Real-time Data Processing using Azure Event Hub and Azure Databricks

(Traffic Monitoring)

Table of Contents

Project Statement

The aim of this project is to design and implement a real-time data processing pipeline that ingests live traffic IoT sensor data through Azure Event Hub, processes it in near real-time using Azure Databricks (Apache Spark Structured Streaming), and stores the results in Delta Lake for analytics. The solution demonstrates how streaming data can be transformed from raw ingestion (Bronze) to refined analytics (Gold).

Project Overview

Traffic congestion is a critical urban problem. IoT devices and sensors mounted on vehicles generate continuous telemetry such as speed, timestamp, and vehicle ID. The project showcases:

- Ingestion Layer (Event Hub): Capturing raw traffic events from multiple vehicles in real time.
- Processing Layer (Databricks): Using Spark Structured Streaming to process, cleanse, and aggregate events.
- Storage Layer (Delta Lake): Persisting data into Bronze (raw), Silver (cleaned), and Gold (aggregated) tables for reporting.
- Analytics: Average vehicle speed and traffic density computed over sliding time windows.

Prerequisites

1. Python knowledge (PySpark for Structured Streaming).
2. Databricks cluster with Delta Lake enabled.
3. Azure Subscription: Have an active Azure subscription for resource management.
4. Azure subscription with Event Hub
5. Azure Databricks :Set up an Azure Databricks workspace for Spark processing.
6. Databricks Cluster: Set up a Databricks cluster for Spark jobs.

7. Libraries and Dependencies: Install required libraries in Databricks.
8. Monitoring and Logging: Set up monitoring in Databricks.
9. Azure Data Lake Storage (ADLS Gen2) mounted in Databricks.

Project Requirements

1. **Technical Infrastructure**

- **Azure Event Hub**: Provision and configure Event Hub namespace and instance for ingesting high-throughput IoT events.
- **Azure Databricks**: Workspace and clusters for running Spark Structured Streaming workloads.
- **Delta Lake**: Implement Delta Lake on top of Azure Data Lake Storage (ADLS) for maintaining Bronze, Silver, and Gold layers with ACID compliance.
- **Azure Storage Accounts**: Blob or Data Lake Gen2 accounts for storing raw, cleaned, and aggregated data.

2. **Data Sources (Streaming IoT Events)**:

    - Schema:
    - vehicle_id (INT)
    - speed (INT)
    - timestamp (STRING → converted to TIMESTAMP)

Destination (Delta Lake Layers):

- **Bronze**: Raw ingested events from Event Hub (append-only, minimal transformation).
- **Silver**: Cleaned and structured data with proper schema and timestamp conversions.
- **Gold**: Aggregated/curated tables (e.g., average speed per vehicle, congestion detection).

3. Development Tools

- **Databricks Notebooks**: PySpark code for ingestion, transformation, and aggregation.
- **Spark Structured Streaming**: For real-time data processing from Event Hub to Delta tables.
- **Git Repository**: Version control for notebooks, configuration files, and deployment scripts.

4. Security and Compliance

- **Access Controls**: Role-based access for Event Hub, ADLS, and Databricks using Azure RBAC.
- **Encryption**: Encrypt data in transit (TLS) and at rest (Storage encryption + Delta Lake).

5. Performance and Scalability

- **Databricks Cluster Sizing**: Provision auto-scaling clusters based on event throughput.
- **Event Hub Throughput Units**: Adjust partitions and throughput units to handle high-velocity IoT streams.
- **Scalability**: Pipeline should seamlessly scale as more vehicles or higher event rates are ingested.

6. Monitoring and Logging

- **Streaming Pipeline Monitoring**: Use Azure Monitor, Databricks Jobs UI, and Event Hub metrics.
- **Delta Lake Transaction Logs**: Track schema evolution, updates, and failures.
- **Error Handling**: Dead-letter queues or error storage for malformed records.

7. Documentation and Training

- **Technical Documentation**: Detailed guide covering:
    - Event Hub setup and integration with Databricks
    - Structured Streaming configuration (checkpointing, watermarking)

- Delta Lake Bronze/Silver/Gold design patterns

- **Training Sessions**: For data engineers to operate and maintain the streaming pipeline.

8. Project Management

- **Timeline**: Define milestones for environment setup, ingestion pipeline, Silver transformations, Gold aggregations, and final validations.
- **Budget**: Costs for Event Hub throughput, Databricks compute, and storage in ADLS.
- **Risk Management**:
  - Event Hub throttling (mitigation: increase throughput units)
  - Databricks cluster overuse (mitigation: auto-scaling + monitoring)
  - Schema drift in IoT data (mitigation: Delta Lake schema evolution + error handling)

Execution Overview

1. Simulate IoT data: Python producer sends random traffic events into Event Hub.
2. Bronze Layer: Raw ingestion of events from Event Hub into Delta table.
3. Silver Layer: Data parsing, schema enforcement, timestamp conversion.
4. Gold Layer: Aggregations using Spark Structured Streaming (e.g., average speed per 5-minute window per vehicle).
5. Analytics: Query Gold Delta tables for dashboards/Power BI.

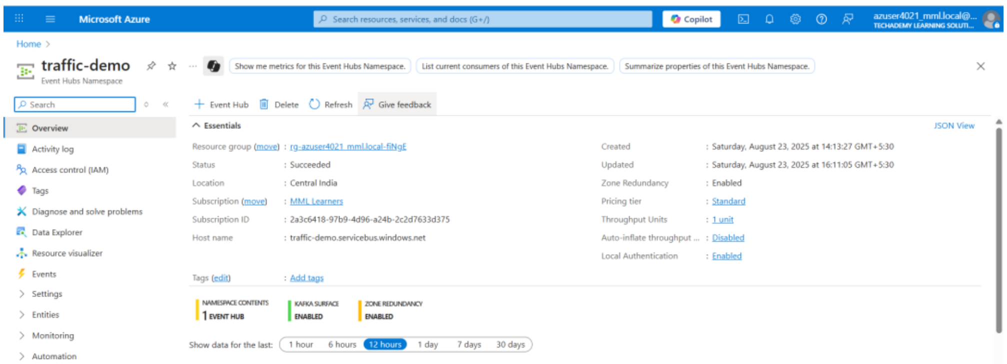
Source Data Files (Simulated streaming data from IoT traffic sensors in JSON format)

