is perfect to actually see flags with our dataset. Thresholds may vary according to different data. It can be changed according to the dataset and size of batches.
4. The line of code: .

```
>withColumn("pickup_time", to_timestamp("tpep_pickup_datetime"))
```

For both gold_df and hotspot_df, is changed to:

```
>withColumn("pickup_time", col("pickup_datetime"))
```

This results in a cleaner pipeline and avoids redundant parsing.
5. Querying in dashboards becomes easier when we have the actual time grouping info.
So the line of code:

```
>drop("window")
```

Is replaced by:

```
>.withColumn("window_start", col("window.start")) \
 .withColumn("window_end", col("window.end")) \
 .drop("window")
```

In both gold_df and hotspot_df; but we also need to make changes in the tax_streaming tables in cqlsh too for that:
So in cqlsh;

```
>ALTER TABLE taxi_streaming.gold_rides ADD window_start timestamp;
>ALTER TABLE taxi_streaming.gold_rides ADD window_end timestamp;
>ALTER TABLE taxi_streaming.hotspot_rides ADD window_start timestamp;
>ALTER TABLE taxi_streaming.hotspot_rides ADD window_end timestamp;
```

STEP XV: ALL UPDATED CODES TILL NOW

1. docker-compose.yml file:

```yaml
services:

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
```

```yaml
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  cassandra:
    image: cassandra:4.1
    container_name: cassandra
    ports:
      - "9042:9042"
    environment:
      - CASSANDRA_CLUSTER_NAME=TaxiCluster
    volumes:
      - cassandra_data:/var/lib/cassandra

volumes:
  cassandra_data:
```

2. producer.py file:

```python
from kafka import KafkaProducer
import pandas as pd
import json
import time

# Kafka connection
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Load dataset
df = pd.read_parquet('data/yellow_tripdata_2025-01.parquet')
```

```
# Limit rows for testing
df = df.head(250)

topic = "taxi_rides"

for index, row in df.iterrows():

    data = row.to_dict()

    # Convert timestamps to string
    for key, value in data.items():
        if hasattr(value, "isoformat"):
            data[key] = value.isoformat()

    producer.send(topic, value=data)

    print(f"Sent ride {index}")

    time.sleep(0.5) # simulate real-time streaming
```

3. stream_processor.py file:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date
from pyspark.sql.functions import col, from_json, current_timestamp, expr,
avg, count, to_timestamp
from pyspark.sql.types import StructType, StringType, DoubleType,
IntegerType
from pyspark.sql.functions import window

# -------------------- Spark Session --------------------
spark = SparkSession.builder \
    .appName("TaxiStreamingProcessor") \
    .config("spark.sql.shuffle.partitions", "2") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .config("spark.driver.host", "127.0.0.1") \
    .config("spark.sql.streaming.stateStore.providerClass",

"org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvid
er") \
```

```
    .getOrCreate()

spark.catalog.clearCache()
spark.sparkContext.setLogLevel("WARN")

# -------------------- Full schema (all fields from your Kafka JSON)
--------------------
schema = StructType() \
    .add("VendorID", IntegerType()) \
    .add("tpep_pickup_datetime", StringType()) \
    .add("tpep_dropoff_datetime", StringType()) \
    .add("passenger_count", DoubleType()) \
    .add("trip_distance", DoubleType()) \
    .add("RatecodeID", DoubleType()) \
    .add("store_and_fwd_flag", StringType()) \
    .add("PULocationID", IntegerType()) \
    .add("DOLocationID", IntegerType()) \
    .add("payment_type", IntegerType()) \
    .add("fare_amount", DoubleType()) \
    .add("extra", DoubleType()) \
    .add("mta_tax", DoubleType()) \
    .add("tip_amount", DoubleType()) \
    .add("tolls_amount", DoubleType()) \
    .add("improvement_surcharge", DoubleType()) \
    .add("total_amount", DoubleType()) \
    .add("congestion_surcharge", DoubleType()) \
    .add("Airport_fee", DoubleType()) \
    .add("cbd_congestion_fee", DoubleType())

# -------------------- Read Kafka Stream --------------------
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "taxi_rides") \
    .option("startingOffsets", "latest") \
    .load()

json_df = df.selectExpr("CAST(value AS STRING) AS raw_event") \
            .withColumn("ride_id", expr("uuid()"))
# -------------------- BRONZE --------------------
```

```python
bronze_df = json_df.withColumn("created_at", current_timestamp())
bronze_query = bronze_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="bronze_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/bronze") \
    .outputMode("append") \
    .start()

# -------------------- SILVER --------------------
silver_df = json_df.select(
    from_json(col("raw_event"), schema).alias("data"),
    col("ride_id")
).select("ride_id", "data.*")

silver_df = silver_df.select(
    col("ride_id"),
    col("VendorID").alias("vendorid"),
    col("tpep_pickup_datetime"),
    col("tpep_dropoff_datetime"),
    col("passenger_count"),
    col("trip_distance"),
    col("RatecodeID").alias("ratecodeid"),
    col("store_and_fwd_flag"),
    col("PULocationID").alias("pulocationid"),
    col("DOLocationID").alias("dolocationid"),
    col("payment_type"),
    col("fare_amount"),
    col("extra"),
    col("mta_tax"),
    col("tip_amount"),
    col("tolls_amount"),
    col("improvement_surcharge"),
    col("total_amount"),
    col("congestion_surcharge"),
    col("Airport_fee").alias("airport_fee"),
    col("cbd_congestion_fee"),
```

```
    to_timestamp("tpep_pickup_datetime").alias("pickup_datetime"),
    to_timestamp("tpep_dropoff_datetime").alias("dropoff_datetime")
)
# -------------------- GOLD --------------------
from pyspark.sql.functions import when

gold_df = (
    silver_df
    .withColumn("pickup_time", col("pickup_datetime"))
    .withWatermark("pickup_time", "10 minutes")
    .groupBy(window("pickup_time", "1 minute"))
    .agg(
        avg("fare_amount").alias("avg_fare"),
        avg("trip_distance").alias("avg_trip_distance"),
        count("*").alias("ride_count")
    )
    .withColumn(
        "surge_flag",
        when(col("ride_count") > 5, "HIGH_DEMAND").otherwise("NORMAL")
    )
    .withColumn(
        "fare_anomaly",
        when(col("avg_fare") > 100, "ABNORMAL").otherwise("NORMAL")
    )
    .withColumn("agg_time", current_timestamp())
    .withColumn("agg_date", to_date("agg_time"))
    .withColumn("window_start", col("window.start"))
    .withColumn("window_end", col("window.end"))
    .drop("window")
)

hotspot_df = (
    silver_df
    .withColumn("pickup_time", col("pickup_datetime"))
    .withWatermark("pickup_time", "10 minutes")
    .groupBy(
        window("pickup_time", "5 minutes"),
        col("pulocationid")
    )
    .agg(
```

```
        count("*").alias("ride_count")
    )
    .withColumn(
        "hotspot_flag",
        when(col("ride_count") > 10, "HOTSPOT").otherwise("NORMAL")
    )
    .withColumn("agg_time", current_timestamp())
    .withColumn("agg_date", to_date("agg_time"))
    .withColumn("window_start", col("window.start"))
    .withColumn("window_end", col("window.end"))
    .drop("window")
)

gold_query = gold_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="gold_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/gold") \
    .outputMode("update") \
    .start()

hotspot_query = hotspot_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="hotspot_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/hotspot") \
    .outputMode("update") \
    .start()

