Week 3 case study Report

Name: Balu Lakshmi sudha

Introduction:

This report explains how I designed and built a complete data pipeline to process, validate, transform, mask, and store customer transactions data using Spark Structured Streaming, Delta Lake, Great Expectations, and Apache NiFi.

1. Project Overview

Goal:

Build an end-to-end data pipeline that:

Streams synthetic transaction data from a Kafka/Event Hub producer.

Stores raw data in Bronze Delta tables.

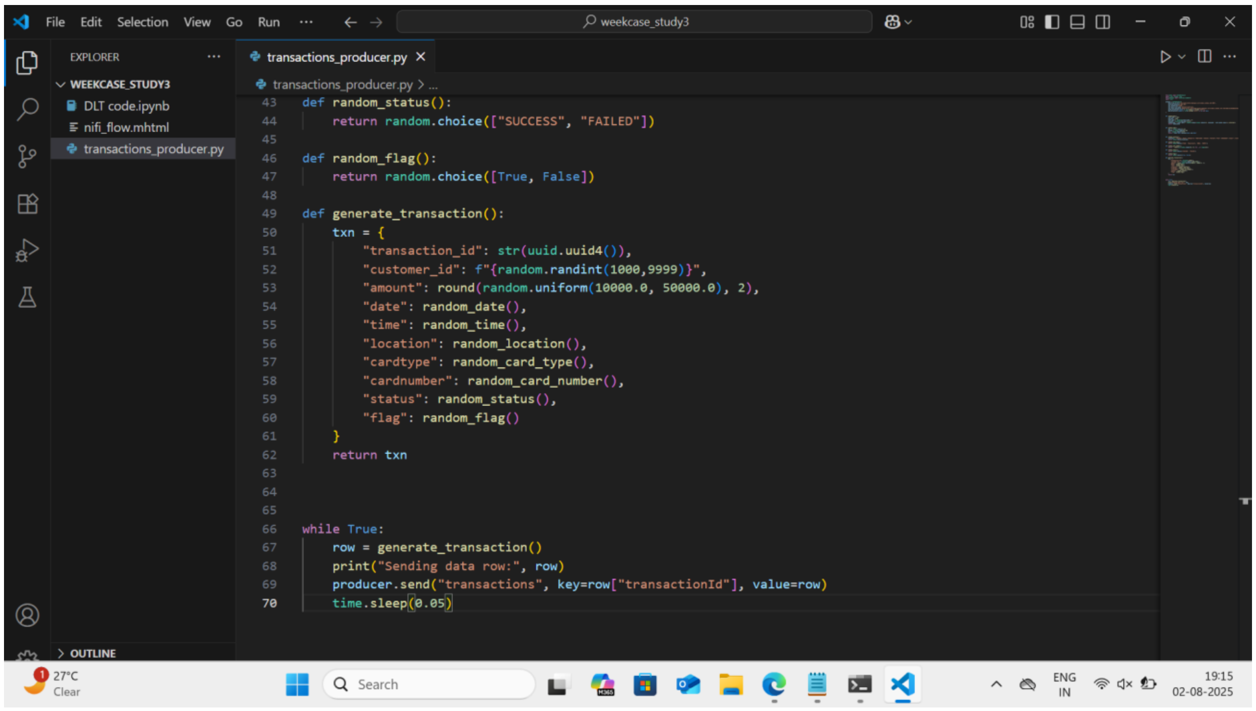
Validates and cleans the data using Great Expectations → Silver tables.

Applies masking, enrichment, business rules → Gold tables.

Uses Apache NiFi for additional transformation and masking.