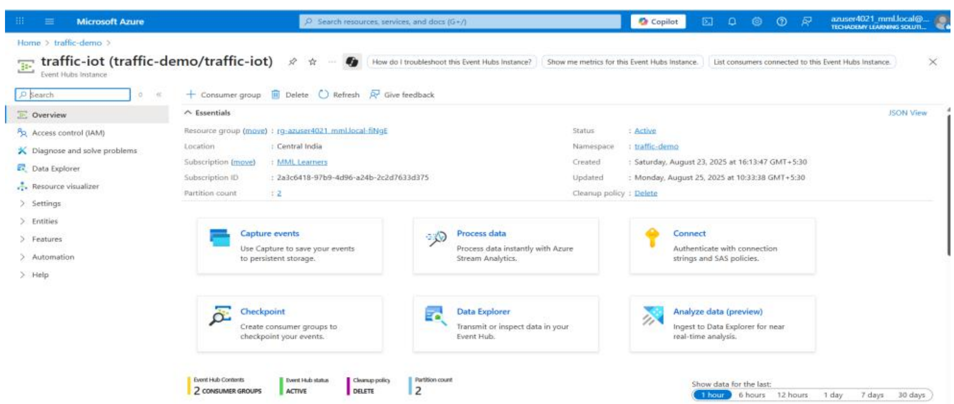

```
{
 "vehicle_id": 1234,
 "speed": 85,
 "timestamp": "2025-08-26T10:05:12.101Z"
}
```

Implementation – Tasks Performed

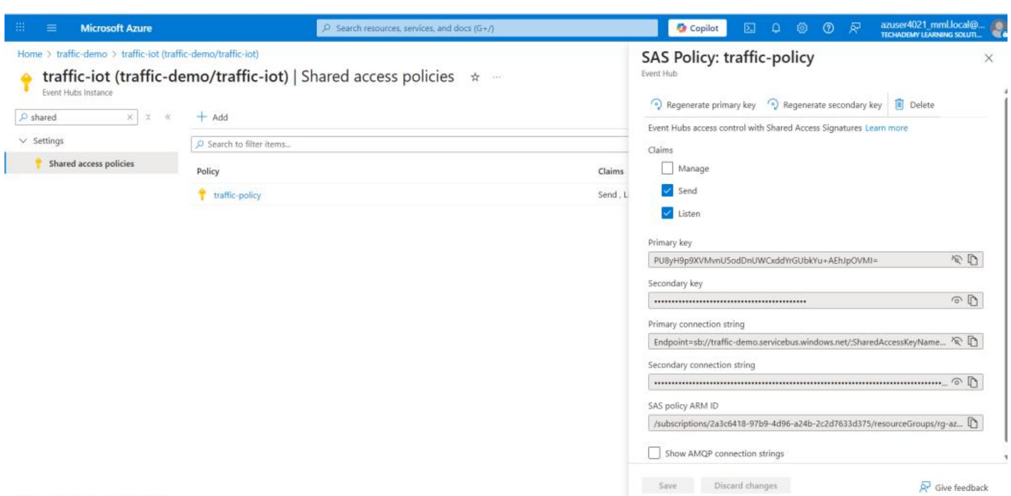
1. Create Event Hub

- Provisioned Event Hub namespace and event hub traffic-iot.





- Created Shared Access Policy for connection string.



2. Producer – Send IoT Data

Python script to simulate streaming:

```
from azure.eventhub import EventHubConsumerClient
CONNECTION_STR = "Endpoint=sb://traffic-
demo.servicebus.windows.net/;SharedAccessKeyName=traffic-
policy;SharedAccessKey=PU8yH9p9XVMvnU5odDnUWCxddYrGUbkYu+AEh
JpOVMI=;EntityPath=traffic-iot"
CONSUMER_GROUP = "traffic-app"  # Default consumer group
EVENTHUB_NAME = "traffic-iot"

def on_event(partition_context, event):
   print(f"Received event from partition: {partition_context.partition_id}")
   print(event.body_as_str())
   partition_context.update_checkpoint(event)

client = EventHubConsumerClient.from_connection_string(
   conn_str=CONNECTION_STR,
   consumer_group=CONSUMER_GROUP,
   eventhub_name=EVENTHUB_NAME
)

with client:
   client.receive(
      on_event=on_event,
      starting_position="-1",  # "-1" = read from beginning of stream
   )
```

```
{"vehicle_id": 5048, "speed": 91, "timestamp": "2025-08-26 11:24:33.878532"}
Received event from partition: 0
{"vehicle_id": 3320, "speed": 23, "timestamp": "2025-08-26 11:24:03.196363"}
Received event from partition: 1
{"vehicle_id": 2339, "speed": 94, "timestamp": "2025-08-26 11:24:48.890203"}
Received event from partition: 0
{"vehicle_id": 1717, "speed": 109, "timestamp": "2025-08-26 11:24:14.996991"}
Received event from partition: 0
{"vehicle_id": 4382, "speed": 26, "timestamp": "2025-08-26 11:24:26.283237"}
Received event from partition: 0
{"vehicle_id": 9789, "speed": 109, "timestamp": "2025-08-26 11:24:29.981159"}
Received event from partition: 0
{"vehicle_id": 7277, "speed": 94, "timestamp": "2025-08-26 11:24:37.613561"}
Received event from partition: 0
{"vehicle_id": 5131, "speed": 74, "timestamp": "2025-08-26 11:24:41.442749"}
Received event from partition: 0
{"vehicle_id": 5495, "speed": 85, "timestamp": "2025-08-26 11:24:45.183425"}
Received event from partition: 0
{"vehicle_id": 9106, "speed": 39, "timestamp": "2025-08-26 11:24:52.573080"}
Received event from partition: 1
{"vehicle_id": 7002, "speed": 105, "timestamp": "2025-08-26 11:24:56.290816"}
```