# -------------------- Await Termination --------------------
spark.streams.awaitAnyTermination()
```

BRIEF OF WHAT WE HAVE DONE TILL NOW

1. We have now created a system that can be used by ride-hailing platforms to detect demand surges, pricing anomalies, and location-based hotspots in real-time for dynamic pricing and driver allocation.

We have:

- ride_count
- avg_fare
- surge detection
- anomaly detection
- hotspot detection

2. How to run project for now:

CASE I : WHEN THERE IS NO CHECKPOINT FOLDER YET:

We delete the checkpoint folder from the main project file taxistream, if it's already present.

1. We go to Task Manager, and end these tasks if they exist:
    - Docker Desktop
    - Docker Desktop Backend
    - com.docker.backend

    Then we start Docker Desktop again, just by clicking on the Docker Desktop icon.

2. Now, in Terminal 1, we run the follow commands:

    >activate taxic

    >cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

    >docker compose down

    >docker compose up -d

    >docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

    (to create a kafka topic, has to be done everytime the docker has been up, after down)

    >docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092

    (to verify whether docker is running or not)

    >set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe

    >**spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py**

    And then basically it'll go up to somewhere like:

```
26/03/16 23:19:58 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 127.0.0.1, 62535, None)
26/03/16 23:19:58 INFO SharedState: Setting hive.metastore.warehouse.dir ('null') to the value of spark.sql.warehouse.di
r.
26/03/16 23:19:58 INFO SharedState: Warehouse path is 'file:/C:/Users/rajve/Desktop/SkillIssues/Personal%20Projects/taxi
stream/spark-warehouse'.
26/03/16 23:20:03 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Dataset
s and will be disabled.
26/03/16 23:20:04 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commi
t, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
26/03/16 23:20:04 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Dataset
s and will be disabled.
26/03/16 23:20:04 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commi
t, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
```

    ……..

    4. Meanwhile in Terminal 2:

    >activate taxic

```
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>python producer/producer.py
```

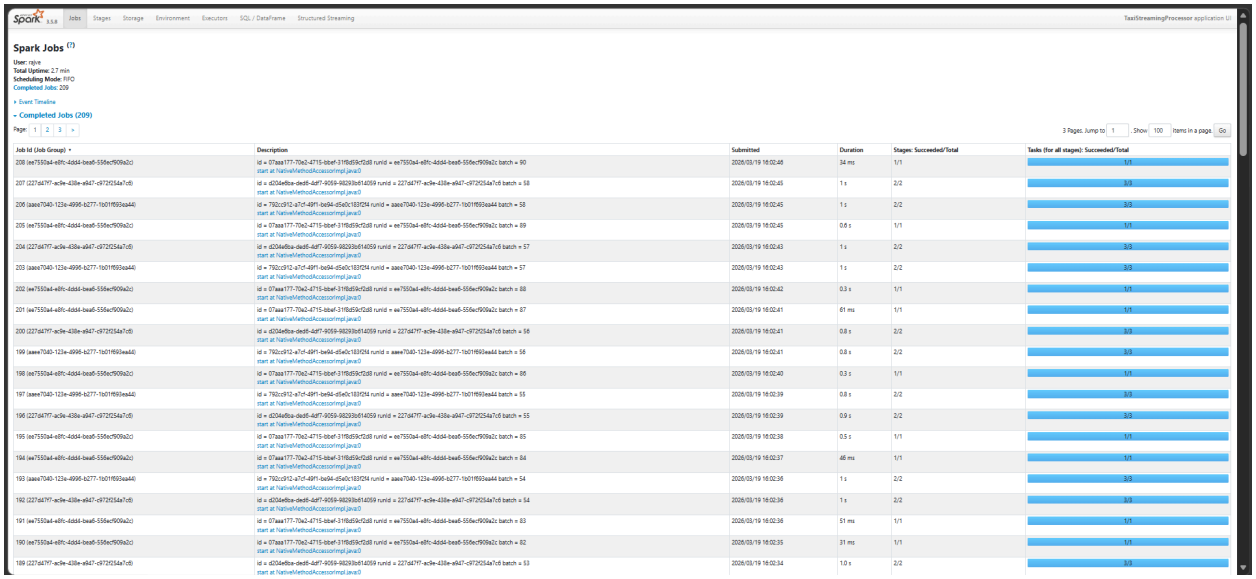
It'll send rides, this time till 250, since we set that. Later on we should put a while loop for continuous streaming of data from the dataset.

……

To Verify, We go to local host:

http://localhost:4040/jobs/ ([TaxiStreamingProcessor - Spark Jobs]())

Here, we'll be able to see the Spark UI:



For 250 rides sent, 209 jobs were completed.

This good as one batch of work can have more than 1 ride, or sometimes 2 or 3 too.

We can also check whether Cassandra is giving results or not, by going to the cqlsh shell. This is also to be done in the terminal 2, as terminal 1 is still running spark.

```
>docker exec -t cassandra cqlsh
cqlsh> SELECT ride_count, surge_flag, fare_anomaly, window_start FROM
        taxi_streaming.gold_rides LIMIT 10;
cqlsh> SELECT pulocationid, ride_count, hotspot_flag, window_start FROM
        taxi_streaming.hotspot_rides LIMIT 10;
cqlsh> SELECT window_start, window_end FROM taxi_streaming.gold_rides LIMIT 5;
```

```
cqlsh> SELECT ride_count, surge_flag, fare_anomaly, window_start FROM taxi_streaming.gold_rides LIMIT 10;

 ride_count | surge_flag  | fare_anomaly | window_start
------------+-------------+--------------+---------------------------------
          3 |      NORMAL |       NORMAL | 2024-12-31 19:28:00.000000+0000
          6 | HIGH_DEMAND |       NORMAL | 2024-12-31 19:23:00.000000+0000
          4 |      NORMAL |       NORMAL | 2024-12-31 19:24:00.000000+0000
          5 |      NORMAL |       NORMAL | 2024-12-31 19:23:00.000000+0000
          2 |      NORMAL |       NORMAL | 2024-12-31 19:28:00.000000+0000
          1 |      NORMAL |       NORMAL | 2024-12-31 19:31:00.000000+0000
          3 |      NORMAL |       NORMAL | 2024-12-31 19:24:00.000000+0000
          4 |      NORMAL |       NORMAL | 2024-12-31 19:23:00.000000+0000
          4 |      NORMAL |       NORMAL | 2024-12-31 19:22:00.000000+0000
          1 |      NORMAL |       NORMAL | 2024-12-31 19:28:00.000000+0000

(10 rows)
cqlsh> SELECT pulocationid, ride_count, hotspot_flag, window_start FROM taxi_streaming.hotspot_rides LIMIT 10;

 pulocationid | ride_count | hotspot_flag | window_start
--------------+------------+--------------+---------------------------------
           68 |          1 |       NORMAL | 2024-12-31 19:25:00.000000+0000
          107 |          1 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          142 |          2 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          237 |          3 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          263 |          1 |       NORMAL | 2024-12-31 19:25:00.000000+0000
          261 |          1 |       NORMAL | 2024-12-31 19:30:00.000000+0000
          229 |          2 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          229 |          1 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          263 |          2 |       NORMAL | 2024-12-31 19:20:00.000000+0000
          249 |          2 |       NORMAL | 2024-12-31 19:20:00.000000+0000
```

```
cqlsh> SELECT window_start, window_end FROM taxi_streaming.gold_rides LIMIT 5;

 window_start                     | window_end
----------------------------------+----------------------------------
 2024-12-31 19:28:00.000000+0000 | 2024-12-31 19:29:00.000000+0000
 2024-12-31 19:23:00.000000+0000 | 2024-12-31 19:24:00.000000+0000
 2024-12-31 19:24:00.000000+0000 | 2024-12-31 19:25:00.000000+0000
 2024-12-31 19:23:00.000000+0000 | 2024-12-31 19:24:00.000000+0000
 2024-12-31 19:28:00.000000+0000 | 2024-12-31 19:29:00.000000+0000

(5 rows)
```

cqlsh> SELECT window_start, ride_count FROM taxi_streaming.gold_rides WHERE surge_flag = 'HIGH_DEMAND' ALLOW FILTERING;