Data: Simulated transactions with transaction_id, customer_id, amount, date, time, location, cardtype, cardnumber, status, flag

2. continuous streaming data generation



```python
def random_status():
    return random.choice(["SUCCESS", "FAILED"])

def random_flag():
    return random.choice([True, False])

def generate_transaction():
    txn = {
        "transaction_id": str(uuid.uuid4()),
        "customer_id": f"{random.randint(1000,9999)}",
        "amount": round(random.uniform(10000.0, 50000.0), 2),
        "date": random_date(),
        "time": random_time(),
        "location": random_location(),
        "cardtype": random_card_type(),
        "cardnumber": random_card_number(),
        "status": random_status(),
        "flag": random_flag()
    }
    return txn



while True:
    row = generate_transaction()
    print("Sending data row:", row)
    producer.send("transactions", key=row["transactionId"], value=row)
    time.sleep(0.05)
```

I Wrote a transactions_producer.py:

Uses uuid for transaction_id

random.randint for customer_id, amount

random_date and random_time to create date between (2015–2024) and time in (24-hour format)

random.choice for location (10 major Indian cities)

random.choice for cardtype (VISA, MasterCard, AMEX, RUPAY)

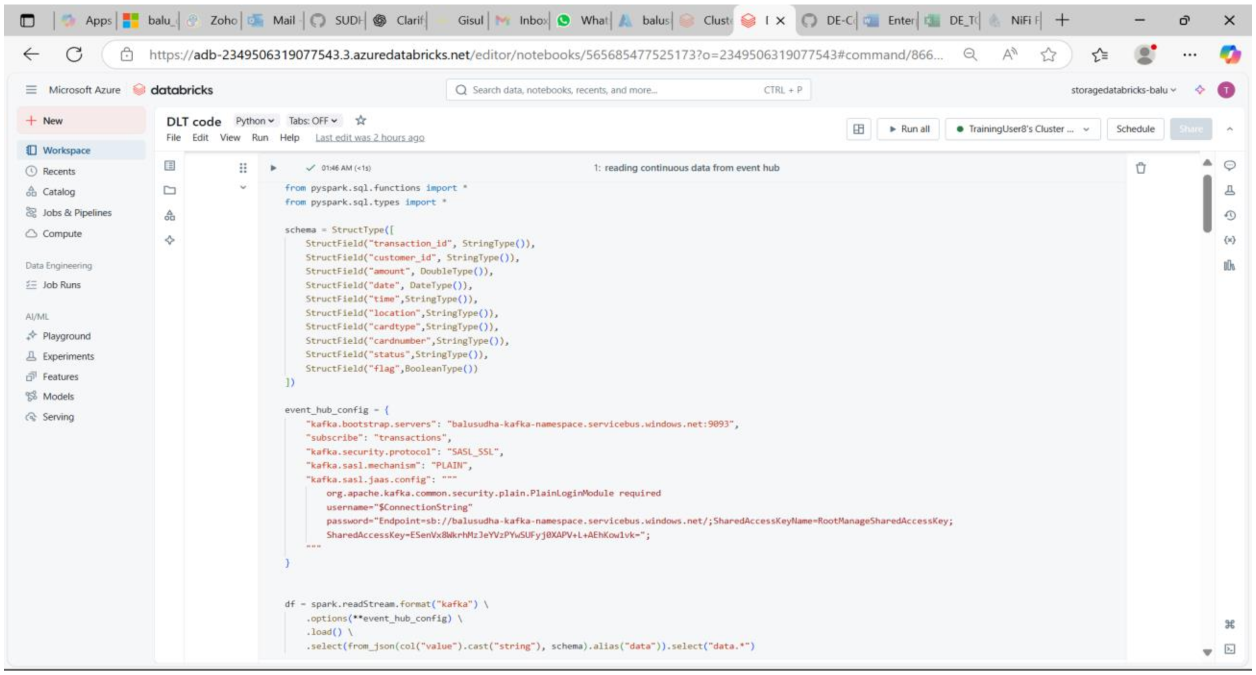
cardnumber generated as random 16-digit string

status and flag set randomly.

Sent data to transactions Event Hub.