3. Bronze Layer – Raw Ingestion

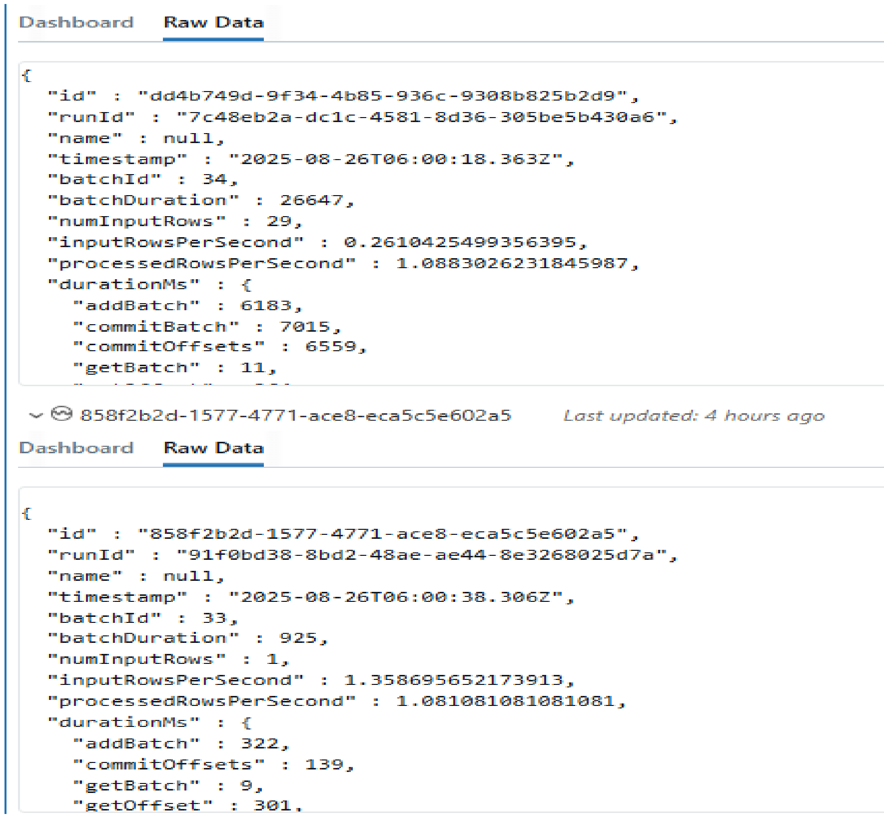
Code:

```
from pyspark.sql.types import StructType, StringType
from pyspark.sql.functions import col, from_json, coalesce, to_timestamp

# ---- 1) Schema: keep everything as STRING, cast later (avoids nulls on type mismatch)
schema = (StructType()
    .add("vehicle_id", StringType())
    .add("speed", StringType())
    .add("timestamp", StringType())    # some producers use "timestamp"
    .add("event_time", StringType()))  # others use "event_time"

# ---- 2) Secure your connection string (example uses a placeholder)
# connectionString = dbutils.secrets.get("scope", "eh-conn")   # recommended
```

```
connectionString ="Endpoint=sb://traffic-
demo.servicebus.windows.net/;SharedAccessKeyName=traffic-
policy;SharedAccessKey=PU8yH9p9XVMvnU5odDnUWCxddYrGUbkYu+AEh
JpOVMI=;EntityPath=traffic-iot"
eh_conf = {
  "eventhubs.connectionString":
sc._jvm.org.apache.spark.eventhubs.EventHubsUtils.encrypt(connectionString),
  "eventhubs.consumerGroup": "$Default",
  # Start from the earliest retained events
  "eventhubs.startingPosition": """{
   "offset": "-1",
   "seqNo": -1,
   "enqueuedTime": null,
   "isInclusive": true
  }"""
}

# ---- 3) Read Event Hubs stream
raw_stream = (spark.readStream
       .format("eventhubs")
       .options(**eh_conf)
       .load())

# ---- 4) Parse JSON safely
parsed = raw_stream.select(
  col("enqueuedTime").alias("ingest_time"),
  from_json(col("body").cast("string"), schema).alias("data")
)
```

```
iot_df = (parsed
    .select(
        col("data.vehicle_id").cast("int").alias("vehicle_id"),
        col("data.speed").cast("int").alias("speed"),
        # prefer payload time; fall back to Event Hubs enqueued time
        coalesce(col("data.timestamp"), col("data.event_time")).alias("ts_raw"),
        col("ingest_time")
    )
    .withColumn(
        "event_time",
        coalesce(
            to_timestamp(col("ts_raw")),                    # try default parsing
            col("ingest_time").cast("timestamp")            # fallback
        )
    )
    .drop("ts_raw")
)
# ---- 5) Write to Delta (BRONZE) with a dedicated checkpoint
delta_query = (iot_df.writeStream
    .format("delta")
    .outputMode("append")
    .option("checkpointLocation", "/mnt/bronze/_checkpoint_iot")
    .option("mergeSchema", "true")    # allow new/changed columns
    .start("/mnt/bronze/iotdata"))

# ---- 6) OPTIONAL: Also stream to console so you can verify rows immediately
debug_query = (iot_df.writeStream
```

```
.format("console")
.option("truncate", False)
.option("numRows", 20)
.start())

bronze_df = (spark.readStream
 .format("eventhubs")
 .option("eventhubs.connectionString", EVENTHUB_CONNECTION_STRING)
 .load())

bronze_df.writeStream.format("delta") \
.outputMode("append") \
.option("checkpointLocation", "/mnt/bronze/_checkpoint") \
.start("/mnt/bronze/iotdata")
```


Dashboard Raw Data


```
{
  "id" : "dd4b749d-9f34-4b85-936c-9308b825b2d9",
  "runId" : "7c48eb2a-dc1c-4581-8d36-305be5b430a6",
  "name" : null,
  "timestamp" : "2025-08-26T06:00:18.363Z",
  "batchId" : 34,
  "batchDuration" : 26647,
  "numInputRows" : 29,
  "inputRowsPerSecond" : 0.2610425499356395,
  "processedRowsPerSecond" : 1.0883026231845987,
  "durationMs" : {
    "addBatch" : 6183,
    "commitBatch" : 7015,
    "commitOffsets" : 6559,
    "getBatch" : 11,
```


858f2b2d-1577-4771-ace8-eca5c5e602a5 Last updated: 4 hours ago

Dashboard Raw Data


```
{
  "id" : "858f2b2d-1577-4771-ace8-eca5c5e602a5",
  "runId" : "91f0bd38-8bd2-48ae-ae44-8e3268025d7a",
  "name" : null,
  "timestamp" : "2025-08-26T06:00:38.306Z",
  "batchId" : 33,
  "batchDuration" : 925,
  "numInputRows" : 1,
  "inputRowsPerSecond" : 1.358695652173913,
  "processedRowsPerSecond" : 1.081081081081081,
  "durationMs" : {
    "addBatch" : 322,
    "commitOffsets" : 139,
    "getBatch" : 9,
    "getOffset" : 301,
```


\

4. Silver Layer – Cleaned Data

Code

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
from pyspark.sql.functions import from_json, col, to_timestamp,
current_timestamp
# 1) Schema
schema = StructType([
    StructField("vehicle_id", IntegerType(), True),
    StructField("speed", IntegerType(), True),
    StructField("timestamp", StringType(), True)  # raw string

])
```

```
# 2) Parse + Add event_time

parsed_df = (raw_stream

    .select(from_json(col("body").cast("string"), schema).alias("data"))

    .select("data.*")