```
cqlsh> SELECT window_start, ride_count FROM taxi_streaming.gold_rides WHERE surge_flag = 'HIGH_DEMAND' ALLOW FILTERING;

 window_start                     | ride_count
----------------------------------+------------
 2024-12-31 19:23:00.000000+0000 |          6
 2024-12-31 19:25:00.000000+0000 |          7
 2024-12-31 19:25:00.000000+0000 |          7
 2024-12-31 19:25:00.000000+0000 |          6
```

cqlsh> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;

```
cqlsh> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;

 agg_date   | agg_time                        | avg_fare | avg_trip_distance | fare_anomaly | ride_count | surge_flag   | wind
ow_end                        | window_start
------------+---------------------------------+----------+-------------------+--------------+------------+--------------+-----
-----------------------------+---------------------------------
 2026-03-19 | 2026-03-19 10:32:40.438000+0000 | 16.06667 |              2.16 |       NORMAL |          3 |       NORMAL | 2024
-12-31 19:29:00.000000+0000 | 2024-12-31 19:28:00.000000+0000
 2026-03-19 | 2026-03-19 10:32:38.350000+0000 |     11.4 |           1.59667 |       NORMAL |          6 | HIGH_DEMAND | 2024
-12-31 19:24:00.000000+0000 | 2024-12-31 19:23:00.000000+0000
 2026-03-19 | 2026-03-19 10:32:32.139000+0000 |     14.9 |               1.7 |       NORMAL |          4 |       NORMAL | 2024
-12-31 19:25:00.000000+0000 | 2024-12-31 19:24:00.000000+0000
 2026-03-19 | 2026-03-19 10:32:24.030000+0000 |    11.26 |             1.628 |       NORMAL |          5 |       NORMAL | 2024
-12-31 19:24:00.000000+0000 | 2024-12-31 19:23:00.000000+0000
 2026-03-19 | 2026-03-19 10:32:19.880000+0000 |    13.15 |              1.87 |       NORMAL |          2 |       NORMAL | 2024
-12-31 19:29:00.000000+0000 | 2024-12-31 19:28:00.000000+0000

(5 rows)
```

If ever asked: Why are we using ‘ALOOW FILTERING’
We say: Cassandra is optimized for query-based modeling. Since this is an exploratory/demo setup, I used ALLOW FILTERING. In production, I would redesign the table based on query patterns.

CASE II : WHEN THE CHECKPOINT FOLDER IS ALREADY THERE, AND DOCKER WAS NEVER DONE DOWN:

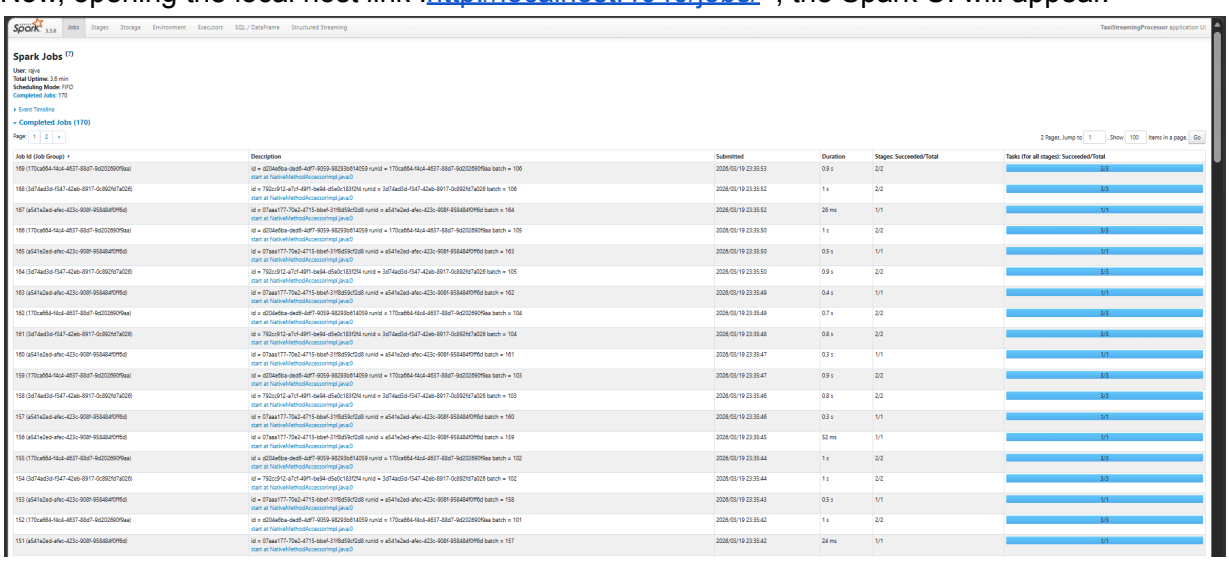
If docker wasn’t done down and the checkpoints were not deleted.
In terminal 1:

>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
>**spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py**

Meanwhile, in terminal 2:

>activate taxic
>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
>python producer/producer.py

Now, opening the local host link :http://localhost:4040/jobs/ , the Spark UI will appear.



-> We notice that now jobs are 170, this is because Spark is resuming from checkpoints and Kafka offsets, not reprocessing everything again.
-> Spark Structured Streaming uses checkpointing and Kafka offset tracking to ensure fault tolerance and exactly-once processing. On restart, it resumes from the last processed offset instead of reprocessing the entire stream, resulting in fewer jobs.
-> Even if the same dataset is sent again, Kafka treats each record as a new event with a new offset. Hence, Spark processes it as new streaming data.
-> But then the question may arise: Why even 170, why not 0? Because literally we gave in the same 250 rides before, so shouldn't it already have been in the checkpoints?
-> The answer is that Spark doesn't track "records". It tracks offsets + state + time windows.
-> So when we ran the producer.py file, Even though it's the same dataset, this line:
>producer.send(topic, value=data) ; sends new messages to Kafka.
-> Kafka sees this as: NEW MESSAGE → NEW OFFSET
NOT: same message → skip
-> The second reason for this is the: Checkpoint state reuse
Spark already knows:

- Window aggregations
- Partial computations
- Previous state

So, it processes faster + fewer batches
-> The third reason is due to Micro-batch timing differences. The time.sleep(0.5), is never identical. So batch grouping may change slightly, and the number of jobs may change.

BRIEF OF WHAT HAS TO BE DONE NOW:

1. Grafana Dashboard
   - Connect Grafana to Cassandra
   - Visualize:
     - ride_count over time
     - surge_flag trends
     - hotspot locations
     - avg_fare trends
     - Create meaningful panels
     - average trip distance
2. Airflow Integration
   - Schedule producer and Spark jobs
   - Manage pipeline execution
   - Create DAG

STEP XVI: MAJOR FIXES AND UPDATES

When we came back after a lot of days, we saw that there were '2K+' changes for git, which weren't to be pushed (because they were from the checkpoints folder, containing `commits/`, `offsets/`, `sources/`, `state/`).
Anyways, we then added a '.gitignore' folder so that things not to be pushed aren't pushed.

So here is the added .gitignore file:

```
# Spark streaming checkpoints (runtime state, not code)
checkpoints/

# Dataset files (too large / not source code)
data/*.parquet

# Python
__pycache__/
*.pyc
.env

# Airflow
airflow/logs/
airflow/*.db
airflow/airflow.cfg

# OS/editor junk
.DS_Store
.vscode/
.env
```

And if that doesn't work, we can explicitly untrack them:

-> git rm -r --cached checkpoints/
-> git rm --cached data/yellow_tripdata_2025-01.parquet
-> git add .gitignore
-> git commit -m "Add .gitignore, stop tracking checkpoints and dataset"

STEP XVII: LOADING THE PROJECT

-> First we go to our Code Base and delete the complete 'Checkpoints' folder.
-> Then we go to Task Manager, and end these tasks if they exist:

- Docker Desktop
- Docker Desktop Backend
- com.docker.backend

Then we start Docker Desktop again, just by clicking on the Docker Desktop icon.
(Using Anaconda Prompt Terminal)

Terminal 1: Docker Infra

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> docker compose down
-> docker compose up -d
-> docker ps

-> docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
-> docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
-> docker exec -it cassandra cqlsh (enter ‘exit’ to exit)
-> DESCRIBE KEYSPACES;
-> USE taxi_streaming;
-> DESCRIBE TABLES;

Terminal 2: Spark
-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
-> spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py

Terminal 3: Producer
-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> python producer/producer.py

[Note: **CASE II : WHEN THE CHECKPOINT FOLDER IS ALREADY THERE, AND DOCKER WAS NEVER DONE DOWN:**
Check Page 48]

So this is how terminal 1 should be looking:

```
(base) C:\Users\rajve>activate taxic

(taxic) C:\Users\rajve>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose down

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose up -d
[+] up 5/5
 ✔Network taxistream_default Created                                                                    0.0s
 ✔Container cassandra        Started                                                                    0.8s
 ✔Container zookeeper        Started                                                                    0.8s
 ✔Container grafana          Started                                                                    0.9s
 ✔Container kafka            Started                                                                    0.9s

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
2a1cd66b3353   confluentinc/cp-kafka:7.5.0        "/etc/confluent/dock…"   13 seconds ago   Up 11 seconds   0.0.0.0:9092->9092/tcp, [::]:9092->9092/tcp            kafka
fb70935dda46   grafana/grafana:11.2.0             "/run.sh"                13 seconds ago   Up 11 seconds   0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp            grafana
139373c6d7e5   cassandra:4.1                      "docker-entrypoint.s…"   13 seconds ago   Up 11 seconds   0.0.0.0:9042->9042/tcp, [::]:9042->9042/tcp            cassandra
50a841b04821   confluentinc/cp-zookeeper:7.5.0    "/etc/confluent/dock…"   13 seconds ago   Up 11 seconds   0.0.0.0:2181->2181/tcp, [::]:2181->2181/tcp            zookeeper

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.
Created topic taxi_rides.

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
taxi_rides

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it cassandra cqlsh
Connected to TaxiCluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.10 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> DESCRIBE KEYSPACES;

my_keyspace  system_distributed  system_views
system       system_schema       system_virtual_schema
system_auth  system_traces       taxi_streaming

cqlsh> USE taxi_streaming;
cqlsh:taxi_streaming> DESCRIBE TABLES;

bronze_rides  dlq_rides  gold_rides  hotspot_rides  silver_rides

cqlsh:taxi_streaming> |
```

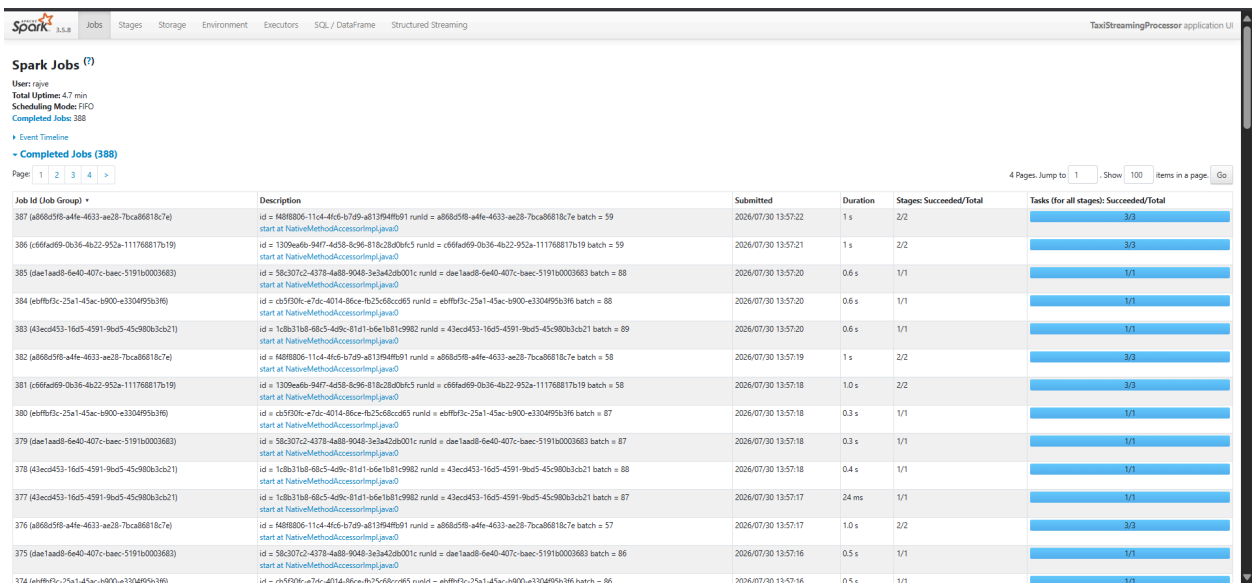
This is how terminal 2 should be looking (at the end):

```
26/07/30 13:53:38 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets and will be disabled.
26/07/30 13:53:38 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commit, max.poll.records, auto.offset.res
et]' were supplied but are not used yet.
26/07/30 13:53:39 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commit, max.poll.records, auto.offset.res
et]' were supplied but are not used yet.
```

This is how terminal 3 should look like: (rides being sent to whatever is set in producer, for now 249)

```
(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>python producer/producer.py
Sent ride 0
Sent ride 1
Sent ride 2
Sent ride 3
Sent ride 4
Sent ride 5
Sent ride 6
Sent ride 7
Sent ride 8
Sent ride 9
Sent ride 10
Sent ride 11
Sent ride 12
Sent ride 13
Sent ride 14
Sent ride 15
Sent ride 16
Sent ride 17
Sent ride 18
Sent ride 19
```

Then when we go to http://localhost:4040/jobs/ in the browser , this is how it should look like:



Now, if we go back to terminal 1, and run something like:
SELECT * FROM taxi_streaming.gold_rides LIMIT 5;
We should be able to see `avg_fare`, `ride_count`, `surge_flag` .

```
cqlsh> USE taxi_streaming;
cqlsh:taxi_streaming> DESCRIBE TABLES;

bronze_rides  dlq_rides  gold_rides  hotspot_rides  silver_rides

cqlsh:taxi_streaming> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;

 agg_date   | agg_time                        | avg_fare | avg_trip_distance | fare_anomaly | ride_count | surge_flag  | window_end                      | window_start
------------+---------------------------------+----------+-------------------+--------------+------------+-------------+---------------------------------+---------------------------------
 2026-07-30 | 2026-07-30 08:27:16.349000+0000 | 16.06667 |              2.16 |       NORMAL |          3 |      NORMAL | 2024-12-31 19:29:00.000000+0000 | 2024-12-31 19:28:00.000000+0000
 2026-07-30 | 2026-07-30 08:27:12.464000+0000 |    11.4  |           1.59667 |       NORMAL |          6 | HIGH_DEMAND | 2024-12-31 19:24:00.000000+0000 | 2024-12-31 19:23:00.000000+0000
 2026-07-30 | 2026-07-30 08:27:06.111000+0000 |    14.9  |               1.7 |       NORMAL |          4 |      NORMAL | 2024-12-31 19:25:00.000000+0000 | 2024-12-31 19:24:00.000000+0000
 2026-07-30 | 2026-07-30 08:26:59.548000+0000 |    11.26 |             1.628 |       NORMAL |          5 |      NORMAL | 2024-12-31 19:24:00.000000+0000 | 2024-12-31 19:23:00.000000+0000
 2026-07-30 | 2026-07-30 08:26:55.479000+0000 |    13.15 |              1.87 |       NORMAL |          2 |      NORMAL | 2024-12-31 19:29:00.000000+0000 | 2024-12-31 19:28:00.000000+0000
```

STEP XVIII: Implementing DLQ (Dead-Letter-Qeue)

So, first of all, here are all the updated codes:

1. **producer.py:**

```
from kafka import KafkaProducer
import pandas as pd
import json
import time

# Kafka connection
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Load dataset
df = pd.read_parquet('data/yellow_tripdata_2025-01.parquet')