3.consumer

Consumes the streaming transaction data coming from the Event Hub (configured as a Kafka-compatible source) and parses it into structured format for further processing.



```python
from pyspark.sql.functions import *
from pyspark.sql.types import *

schema = StructType([
    StructField("transaction_id", StringType()),
    StructField("customer_id", StringType()),
    StructField("amount", DoubleType()),
    StructField("date", DateType()),
    StructField("time",StringType()),
    StructField("location",StringType()),
    StructField("cardtype",StringType()),
    StructField("cardnumber",StringType()),
    StructField("status",StringType()),
    StructField("flag",BooleanType())
])

event_hub_config = {
    "kafka.bootstrap.servers": "balusudha-kafka-namespace.servicebus.windows.net:9093",
    "subscribe": "transactions",
    "kafka.security.protocol": "SASL_SSL",
    "kafka.sasl.mechanism": "PLAIN",
    "kafka.sasl.jaas.config": """
        org.apache.kafka.common.security.plain.PlainLoginModule required
        username="$ConnectionString"
        password="Endpoint=sb://balusudha-kafka-namespace.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;
        SharedAccessKey=ESenVx8WkrhMzJeYVzPYwSUFyj0XAPV+L+AEhKow1vk=";
    """
}


df = spark.readStream.format("kafka") \
    .options(**event_hub_config) \
    .load() \
    .select(from_json(col("value").cast("string"), schema).alias("data")).select("data.*")
```

4. Data Ingestion (Bronze)

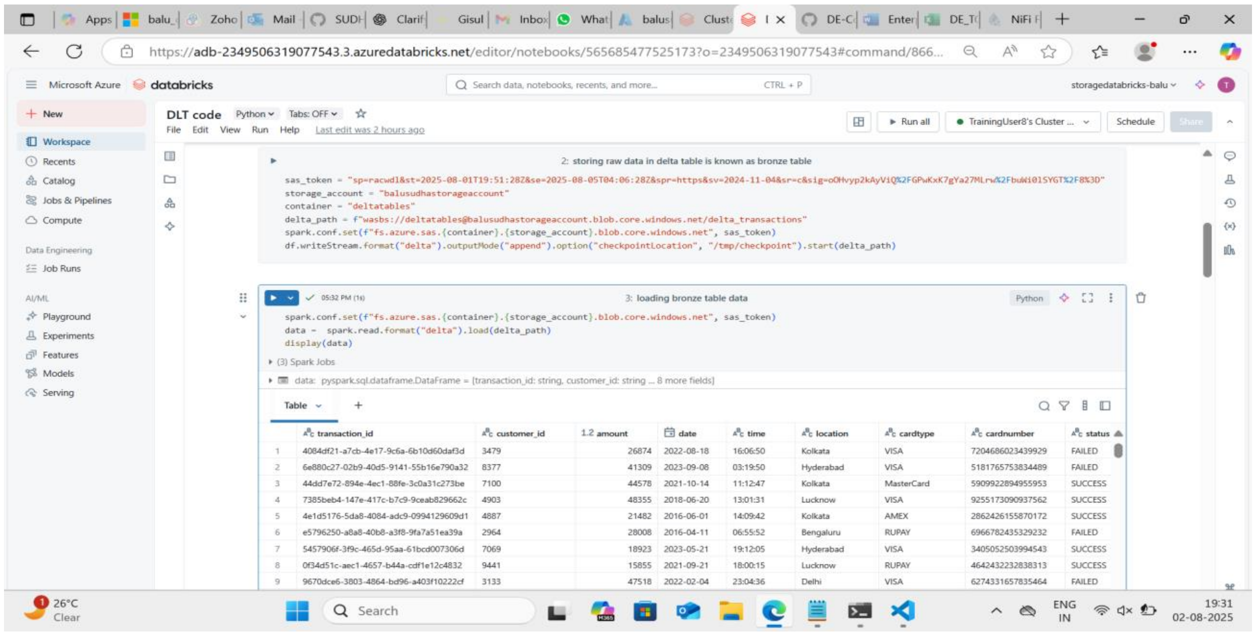
Used Spark Structured Streaming to connect to the Event Hub (Kafka-compatible).

Configured .readStream.format("kafka") with Event Hub connection settings.

Defined a schema and used from_json to parse the incoming JSON messages.

Extracted required fields from the JSON value column.

Stored the raw streaming data into a Delta Lake table: /delta_transactions.