        .withColumn("event_time", to_timestamp("timestamp"))  # convert to
timestamp

    .withColumn("ingest_time", current_timestamp())     # optional lineage

)

# 3) Paths

silver_path = "dbfs:/mnt/silver/iotdata"

checkpoint_path = "dbfs:/mnt/silver/_checkpoint_iot"

# 4) Write to Silver (Delta Lake)

delta_query = (parsed_df.writeStream

    .format("delta")

    .outputMode("append")

    .option("checkpointLocation", checkpoint_path)

    .option("mergeSchema", "true")   # allow schema evolution

    .start(silver_path)

)

# 5) Debug Console Sink (for monitoring)

console_query = (parsed_df.writeStream
```

```python
        .format("console")

        .outputMode("append")

        .option("truncate", False)

        .start()

    )

# 6) ---- In a separate cell, AFTER stream runs ----

# Read from Silver Delta table

silver_df = spark.read.format("delta").load(silver_path)

# Show only 10 rows

silver_df.show(10, truncate=False)
```

```
▸ (1) Spark Jobs
▾ silver_df: pyspark.sql.dataframe.DataFrame
Schema   Details   History

    vehicle_id: integer
    speed: integer
    timestamp: string
    event_time: timestamp
    ingest_time: timestamp

+----------+-----+--------------------------+--------------------------+-----------------------+
|vehicle_id|speed|timestamp                 |event_time                |ingest_time            |
+----------+-----+--------------------------+--------------------------+-----------------------+
|4302      |77   |2025-08-26 10:52:01.729790|2025-08-26 10:52:01.72979 |2025-08-26 05:22:59.146|
|6153      |54   |2025-08-26 10:52:05.353472|2025-08-26 10:52:05.353472|2025-08-26 05:22:59.146|
|4564      |101  |2025-08-26 10:52:08.999575|2025-08-26 10:52:08.999575|2025-08-26 05:22:59.146|
|5388      |45   |2025-08-26 10:52:19.743766|2025-08-26 10:52:19.743766|2025-08-26 05:22:59.146|
|1647      |119  |2025-08-26 10:52:23.442930|2025-08-26 10:52:23.44293 |2025-08-26 05:22:59.146|
|8571      |30   |2025-08-26 10:58:05.309953|2025-08-26 10:58:05.309953|2025-08-26 05:28:58.935|
|2746      |100  |2025-08-26 10:58:12.329023|2025-08-26 10:58:12.329023|2025-08-26 05:28:58.935|
|1577      |66   |2025-08-26 10:58:23.013157|2025-08-26 10:58:23.013157|2025-08-26 05:28:58.935|
|2226      |95   |2025-08-26 10:58:37.305759|2025-08-26 10:58:37.305759|2025-08-26 05:29:26.827|
|6887      |24   |2025-08-26 10:58:44.339358|2025-08-26 10:58:44.339358|2025-08-26 05:29:26.827|
+----------+-----+--------------------------+--------------------------+-----------------------+
only showing top 10 rows
```

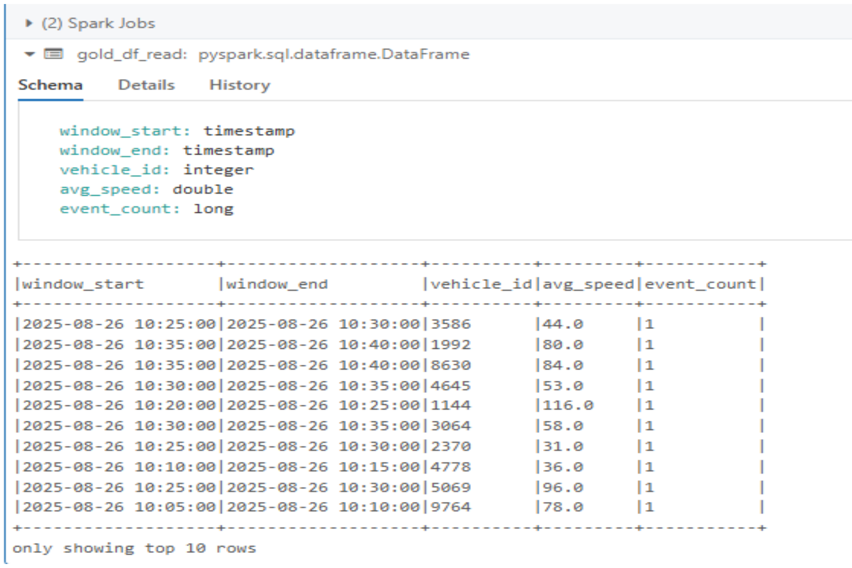
5. Gold Layer – Aggregated Analytics

Code:

```
from pyspark.sql.functions import col, avg, count, window

# 1. Read from silver (streaming)

silver_df = (

    spark.readStream

    .format("delta")

    .load("/mnt/silver/iotdata")

)
```

```
# 2. Aggregate for gold

gold_df = (

    silver_df

    .withWatermark("event_time", "10 minutes")   # use watermark on event_time

    .groupBy(

        window(col("event_time"), "5 minutes"),  # tumbling window of 5 mins

        col("vehicle_id")

    )

    .agg(

        avg("speed").alias("avg_speed"),

        count("*").alias("event_count")