# Limit rows for testing
df = df.head(250)

topic = "taxi_rides"

for index, row in df.iterrows():

    data = row.to_dict()

    # Convert timestamps to string
    for key, value in data.items():
        if hasattr(value, "isoformat"):
            data[key] = value.isoformat()

    producer.send(topic, value=data)

    print(f"Sent ride {index}")

    time.sleep(0.5) # simulate real-time streaming
```

2. **Stream_processor.py:**

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date
```

```
from pyspark.sql.functions import col, from_json, current_timestamp, expr,
avg, count, to_timestamp, when, to_json, struct, concat_ws
from pyspark.sql.types import StructType, StringType, DoubleType,
IntegerType
from pyspark.sql.functions import window

# -------------------- Spark Session --------------------
spark = SparkSession.builder \
    .appName("TaxiStreamingProcessor") \
    .config("spark.sql.shuffle.partitions", "2") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .config("spark.driver.host", "127.0.0.1") \
    .config("spark.sql.streaming.stateStore.providerClass",

"org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvid
er") \
    .getOrCreate()

spark.catalog.clearCache()
spark.sparkContext.setLogLevel("WARN")

# -------------------- Full schema (all fields from your Kafka JSON)
--------------------
schema = StructType() \
    .add("VendorID", IntegerType()) \
    .add("tpep_pickup_datetime", StringType()) \
    .add("tpep_dropoff_datetime", StringType()) \
    .add("passenger_count", DoubleType()) \
    .add("trip_distance", DoubleType()) \
    .add("RatecodeID", DoubleType()) \
    .add("store_and_fwd_flag", StringType()) \
    .add("PULocationID", IntegerType()) \
    .add("DOLocationID", IntegerType()) \
    .add("payment_type", IntegerType()) \
    .add("fare_amount", DoubleType()) \
    .add("extra", DoubleType()) \
    .add("mta_tax", DoubleType()) \
    .add("tip_amount", DoubleType()) \
    .add("tolls_amount", DoubleType()) \
    .add("improvement_surcharge", DoubleType()) \
```

```
    .add("total_amount", DoubleType()) \
    .add("congestion_surcharge", DoubleType()) \
    .add("Airport_fee", DoubleType()) \
    .add("cbd_congestion_fee", DoubleType())

# -------------------- Read Kafka Stream --------------------
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "taxi_rides") \
    .option("startingOffsets", "latest") \
    .load()

json_df = df.selectExpr("CAST(value AS STRING) AS raw_event") \
            .withColumn("ride_id", expr("uuid()"))
# -------------------- BRONZE --------------------
bronze_df = json_df.withColumn("created_at", current_timestamp())
bronze_query = bronze_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="bronze_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/bronze") \
    .outputMode("append") \
    .start()


# -------------------- PARSE + VALIDATE --------------------
parsed_df = json_df.select(
    from_json(col("raw_event"), schema).alias("data"),
    col("ride_id"),
    col("raw_event")
).select("ride_id", "raw_event", "data.*")

validated_df = parsed_df.withColumn(
    "validation_errors",
    concat_ws(", ",
```

```
        when(col("fare_amount").isNull() | (col("fare_amount") < 0),
"invalid_fare"),
        when(col("trip_distance").isNull() | (col("trip_distance") < 0),
"invalid_distance"),
        when(col("tpep_pickup_datetime").isNull(), "missing_pickup_time"),
        when(col("tpep_dropoff_datetime").isNull(),
"missing_dropoff_time"),
        when(col("PULocationID").isNull(), "missing_pu_location"),
        when(col("DOLocationID").isNull(), "missing_do_location")
    )
).withColumn("is_valid", col("validation_errors") == "")

valid_df = validated_df.filter(col("is_valid")).drop("is_valid",
"validation_errors")
invalid_df = validated_df.filter(~col("is_valid"))

# -------------------- SILVER (from valid records only)
--------------------
silver_df = valid_df.select(
    col("ride_id"),
    col("VendorID").alias("vendorid"),
    col("tpep_pickup_datetime"),
    col("tpep_dropoff_datetime"),
    col("passenger_count"),
    col("trip_distance"),
    col("RatecodeID").alias("ratecodeid"),
    col("store_and_fwd_flag"),
    col("PULocationID").alias("pulocationid"),
    col("DOLocationID").alias("dolocationid"),
    col("payment_type"),
    col("fare_amount"),
    col("extra"),
    col("mta_tax"),
    col("tip_amount"),
    col("tolls_amount"),
    col("improvement_surcharge"),
    col("total_amount"),
    col("congestion_surcharge"),
    col("Airport_fee").alias("airport_fee"),
    col("cbd_congestion_fee"),
```

```
    to_timestamp("tpep_pickup_datetime").alias("pickup_datetime"),
    to_timestamp("tpep_dropoff_datetime").alias("dropoff_datetime")
)

# -------------------- DLQ SINK 1: Kafka topic --------------------
dlq_kafka_df = invalid_df.select(
    col("ride_id").alias("key"),
    to_json(struct(col("ride_id"), col("validation_errors"),
col("raw_event"))).alias("value")
)

dlq_kafka_query = dlq_kafka_df.writeStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "taxi_rides_dlq") \
    .option("checkpointLocation", "checkpoints/dlq_kafka") \
    .outputMode("append") \
    .start()

# -------------------- DLQ SINK 2: Cassandra (for easy demo/verification)
--------------------
dlq_cassandra_df = invalid_df.select(
    col("ride_id"),
    col("validation_errors").alias("reason"),
    col("raw_event"),
    current_timestamp().alias("rejected_at")
)

dlq_cassandra_query = dlq_cassandra_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="dlq_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/dlq_cassandra") \
    .outputMode("append") \
    .start()
# -------------------- GOLD --------------------
```

```
gold_df = (
    silver_df
    .withColumn("pickup_time", col("pickup_datetime"))
    .withWatermark("pickup_time", "10 minutes")
    .groupBy(window("pickup_time", "1 minute"))
    .agg(
        avg("fare_amount").alias("avg_fare"),
        avg("trip_distance").alias("avg_trip_distance"),
        count("*").alias("ride_count")
    )
    .withColumn(
        "surge_flag",
        when(col("ride_count") > 5, "HIGH_DEMAND").otherwise("NORMAL")
    )
    .withColumn(
        "fare_anomaly",
        when(col("avg_fare") > 100, "ABNORMAL").otherwise("NORMAL")
    )
    .withColumn("agg_time", current_timestamp())
    .withColumn("agg_date", to_date("agg_time"))
    .withColumn("window_start", col("window.start"))
    .withColumn("window_end", col("window.end"))
    .drop("window")
)

hotspot_df = (
    silver_df
    .withColumn("pickup_time", col("pickup_datetime"))
    .withWatermark("pickup_time", "10 minutes")
    .groupBy(
        window("pickup_time", "5 minutes"),
        col("pulocationid")
    )
    .agg(
        count("*").alias("ride_count")
    )
    .withColumn(
        "hotspot_flag",
        when(col("ride_count") > 10, "HOTSPOT").otherwise("NORMAL")
    )
```

```
    .withColumn("agg_time", current_timestamp())
    .withColumn("agg_date", to_date("agg_time"))
    .withColumn("window_start", col("window.start"))
    .withColumn("window_end", col("window.end"))
    .drop("window")
)

gold_query = gold_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="gold_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/gold") \
    .outputMode("update") \
    .start()

hotspot_query = hotspot_df.writeStream \
    .foreachBatch(lambda batch_df, _: batch_df.write
        .format("org.apache.spark.sql.cassandra")
        .options(table="hotspot_rides", keyspace="taxi_streaming")
        .mode("append")
        .save()
    ) \
    .option("checkpointLocation", "checkpoints/hotspot") \
    .outputMode("update") \
    .start()

# -------------------- Await Termination --------------------
spark.streams.awaitAnyTermination()
```