2: storing raw data in delta table is known as bronze table

```
sas_token = "sp=racwdl&st=2025-08-01T19:51:28Z&se=2025-08-05T04:06:28Z&spr=https&sv=2024-11-04&sr=c&sig=oOHvyp2kAyViQ%2FGPwKxK7gYa27MLrw%2FbuWi0lSYGT%2F8%3D"
storage_account = "balusudhastorageaccount"
container = "deltatables"
delta_path = f"wasbs://deltatables@balusudhastorageaccount.blob.core.windows.net/delta_transactions"
spark.conf.set(f"fs.azure.sas.{container}.{storage_account}.blob.core.windows.net", sas_token)
df.writeStream.format("delta").outputMode("append").option("checkpointLocation", "/tmp/checkpoint").start(delta_path)
```

3: loading bronze table data

```
spark.conf.set(f"fs.azure.sas.{container}.{storage_account}.blob.core.windows.net", sas_token)
data =  spark.read.format("delta").load(delta_path)
display(data)
```

```
(3) Spark Jobs
data: pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
```

	transaction_id	customer_id	amount	date	time	location	cardtype	cardnumber	status
1	4084df21-a7cb-4e17-9c6a-6b10d60daf3d	3479	26874	2022-08-18	16:06:50	Kolkata	VISA	7204686023439929	FAILED
2	6e880c27-02b9-40d5-9141-55b16e790a32	8377	41309	2023-09-08	03:19:50	Hyderabad	VISA	5181765753834489	FAILED
3	44dd7e72-894e-4ec1-88fe-3c0a31c273be	7100	44578	2021-10-14	11:12:47	Kolkata	MasterCard	5909922894955953	SUCCESS
4	7385beb4-147e-417c-b7c9-9ceab829662c	4903	48355	2018-06-20	13:01:31	Lucknow	VISA	9255173090937562	SUCCESS
5	4e1d5176-5da8-4084-adc9-0994129609d1	4887	21482	2016-06-01	14:09:42	Kolkata	AMEX	2862426155870172	SUCCESS
6	e5796250-a8a8-40b8-a3f8-9fa7a51ea39a	2964	28008	2016-04-11	06:55:52	Bengaluru	RUPAY	6966782435329232	FAILED
7	5457906f-3f9c-465d-95aa-61bcd007306d	7069	18923	2023-05-21	19:12:05	Hyderabad	VISA	3405052503994543	SUCCESS
8	0f34d51c-aec1-4657-b44a-cdf1e12c4832	9441	15855	2021-09-21	18:00:15	Lucknow	RUPAY	4642432232838313	SUCCESS
9	9670dce6-3803-4864-bd96-a403f10222cf	3133	47518	2022-02-04	23:04:36	Delhi	VISA	6274331657835464	FAILED

5. Data Validation on Bronze Layer (Great Expectations)

Connected Bronze Delta table using Spark to load the raw streaming data.

Applied Great Expectations (GE) to check data quality on the ingested raw data.

Defined rules:

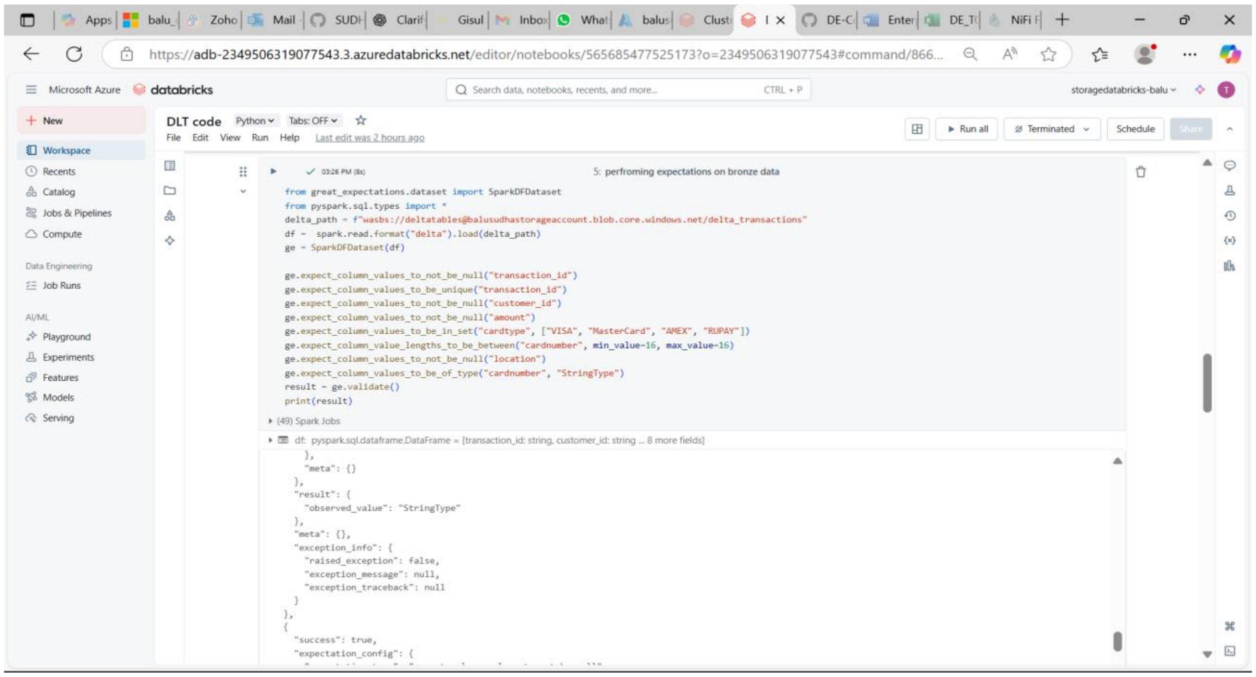
Checked that transaction_id is not null and unique.