    )
```

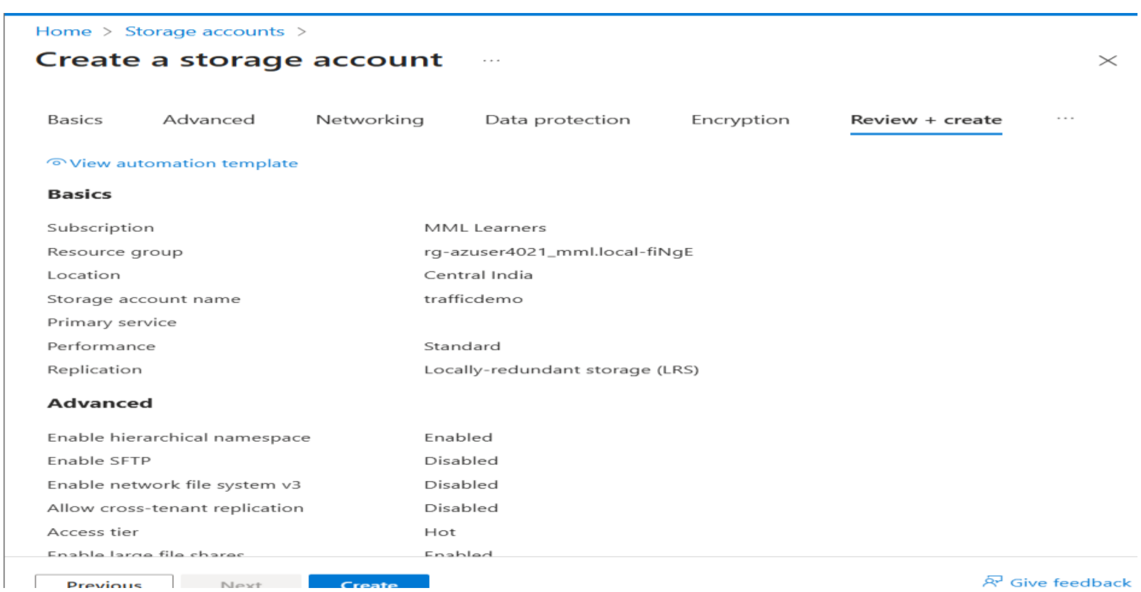
```
    .select(
        col("window.start").alias("window_start"),
        col("window.end").alias("window_end"),
        col("vehicle_id"),
        col("avg_speed"),
        col("event_count")
    )
)
# 3. Write to gold (Delta table)
query = (
    gold_df.writeStream
    .format("delta")
    .outputMode("append")     # required for aggregates with watermark
    .option("checkpointLocation", "/mnt/gold/_checkpoint_iot")
    .start("/mnt/gold/iotdata")
)
# Register the gold delta folder as a temporary view
gold_df_read = spark.read.format("delta").load("/mnt/gold/iotdata")
# Show 10 rows
gold_df_read.show(10, truncate=False)
```

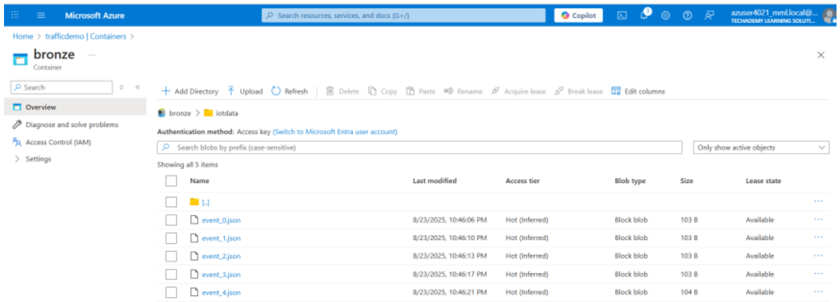


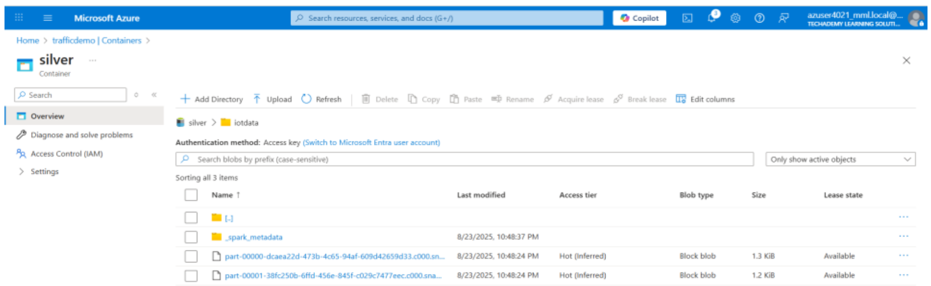