3. .env:

```
GRAFANA_ADMIN_USER=admin
GRAFANA_ADMIN_PASSWORD={whatever we want to put here}
```

4. .gitignore:

```
# Spark streaming checkpoints (runtime state, not code)
```

```
checkpoints/

# Dataset files (too large / not source code)
data/*.parquet

# Python
__pycache__/
*.pyc
.env

# Airflow
airflow/logs/
airflow/*.db
airflow/airflow.cfg

# OS/editor junk
.DS_Store
.vscode/
.env
```

5. Docker-compose.yml:

```
services:

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
```

```
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  cassandra:
    image: cassandra:4.1
    container_name: cassandra
    ports:
      - "9042:9042"
    environment:
      - CASSANDRA_CLUSTER_NAME=TaxiCluster
    volumes:
      - cassandra_data:/var/lib/cassandra

  grafana:
    image: grafana/grafana:11.2.0
    container_name: grafana
    depends_on:
      - cassandra
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_USER=${GRAFANA_ADMIN_USER}
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWORD}
      - GF_INSTALL_PLUGINS=hadesarchitect-cassandra-datasource
    volumes:
      - grafana_data:/var/lib/grafana

volumes:
  cassandra_data:
  grafana_data:
```

6. Requirements.txt:

```
# packages in environment at C:\Users\rajve\anaconda3\envs\taxic:
#
# Name                    Version          Build            Channel
bzip2                     1.0.8            h2bbff1b_6
ca-certificates           2025.12.2        haa95532_0
cassandra-driver          3.29.3           pypi_0           pypi
click                     8.3.1            pypi_0           pypi
```

```
colorama                  0.4.6            pypi_0           pypi
expat                     2.7.4            hd7fb8db_0
findspark                 2.0.1            pypi_0           pypi
geomet                    1.1.0            pypi_0           pypi
kafka-python              2.3.0            pypi_0           pypi
libexpat                  2.7.4            hd7fb8db_0
libffi                    3.4.4            hd77b12b_1
libzlib                   1.3.1            h02ab6af_0
numpy                     2.2.6            pypi_0           pypi
openssl                   3.5.5            hbb43b14_0
packaging                 25.0             py310haa95532_1
pandas                    2.3.3            pypi_0           pypi
pip                       26.0.1           pyhc872135_0
py4j                      0.10.9.9         pypi_0           pypi
pyarrow                   23.0.1           pypi_0           pypi
pyspark                   4.1.1            pypi_0           pypi
python                    3.10.19          h981015d_0
python-dateutil           2.9.0.post0      pypi_0           pypi
pytz                      2026.1.post1     pypi_0           pypi
setuptools                80.10.2          py310haa95532_0
six                       1.17.0           pypi_0           pypi
sqlite                    3.51.1           hee5a0db_1
tk                        8.6.15           hf199647_0
tzdata                    2025.3           pypi_0           pypi
ucrt                      10.0.22621.0     haa95532_0
vc                        14.3             h2df5915_10
vc14_runtime              14.44.35208      h4927774_10
vs2015_runtime            14.44.35208      ha6b5a95_10
wheel                     0.46.3           py310haa95532_0
xz                        5.8.2            h53af0af_0
zlib                      1.3.1            h02ab6af_0
```

DLQ Implementation:
Design: We validate parsed records in Spark. Valid rows flow through your existing Bronze→Silver→Gold pipeline unchanged. Invalid rows get routed to (a) a new Kafka topic `taxi_rides_dlq` and, (b) mirrored into a Cassandra `dlq_rides` table just so we can verify/demo it with a simple `cqlsh` query instead of running a Kafka consumer every time.

1. Creating the DLQ Kafka Topic:
   -> docker exec -it kafka kafka-topics --create --topic taxi_rides_dlq --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
2. Creating the Cassandra DLQ table:
   -> docker exec -it cassandra cqlsh
   -> USE taxi_streaming;
   -> CREATE TABLE dlq_rides (
      ride_id text PRIMARY KEY,
      reason text,
      raw_event text,
      rejected_at timestamp
      );
3. Update stream_processor.py: (updated code already put up)

STEP XIX: Running the Project with DLQ Implemented

-> First we go to our Code Base and delete the complete 'Checkpoints' folder.
-> Then we go to Task Manager, and end these tasks if they exist:

- Docker Desktop
- Docker Desktop Backend
- com.docker.backend

Then we start Docker Desktop again, just by clicking on the Docker Desktop icon.
(Using Anaconda Prompt Terminal)

Terminal 1: Docker Infra

```
-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> docker compose down
-> docker compose up -d
-> docker ps

-> docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
-> docker exec -it kafka kafka-topics --create --topic taxi_rides_dlq --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
-> docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
```

(We will run cqlsh after terminal 3: when all rides are sent)

```
(base) C:\Users\rajve>activate taxic

(taxic) C:\Users\rajve>cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose down
[+] down 5/5
 ✔Container kafka            Removed                                                                    1.4s
 ✔Container grafana          Removed                                                                    0.5s
 ✔Container cassandra        Removed                                                                    1.5s
 ✔Container zookeeper        Removed                                                                    0.7s
 ✔Network taxistream_default Removed                                                                    0.2s

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker compose up -d
[+] up 5/5
 ✔Network taxistream_default Created                                                                    0.0s
 ✔Container cassandra        Started                                                                    0.7s
 ✔Container zookeeper        Started                                                                    0.7s
 ✔Container grafana          Started                                                                    0.8s
 ✔Container kafka            Started                                                                    0.8s

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS                                          NAMES
3169a6303add   confluentinc/cp-kafka:7.5.0    "/etc/confluent/dock…"   9 seconds ago   Up 7 seconds   0.0.0.0:9092->9092/tcp, [::]:9092->9092/tcp    kafka
2ce0e6fe72de   grafana/grafana:11.2.0         "/run.sh"                9 seconds ago   Up 7 seconds   0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp    grafana
392bec7b0eb1   cassandra:4.1                  "docker-entrypoint.s…"   9 seconds ago   Up 8 seconds   0.0.0.0:9042->9042/tcp, [::]:9042->9042/tcp    cassandra
b7a45103a6e9   confluentinc/cp-zookeeper:7.5.0 "/etc/confluent/dock…"   9 seconds ago   Up 8 seconds   0.0.0.0:2181->2181/tcp, [::]:2181->2181/tcp    zookeeper

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 -
-partitions 1 --replication-factor 1
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.
Created topic taxi_rides.

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --create --topic taxi_rides_dlq --bootstrap-server localhost:90
92 --partitions 1 --replication-factor 1
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.
Created topic taxi_rides_dlq.

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
taxi_rides
taxi_rides_dlq

(taxic) C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream>
```

Before running Terminal 2, if we use cqlsh, we'll find that it shows old rows. To remove them, we can run these in cqlsh:
-> TRUNCATE dlq_rides;
-> TRUNCATE gold_rides;
-> TRUNCATE hotspot_rides;
-> TRUNCATE bronze_rides;
We can run these in cqlsh before starting a fresh pipeline run, so our query results only reflect that specific session

Terminal 2: Spark

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
-> spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py

Terminal 3: Producer

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> python producer/producer.py

Terminal 4: DLQ Kafka Verification (optional, on-demand)

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> docker exec -it kafka kafka-console-consumer --topic taxi_rides_dlq --from-beginning --bootstrap-server localhost:9092

Back To Terminal 1:
-> docker exec -it cassandra cqlsh (enter ‘exit’ to exit)
  -> DESCRIBE KEYSPACES;
  -> USE taxi_streaming;
  -> DESCRIBE TABLES;
Now we can run queries such as:
-> SELECT * FROM dlq_rides LIMIT 5;
-> SELECT * FROM gold_rides LIMIT 5;
-> SELECT * FROM hotspot_rides LIMIT 5;
-> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;
-> SELECT * FROM taxi_streaming.dlq_rides LIMIT 5;