Verified customer_id, amount, and location are not null.

Ensured cardtype values are within an allowed set (Visa, MasterCard, Amex, RuPay).

Verified cardnumber length is exactly 16 digits.

Ran validation using SparkDFDataset and validate() method.



5: perfroming expectations on bronze data

```python
from great_expectations.dataset import SparkDFDataset
from pyspark.sql.types import *
delta_path = f"wasbs://deltatables@balusudhastorageaccount.blob.core.windows.net/delta_transactions"
df =  spark.read.format("delta").load(delta_path)
ge = SparkDFDataset(df)

ge.expect_column_values_to_not_be_null("transaction_id")
ge.expect_column_values_to_be_unique("transaction_id")
ge.expect_column_values_to_not_be_null("customer_id")
ge.expect_column_values_to_not_be_null("amount")
ge.expect_column_values_to_be_in_set("cardtype", ["VISA", "MasterCard", "AMEX", "RUPAY"])
ge.expect_column_value_lengths_to_be_between("cardnumber", min_value=16, max_value=16)
ge.expect_column_values_to_not_be_null("location")
ge.expect_column_values_to_be_of_type("cardnumber", "StringType")
result = ge.validate()
print(result)
```

```
(49) Spark Jobs
df: pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
      },
      "meta": {}
    },
    "result": {
      "observed_value": "StringType"
    },
    "meta": {},
    "exception_info": {
      "raised_exception": false,
      "exception_message": null,
      "exception_traceback": null
    }
  },
  {
    "success": true,
    "expectation_config": {
```

5. Data Transformation and Masking (Silver Layer)

Read Bronze data from Delta Lake.