```
+-------------------+-------------------+----------+---------+-----------+
|window_start       |window_end         |vehicle_id|avg_speed|event_count|
+-------------------+-------------------+----------+---------+-----------+
|2025-08-26 10:25:00|2025-08-26 10:30:00|3586      |44.0     |1          |
|2025-08-26 10:35:00|2025-08-26 10:40:00|1992      |80.0     |1          |
|2025-08-26 10:35:00|2025-08-26 10:40:00|8630      |84.0     |1          |
|2025-08-26 10:30:00|2025-08-26 10:35:00|4645      |53.0     |1          |
|2025-08-26 10:20:00|2025-08-26 10:25:00|1144      |116.0    |1          |
|2025-08-26 10:30:00|2025-08-26 10:35:00|3064      |58.0     |1          |
|2025-08-26 10:25:00|2025-08-26 10:30:00|2370      |31.0     |1          |
|2025-08-26 10:10:00|2025-08-26 10:15:00|4778      |36.0     |1          |
|2025-08-26 10:25:00|2025-08-26 10:30:00|5069      |96.0     |1          |
|2025-08-26 10:05:00|2025-08-26 10:10:00|9764      |78.0     |1          |
+-------------------+-------------------+----------+---------+-----------+
only showing top 10 rows
```

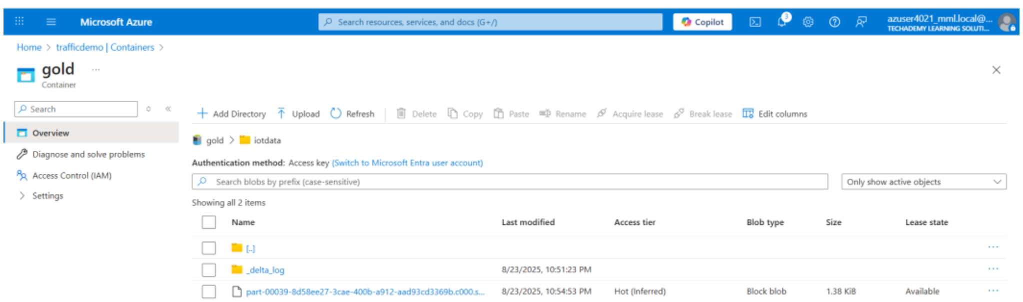
Steps on Practical Implementation on Azure Portal

1. Created Event Hub namespace and traffic event hub.
2. Generated connection string from Shared Access Policy.
3. Built producer script and sent events.
4. Mounted ADLS to Databricks for Delta Lake storage.
5. Implemented Bronze → Silver → Gold streaming layers in Databricks.

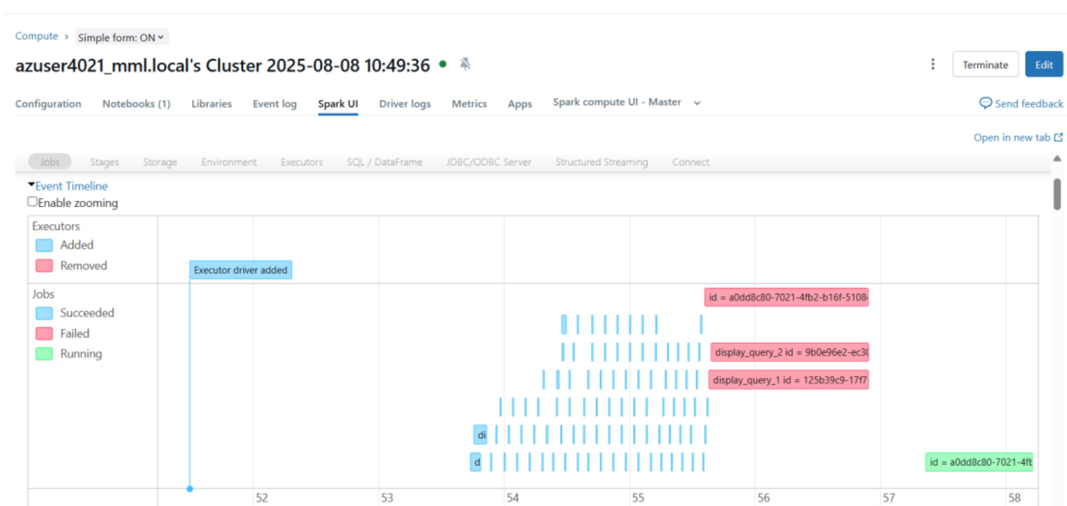


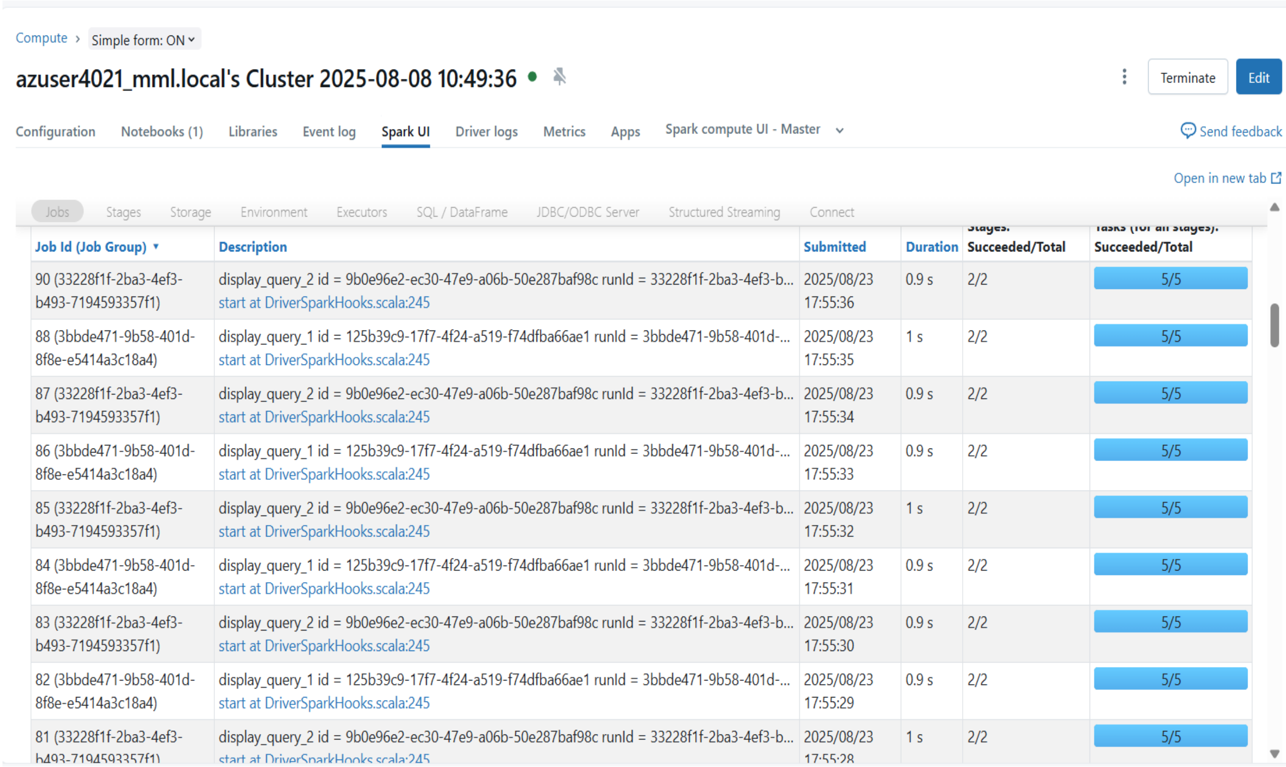




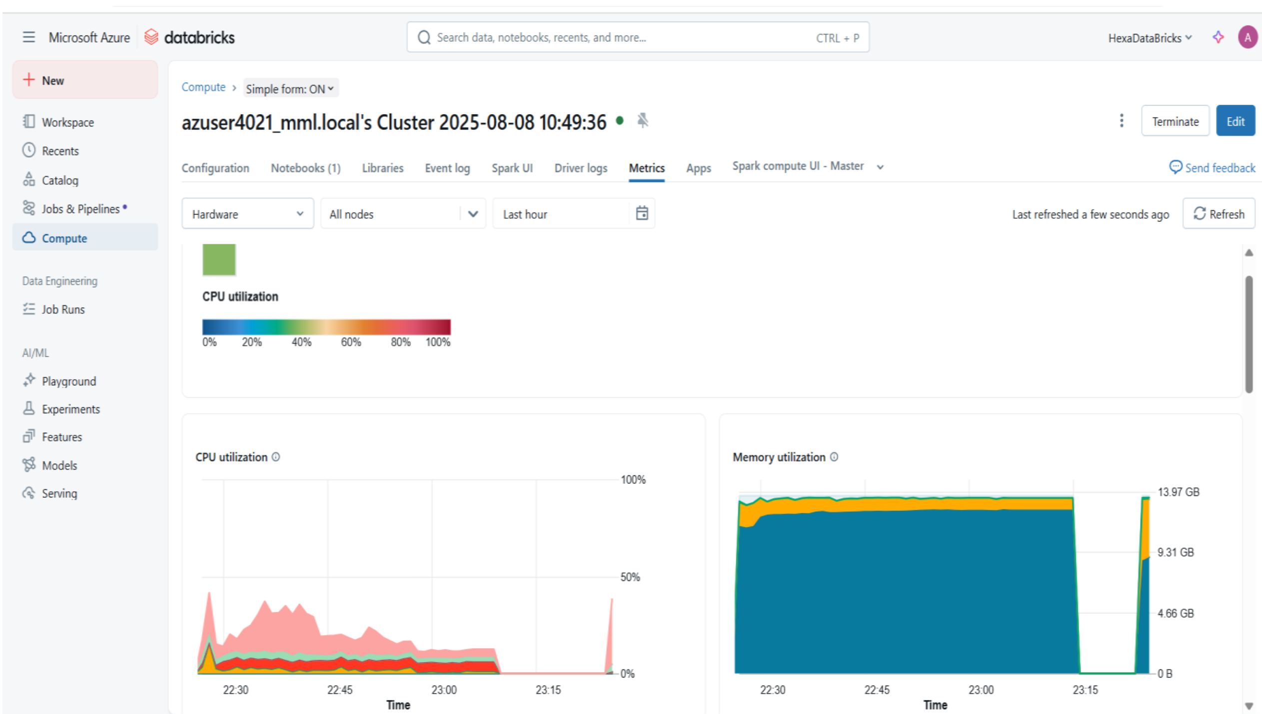


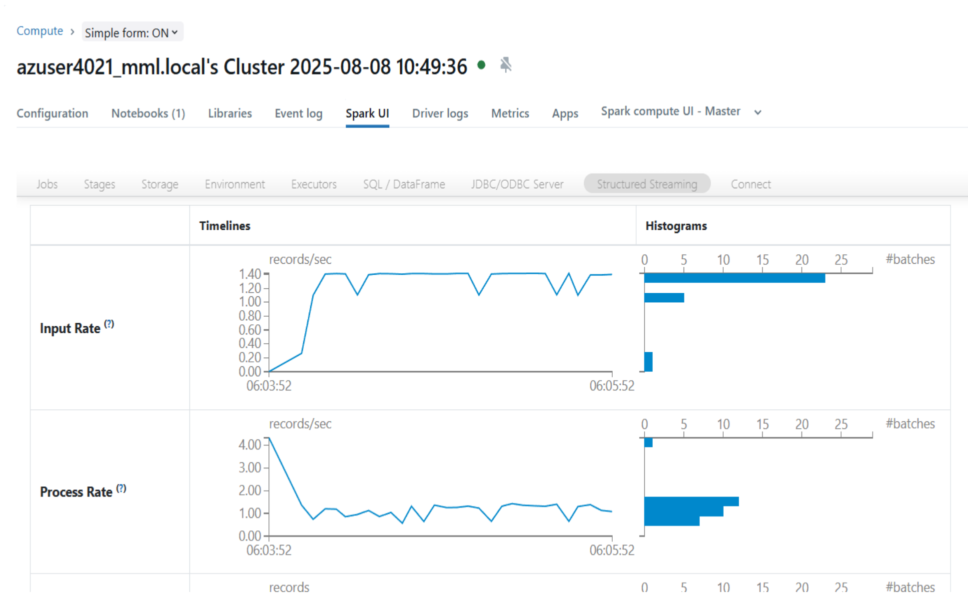
6. Verified outputs with .show() .

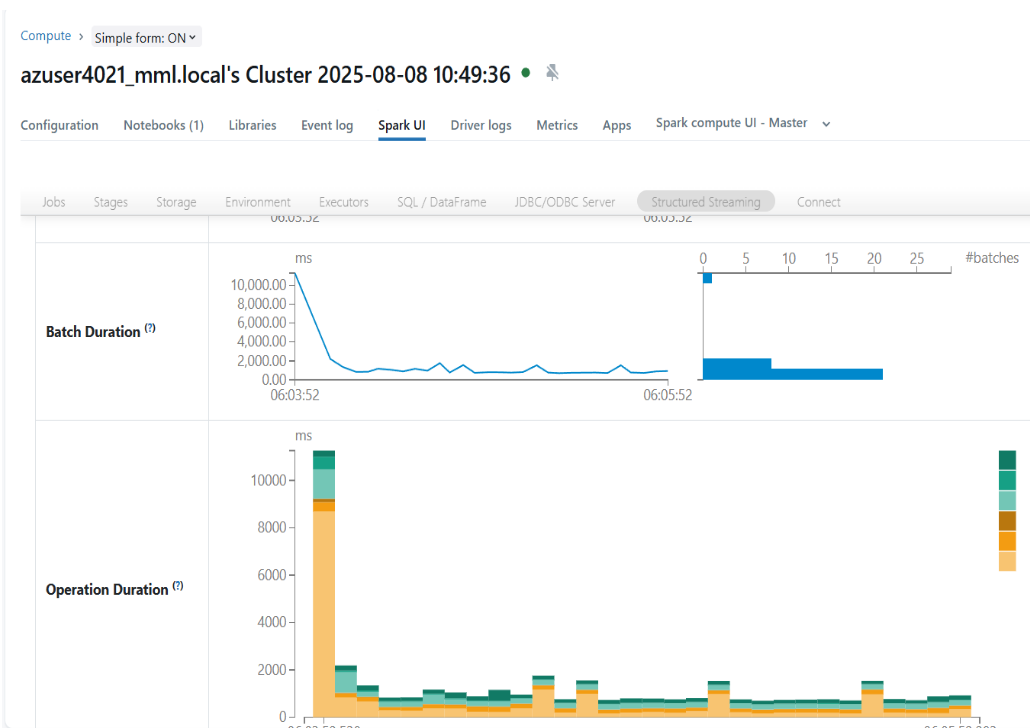




Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
90 (33228f1f-2ba3-4ef3-b493-7194593357f1)	display_query_2 id = 9b0e96e2-ec30-47e9-a06b-50e287baf98c runId = 33228f1f-2ba3-4ef3-b... start at DriverSparkHooks.scala:245	2025/08/23 17:55:36	0.9 s	2/2	5/5
88 (3bbde471-9b58-401d-8f8e-e5414a3c18a4)	display_query_1 id = 125b39c9-17f7-4f24-a519-f74dfba66ae1 runId = 3bbde471-9b58-401d-... start at DriverSparkHooks.scala:245	2025/08/23 17:55:35	1 s	2/2	5/5
87 (33228f1f-2ba3-4ef3-b493-7194593357f1)	display_query_2 id = 9b0e96e2-ec30-47e9-a06b-50e287baf98c runId = 33228f1f-2ba3-4ef3-b... start at DriverSparkHooks.scala:245	2025/08/23 17:55:34	0.9 s	2/2	5/5
86 (3bbde471-9b58-401d-8f8e-e5414a3c18a4)	display_query_1 id = 125b39c9-17f7-4f24-a519-f74dfba66ae1 runId = 3bbde471-9b58-401d-... start at DriverSparkHooks.scala:245	2025/08/23 17:55:33	0.9 s	2/2	5/5
85 (33228f1f-2ba3-4ef3-b493-7194593357f1)	display_query_2 id = 9b0e96e2-ec30-47e9-a06b-50e287baf98c runId = 33228f1f-2ba3-4ef3-b... start at DriverSparkHooks.scala:245	2025/08/23 17:55:32	1 s	2/2	5/5
84 (3bbde471-9b58-401d-8f8e-e5414a3c18a4)	display_query_1 id = 125b39c9-17f7-4f24-a519-f74dfba66ae1 runId = 3bbde471-9b58-401d-... start at DriverSparkHooks.scala:245	2025/08/23 17:55:31	0.9 s	2/2	5/5