Then we can do to: http://localhost:4040/jobs/ to check Spark Jobs

---

STEP XX: Implementing GRAFANA

Step 1. Updated docker-compose.yml with grafana (already updated in Step XIX)
This part:

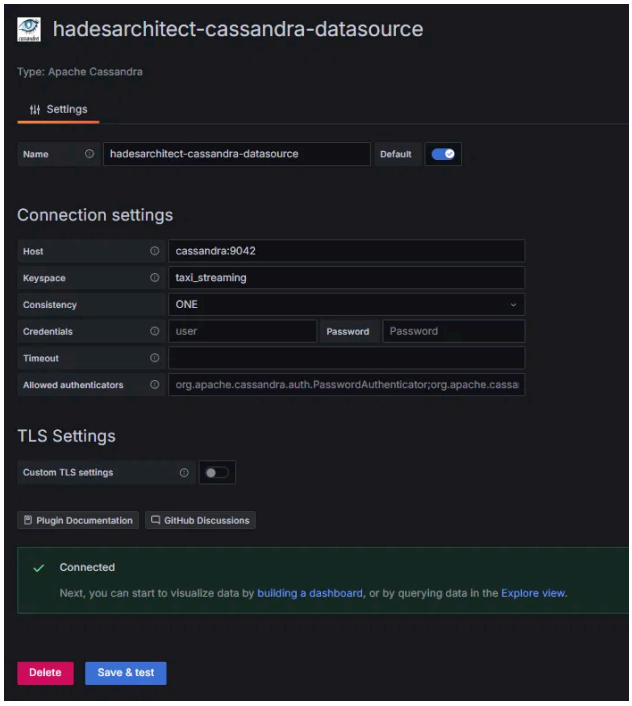
```
  grafana:
    image: grafana/grafana:11.2.0
    container_name: grafana
    depends_on:
      - cassandra
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_USER=${GRAFANA_ADMIN_USER}
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWORD}
      - GF_INSTALL_PLUGINS=hadesarchitect-cassandra-datasource
    volumes:
      - grafana_data:/var/lib/grafana

volumes:
  cassandra_data:
  grafana_data:
```

Step 2: We go to: http://localhost:3000 for Grafana:
And Enter our Username and Password

Step 3: We add Cassandra as a data source

- Left sidebar → Connections → Data sources → Add data source
- Search for and select Cassandra

- Fill in:
    - Host: cassandra:9042 (this works because Grafana and Cassandra are on the same Docker network - cassandra is the container name acting as hostname)
    - Keyspace: taxi_streaming
    - Leave auth blank (no auth configured on your Cassandra container)
- Click Save & Test — it should say it connected successfully.



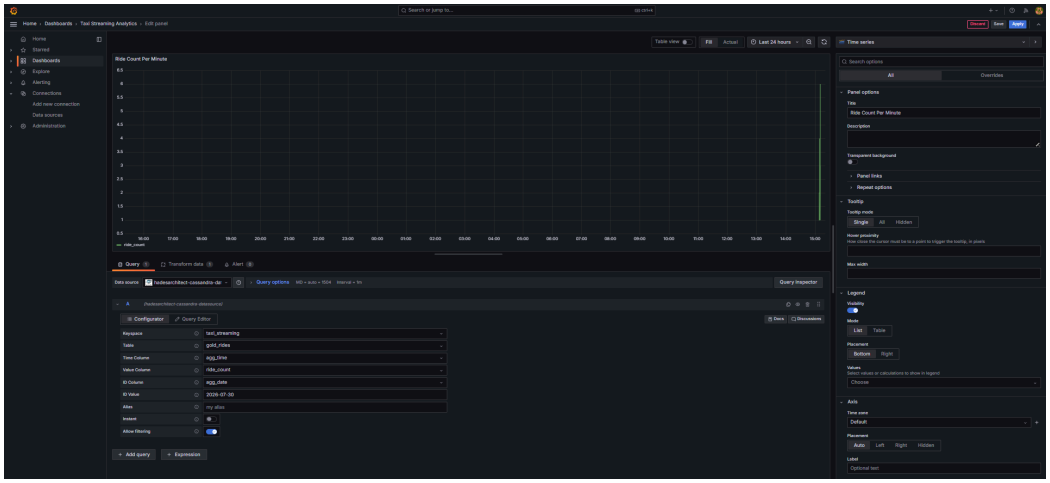
Step 3: Now to create our Dashboards:

-> Left sidebar → Dashboards → New → New Dashboard

-> We Click Add visualization

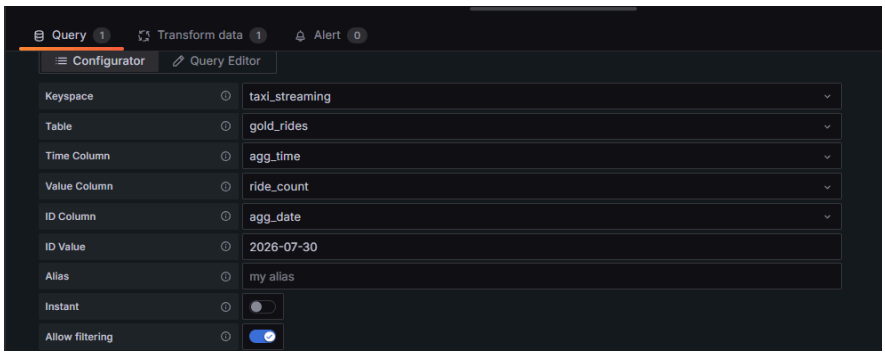
-> We Select our `hadesarchitect-cassandra-datasource` as the data source

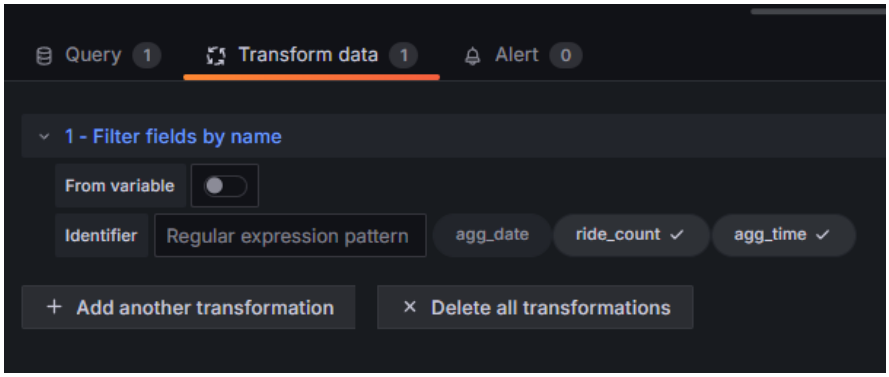
This is how making a dashboard looks like:

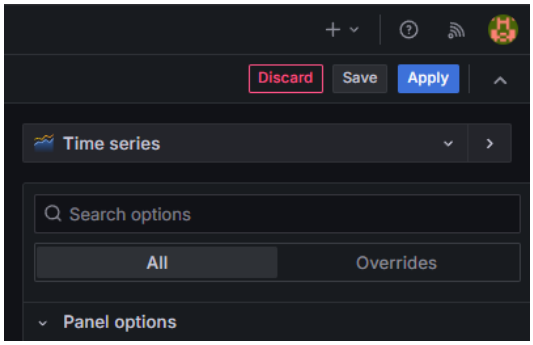


Panel 1: Ride Count Per Minute

- Purpose: Show ride volume trend over time
- Configurator tab:
  - Keyspace: `taxi_streaming` | Table: `gold_rides`
  - Time Column: `agg_time` | Value Column: `ride_count`
  - ID Column: `agg_date` | ID Value: (today's date, e.g. `2026-07-30`)
  - Allow filtering: ON
- Transform Data: Added "Filter fields by name" → deselected `agg_date`, kept `ride_count` + `agg_time` checked (agg_date is a constant-value time field that breaks the x-axis if left in)
- Visualization: Time series