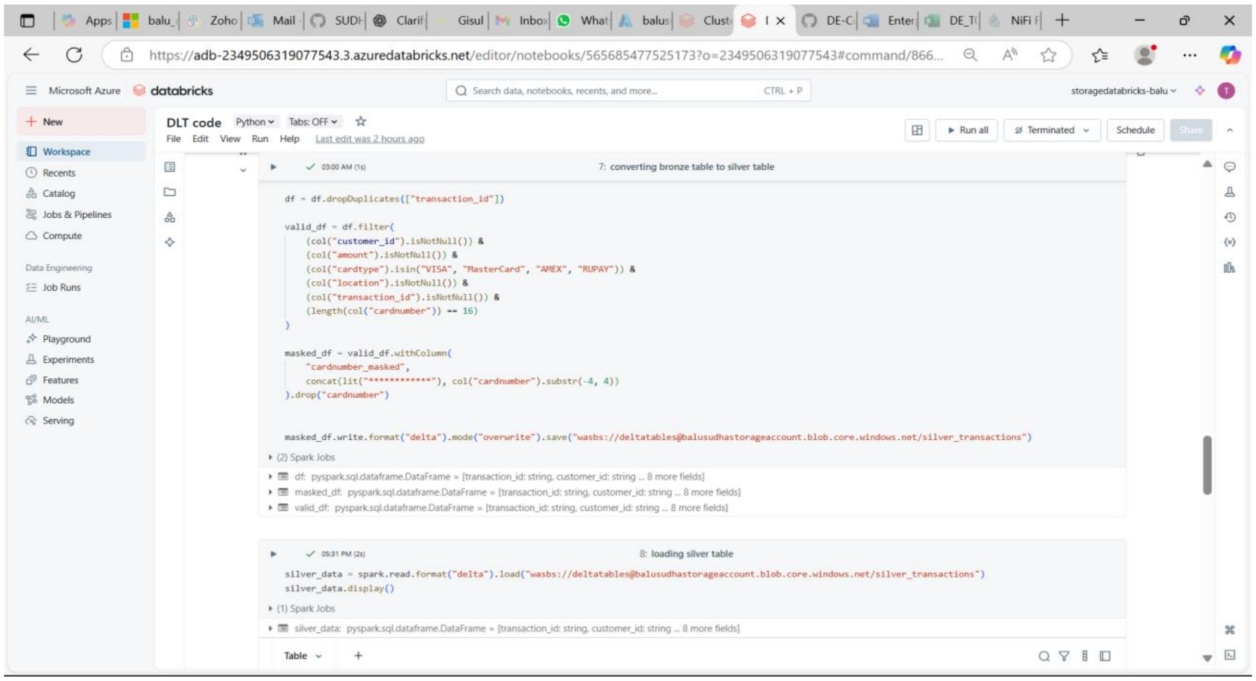
Filtered valid records based on Great Expectations results.

Performed PII masking on the cardnumber column:

Kept the last 4 digits visible.

Replaced first 12 digits with ************ using a concat and substring function.

Stored the cleaned, masked data in the Delta table: /silver_transactions



7: converting bronze table to silver table

03:00 AM (1s)

```
df = df.dropDuplicates(["transaction_id"])

valid_df = df.filter(
    (col("customer_id").isNotNull()) &
    (col("amount").isNotNull()) &
    (col("cardtype").isin("VISA", "MasterCard", "AMEX", "RUPAY")) &
    (col("location").isNotNull()) &
    (col("transaction_id").isNotNull()) &
    (length(col("cardnumber")) == 16)
)

masked_df = valid_df.withColumn(
    "cardnumber_masked",
    concat(lit("************"), col("cardnumber").substr(-4, 4))
).drop("cardnumber")


masked_df.write.format("delta").mode("overwrite").save("wasbs://deltatables@balusudhastorageaccount.blob.core.windows.net/silver_transactions")
```

```
(2) Spark Jobs
df:  pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
masked_df:  pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
valid_df:  pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
```

8: loading silver table

05:31 PM (2s)

```
silver_data = spark.read.format("delta").load("wasbs://deltatables@balusudhastorageaccount.blob.core.windows.net/silver_transactions")
silver_data.display()
```

```
(1) Spark Jobs
silver_data:  pyspark.sql.dataframe.DataFrame = [transaction_id: string, customer_id: string ... 8 more fields]
```

Table

6. Data Aggregation and Analytics (Gold Layer)

Read Silver table data for trusted, cleaned transactions.