83 (33228f1f-2ba3-4ef3-b493-7194593357f1)	display_query_2 id = 9b0e96e2-ec30-47e9-a06b-50e287baf98c runId = 33228f1f-2ba3-4ef3-b... start at DriverSparkHooks.scala:245	2025/08/23 17:55:30	0.9 s	2/2	5/5
82 (3bbde471-9b58-401d-8f8e-e5414a3c18a4)	display_query_1 id = 125b39c9-17f7-4f24-a519-f74dfba66ae1 runId = 3bbde471-9b58-401d-... start at DriverSparkHooks.scala:245	2025/08/23 17:55:29	0.9 s	2/2	5/5
81 (33228f1f-2ba3-4ef3-b493-7194593357f1)	display_query_2 id = 9b0e96e2-ec30-47e9-a06b-50e287baf98c runId = 33228f1f-2ba3-4ef3-b... start at DriverSparkHooks.scala:245	2025/08/23 17:55:28	1 s	2/2	5/5




Compute › Simple form: ON
azuser4021_mml.local's Cluster 2025-08-08 10:49:36
Configuration
Notebooks (1)
Libraries
Event log
Spark UI
Driver logs
Metrics
Apps
Spark compute UI - Master
Jobs
Stages
Storage
Environment
Executors
SQL / DataFrame
JDBC/ODBC Server
Structured Streaming
Connect
Timelines
Histograms
Input Rate (?)
records/sec
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
06:03:52
06:05:52
0 5 10 15 20 25
#batches
Process Rate (?)
records/sec
4.00
3.00
2.00
1.00
0.00
06:03:52
06:05:52
records


Compute › Simple form: ON
azuser4021_mml.local's Cluster 2025-08-08 10:49:36
Configuration
Notebooks (1)
Libraries
Event log
Spark UI
Driver logs
Metrics
Apps
Spark compute UI - Master
Jobs
Stages
Storage
Environment
Executors
SQL / DataFrame
JDBC/ODBC Server
Structured Streaming
Connect
Batch Duration (?)
ms
10,000.00
8,000.00
6,000.00
4,000.00
2,000.00
0.00
06:03:52
06:05:52
0 5 10 15 20 25
#batches
Operation Duration (?)
ms
10000
8000
6000
4000
2000
0

Strategies for Optimizing Process

1. **Data Pipeline Efficiency**

Delta Lake for Incremental Updates & Schema Evolution

- Instead of reprocessing entire datasets, Delta Lake allows only new/changed data (incremental loads) to be processed.
- Handles schema drift automatically (e.g., new fields from IoT sensors won't break pipelines).
- Benefits: Faster ingestion, reduced cost, reliable history tracking (time travel).

2. **Optimize Streaming Jobs with Window Functions & Watermarks**

- Window functions (like 5-min averages of vehicle speed) reduce data granularity while keeping insights useful.
- Watermarks manage late-arriving IoT events by defining how long to wait before discarding delayed data.
- Benefits: Reduces memory usage, ensures real-time insights remain accurate.

3. **Resource Optimization**

Auto-scaling Clusters in Databricks

- Dynamically scale compute resources up/down based on traffic load.
- During peak traffic hours, more nodes are allocated; off-peak hours reduce cluster size automatically.
- Benefits: Saves costs while ensuring performance during demand spikes.

4. Caching & Partitioning Frequently Used Datasets

- Cache intermediate results (like “average traffic speed per zone”) for faster reuse in multiple queries.

- Partition large datasets by date, location, or vehicle type so queries only scan relevant partitions.