Panel 2: Average Fare Trend

- **Purpose:** Show avg fare fluctuation over time
- Same as Panel 1, except **Value Column:** `avg_fare`
- Same Transform (deselect `agg_date`)
- Visualization: Time series

Panel 3: Surge Events (High Demand)

- **Purpose:** Show only time windows flagged as high-demand surges
- **Query Editor tab** (raw CQL, not Configurator — needed for multiple columns):

SELECT agg_time, ride_count, surge_flag FROM gold_rides WHERE agg_date = '2026-07-30' AND surge_flag = 'HIGH_DEMAND' ALLOW FILTERING

- No transform needed
- Visualization: **Table**

Panel 4: Pickup Hotspots

- Purpose: Show ride density per pickup location
- Query Editor tab:

SELECT agg_time, pulocationid, ride_count, hotspot_flag FROM hotspot_rides WHERE agg_date = '2026-07-30' ALLOW FILTERING

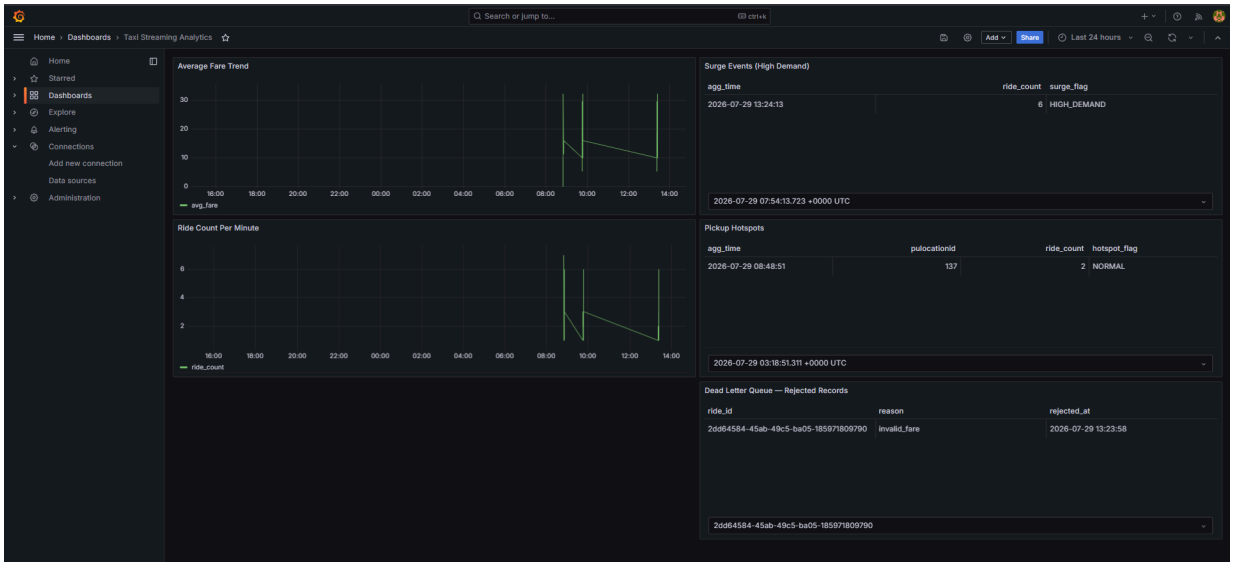
- No transform needed
- Visualization: **Table**

Panel 5: Dead Letter Queue — Rejected Records

- **Purpose:** Show invalid records caught by the DLQ before reaching Gold layer
- **Query Editor tab:**

SELECT ride_id, reason, rejected_at FROM dlq_rides LIMIT 20

- No date filter (shows all current rows regardless of date)
- No transform needed
- Visualization: **Table**



NOTE 1:
Panels 1, 2, 3, 4 filter by agg_date, which is the pipeline processing date (current_timestamp() at Spark write-time), not the taxi trip date. Every time we run a fresh pipeline session on a new day, manually update agg_date in each panel's query/ID Value to the current date, or the panels will show "No data" while querying a stale date.

NOTE 2: KNOWN QUIRK: Configurator/Query Editor tab reset

Occasionally, after editing a panel (especially Surge Events / Pickup Hotspots, which use raw CQL in Query Editor), the panel silently reverts to Configurator mode on its own — causing extra columns like surge_flag, pulocationid, hotspot_flag to disappear, since Configurator only supports one Value Column at a time.

To fix this, if this happens again:

1. Edit the panel → check whether it's showing **Configurator** or **Query Editor** as the active tab
2. If it's on Configurator, click back to **Query Editor**
3. Re-type (or confirm) the raw CQL query is still correct
4. Click **Apply**

This is a plugin behavior quirk, not a data or pipeline issue, the underlying Cassandra data is always fine, it's just the panel's query mode that occasionally resets.

BRIEF OF WHAT WE HAVE DONE TILL NOW

-> Implemented DLQ
-> Completed Creating Grafana Dashboard
Before we move to Airflow,

Grafana: A pre-built, saved dashboards for the specific graphs/tables you'll want to check repeatedly (ride count, fare trend, surge, hotspots, DLQ). and it **auto-refreshes**, so we can just leave it open and watch live updates as the pipeline runs, without re-running any query yourself.

cqlsh: the raw, direct interface to Cassandra. For **ad-hoc, one-off** questions we didn't pre-build a panel for. Like "let me check if ride_id X made it to gold_rides" or "how many total rows are in bronze right now". Quick manual digging, not something we'd want a permanent dashboard tile for.

STEP XXI: HOW TO RUN THE PROJECT TILL THIS STEP

-> First we go to our Code Base and delete the complete 'Checkpoints' folder.
-> Then we go to Task Manager, and end these tasks if they exist:

- Docker Desktop
- Docker Desktop Backend
- com.docker.backend

Then we start Docker Desktop again, just by clicking on the Docker Desktop icon.
(Using Anaconda Prompt Terminal)

Terminal 1: Docker Infra

```
-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> docker compose down
-> docker compose up -d
-> docker ps

-> docker exec -it kafka kafka-topics --create --topic taxi_rides --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
-> docker exec -it kafka kafka-topics --create --topic taxi_rides_dlq --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
-> docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092
```

(We will run cqlsh after terminal 3: when all rides are sent)
Before running Terminal 2, if we use cqlsh, we'll find that it shows old rows. To remove them, we can run these in cqlsh:

```
-> TRUNCATE dlq_rides;
-> TRUNCATE gold_rides;
-> TRUNCATE hotspot_rides;
-> TRUNCATE bronze_rides;
```

We can run these in cqlsh before starting a fresh pipeline run, so our query results only reflect that specific session

Terminal 2: Spark

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> set PYSPARK_PYTHON=C:\Users\rajve\anaconda3\envs\taxic\python.exe
-> spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.8,com.datastax.spark:spark-cassandra-connector_2.12:3.5.0 spark/stream_processor.py

Terminal 3: Producer

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> python producer/producer.py

Terminal 4: DLQ Kafka Verification (optional, on-demand)

-> activate taxic
-> cd "C:\Users\rajve\Desktop\SkillIssues\Personal Projects\taxistream"
-> docker exec -it kafka kafka-console-consumer --topic taxi_rides_dlq --from-beginning --bootstrap-server localhost:9092

Back To Terminal 1:

-> docker exec -it cassandra cqlsh (enter 'exit' to exit)
-> DESCRIBE KEYSPACES;
-> USE taxi_streaming;
-> DESCRIBE TABLES;

Now we can run queries such as:

-> SELECT * FROM dlq_rides LIMIT 5;
-> SELECT * FROM gold_rides LIMIT 5;
-> SELECT * FROM hotspot_rides LIMIT 5;
-> SELECT * FROM taxi_streaming.gold_rides LIMIT 5;
-> SELECT * FROM taxi_streaming.dlq_rides LIMIT 5;

Then, to check Spark Jobs: http://localhost:4040/jobs/
Then, to check Grafana DashBoards: http://localhost:3000 (We change the Dates in Query Editor and ID Value)

STEP XXII: AIRFLOW