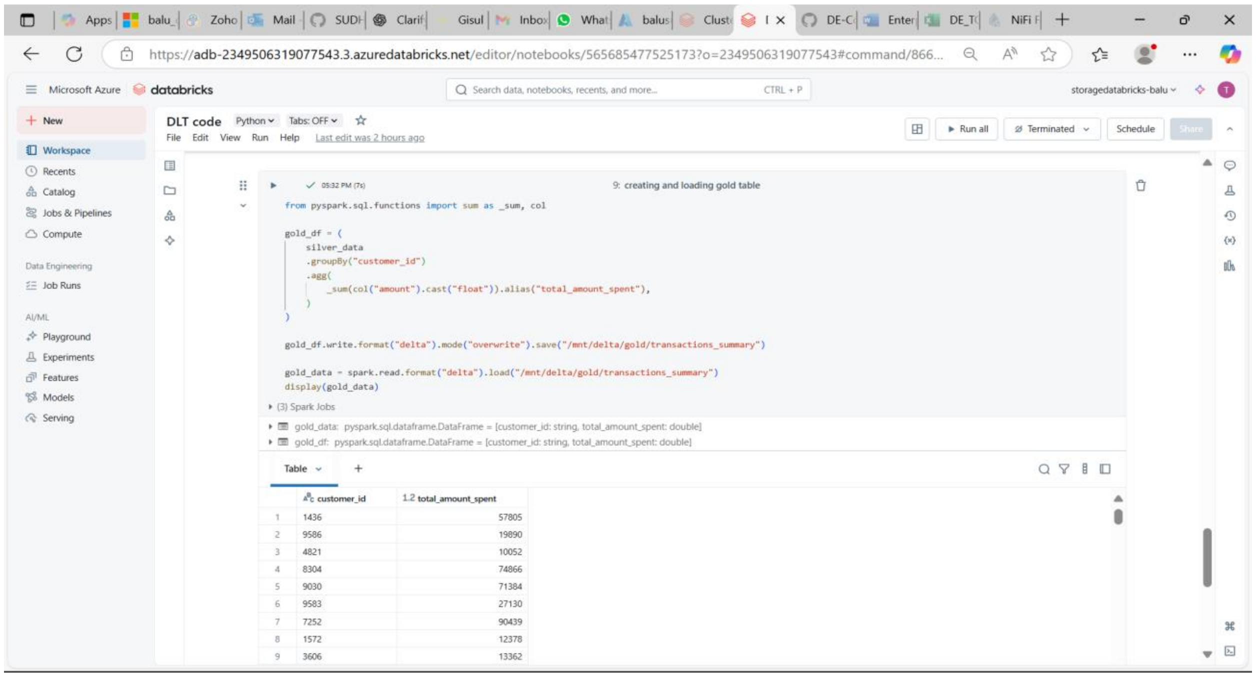
Performed aggregation:

Grouped data by customer_id.

Calculated the total amount spent by each customer.

Generated customer-level insights for reporting and downstream analytics.

Stored the aggregated data in the Delta table: /gold/transactions_summary.



```python
from pyspark.sql.functions import sum as _sum, col

gold_df = (
    silver_data
    .groupBy("customer_id")
    .agg(
        _sum(col("amount").cast("float")).alias("total_amount_spent"),
    )
)

gold_df.write.format("delta").mode("overwrite").save("/mnt/delta/gold/transactions_summary")

gold_data = spark.read.format("delta").load("/mnt/delta/gold/transactions_summary")
display(gold_data)
```

```
(3) Spark Jobs
gold_data: pyspark.sql.dataframe.DataFrame = [customer_id: string, total_amount_spent: double]
gold_df: pyspark.sql.dataframe.DataFrame = [customer_id: string, total_amount_spent: double]
```

	customer_id	total_amount_spent
1	1436	57805
2	9586	19890
3	4821	10052
4	8304	74866
5	9030	71384
6	9583	27130
7	7252	90439
8	1572	12378
9	3606	13362

7. Data Processing using Apache NiFi

Used Apache NiFi to handle additional filtering and record-level transformations.

GetFile Processor: Picked up the source CSV file (Silver data).

QueryRecord Processor:

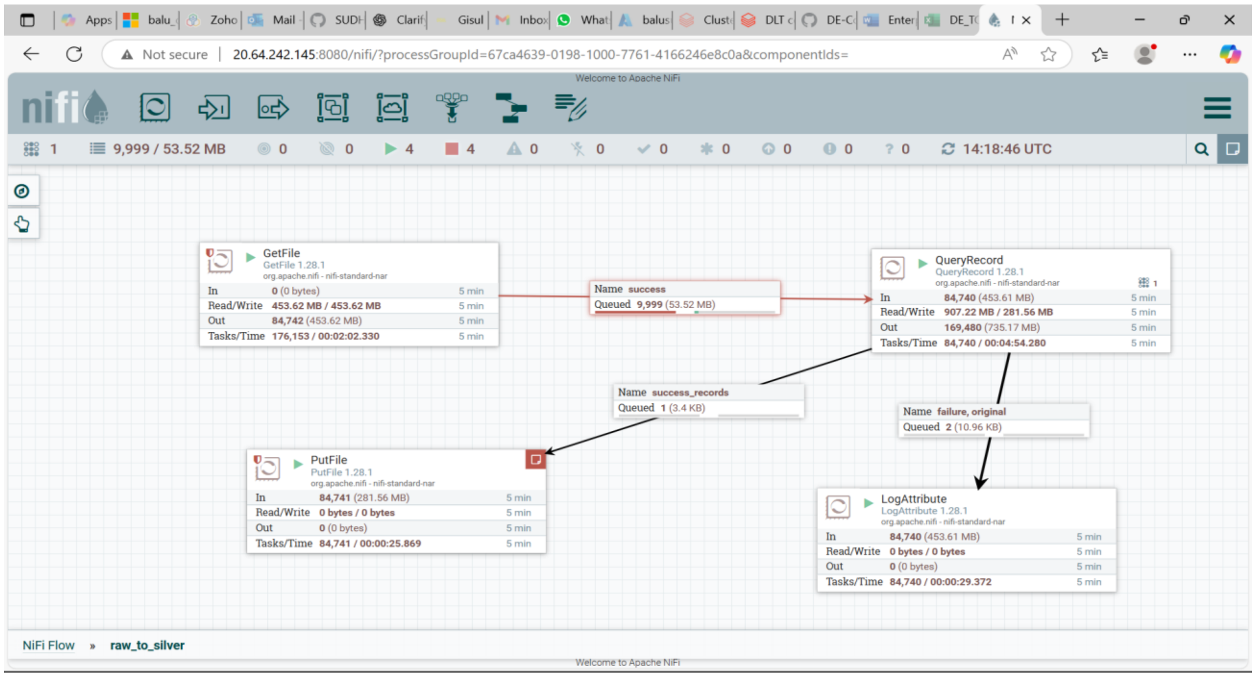
Applied custom SQL-like filters on the ingested records.

Example: Filtered transactions based on conditions such as status = 'SUCCESS' or specific cardtype.

Used Record Reader/Writer with schema for structured processing.

PutFile Processor: Stored the processed (filtered) records into a target directory.

LogAttribute Processor: Captured logs for flow debug and verification — used to inspect records that failed processing.


Not secure | 20.64.242.145:8080/nifi/?processGroupId=67ca4639-0198-1000-7761-4166246e8c0a&componentIds=
Welcome to Apache NiFi
1
9,999 / 53.52 MB
14:18:46 UTC
NiFi Flow » raw_to_silver

GetFile
GetFile 1.28.1
org.apache.nifi - nifi-standard-nar

In	0 (0 bytes)	5 min
Read/Write	453.62 MB / 453.62 MB	5 min
Out	84,742 (453.62 MB)	5 min
Tasks/Time	176,153 / 00:02:02.330	5 min

Name success
Queued 9,999 (53.52 MB)

QueryRecord
QueryRecord 1.28.1
org.apache.nifi - nifi-standard-nar

In	84,740 (453.61 MB)	5 min
Read/Write	907.22 MB / 281.56 MB	5 min
Out	169,480 (735.17 MB)	5 min
Tasks/Time	84,740 / 00:04:54.280	5 min

Name success_records
Queued 1 (3.4 KB)

Name failure, original
Queued 2 (10.96 KB)

PutFile
PutFile 1.28.1
org.apache.nifi - nifi-standard-nar

In	84,741 (281.56 MB)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	84,741 / 00:00:25.869	5 min

LogAttribute
LogAttribute 1.28.1
org.apache.nifi - nifi-standard-nar

In	84,740 (453.61 MB)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	84,740 / 00:00:29.372	5 min

Conclusion:

In this project, I built a complete data pipeline from raw data to final insights.

I ingested streaming data from Event Hub using Spark Structured Streaming.

I validated the raw data with Great Expectations to ensure data quality.

I filtered and masked sensitive information and stored the clean data in Silver tables.

I aggregated the Silver data to create Gold tables for business analysis.

Finally, I used Apache NiFi to filter the processed data further using the QueryRecord processor and saved the filtered results to files for additional downstream use.

This end-to-end flow helps ensure the data is clean, secure, and ready for reporting and analytics.