- Benefits: Lower query latency, faster ETL jobs, reduced cluster workload.

5. Real-Time Performance

Event Hub Batch Optimization (Reduce Latency)

- Instead of processing individual events, group small events into batches before pushing to Databricks.

- Balances between throughput (fewer API calls) and latency (faster processing).

- Benefits: Smoother data flow, avoids bottlenecks during sudden surges in sensor data.

Apply Filtering & Aggregation at Ingestion

- Process raw IoT data at the ingestion layer (Event Hub/Stream Analytics) instead of downstream.

- Example: Only forward “vehicles over 100 km/h” or “congestion alerts” instead of all raw sensor data.

- Benefits: Reduces unnecessary storage, speeds up analysis, keeps Gold layer clean & focused.

6. Monitoring & Alerts

Implement Metrics Dashboards

Build dashboards (Databricks SQL) for real-time KPIs like:

- Avg. vehicle speed by road segment
- Congestion levels across city zones
- Number of alerts generated per hour

Benefits: Provides visibility to both traffic authorities and end-users.

7. Real-Time Anomaly Detection Alerts

- Use ML models or rule-based thresholds to detect unusual traffic (e.g., sudden standstill = possible accident).
- Trigger instant notifications via SMS, App, or Dashboard pop-ups.
- Benefits: Proactive response to incidents, improves road safety, builds user trust.

Conclusion

This project successfully demonstrated a real-time traffic monitoring system using Azure services. By leveraging Event Hub for ingestion and Databricks + Delta Lake for real-time analytics, the pipeline provides actionable insights into traffic speed and density. This architecture is scalable, fault-tolerant, and can be extended for smart city solutions like congestion alerts and predictive traffic flow.

Key Achievements of this Project

- **Real-time Ingestion with Azure Event Hub**: Successfully implemented seamless streaming of high-velocity IoT events (vehicle telemetry) into Azure Event Hub for reliable, low-latency data capture.

- **End-to-End Streaming with Azure Databricks**: Deployed Spark Structured Streaming pipelines in Databricks to process IoT events in near real-time with checkpointing, watermarking, and fault tolerance.

- **Delta Lake Multi-Layer Architecture**: Designed and implemented a robust Bronze–Silver–Gold Delta Lake architecture ensuring data quality, consistency, and optimized access for analytics.

- **Efficient Data Transformation & Aggregation**: Applied schema enforcement, timestamp conversion, and aggregations (e.g., average vehicle speed, congestion detection) to convert raw events into business-ready insights.

- **Future-Ready & Extensible**: Built a scalable architecture that can easily be extended to additional IoT use cases (e.g., traffic prediction, anomaly detection) and integrated with downstream analytics tools like Power BI or Synapse.