End-to-End Retail Analytics Platform on Azure, Databricks, Snowflake and PowerBI

A Capstone Project

Name - Shivendra Tripathi EMP ID - AS1550

1. Project Overview & Objectives

This project documents the design and implementation of a modern, end-to-end data analytics platform for a retail company. The solution addresses the common business challenge of integrating siloed data from various sources—including batch Point-of-Sale (POS) transactions and real-time online events—into a single, reliable source of truth.

Built on Microsoft Azure, Databricks,Snowflake and PowerBI, the platform leverages a Medallion Architecture with Delta Lake for robust and scalable data processing. The primary objectives of this project were to:

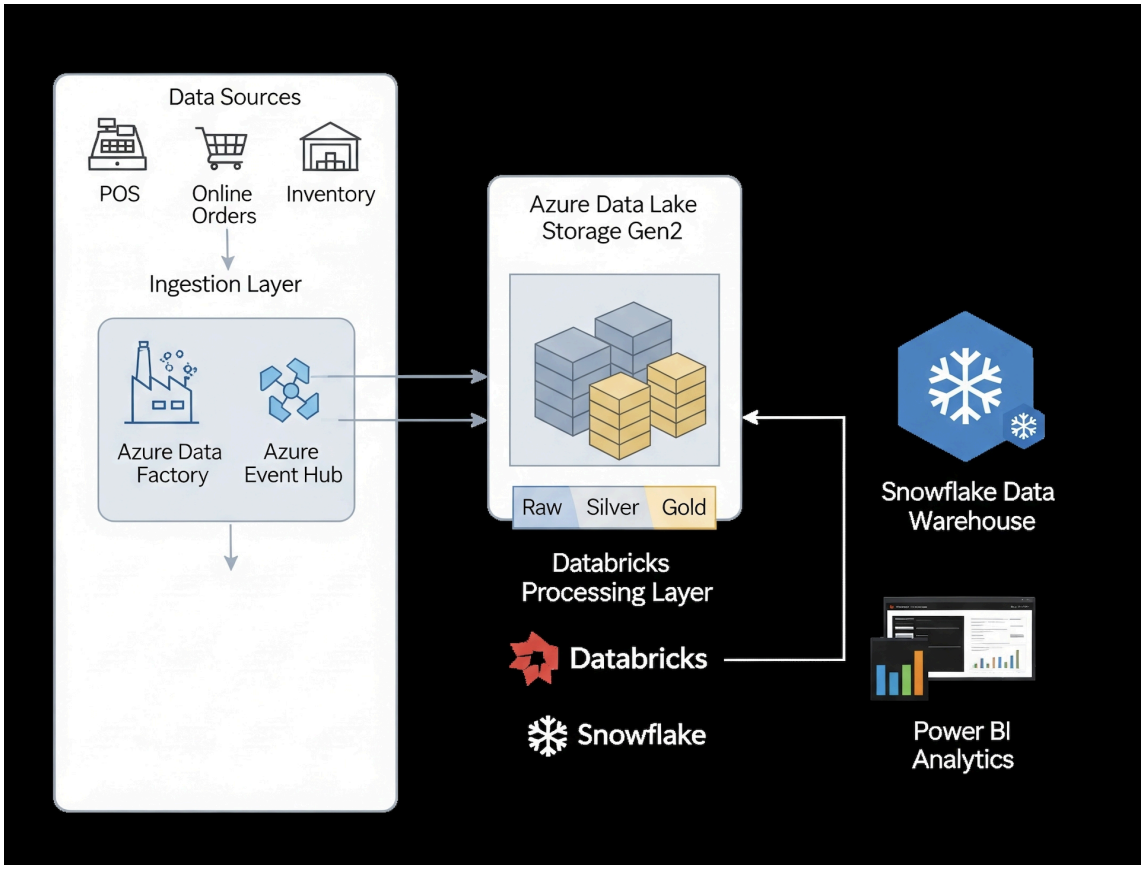
- **Automate Data Ingestion:** Build resilient pipelines for both batch and streaming data.
- **Ensure Data Quality:** Implement a multi-layered architecture (Bronze, Silver, Gold) to progressively cleanse, validate, and enrich the data.
- **Implement Data Security:** Secure sensitive customer data by masking Personally Identifiable Information (PII).
- **Enable Real-Time Analytics:** Develop a streaming pipeline to provide instant alerts for fraud and inventory shortages.
- **Deliver Actionable Insights:** Load the curated data into a Snowflake data warehouse and build interactive Power BI dashboards to facilitate data-driven decisions.

2. Solution Architecture

The platform follows a modern data lakehouse architecture, processing data through Bronze (raw), Silver (cleansed), and Gold (aggregated) layers.

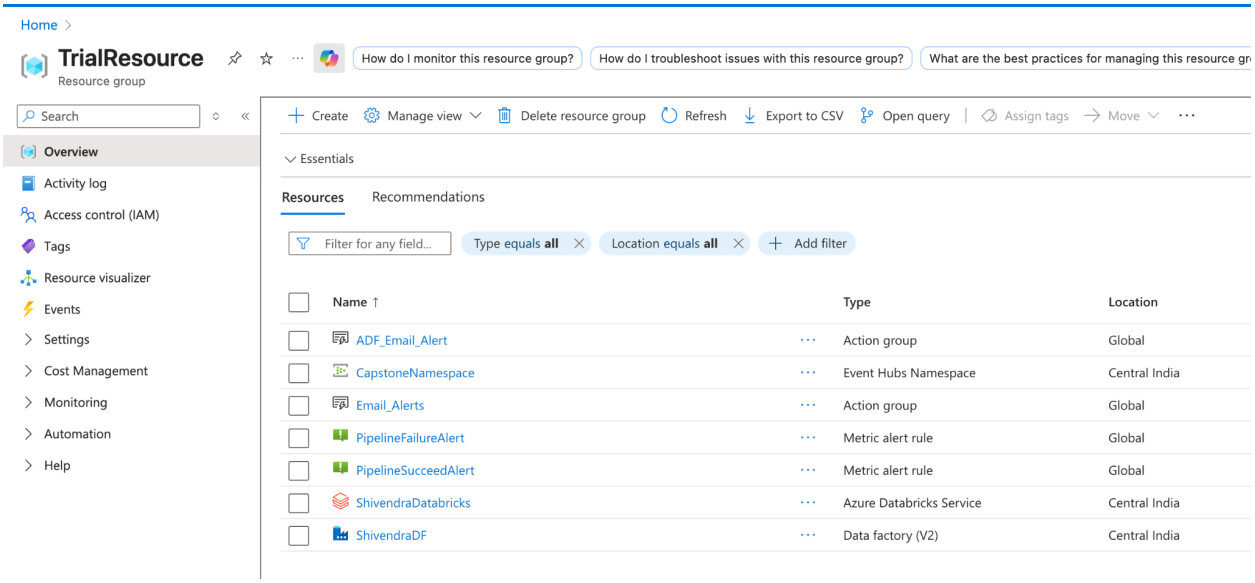
- **Ingestion:** Azure Data Factory (ADF) orchestrates batch ingestion, while Azure Event Hub captures real-time streaming data.
- **Storage:** Azure Data Lake Storage (ADLS) Gen2 serves as the central data lake for all three layers of the Medallion Architecture.
- **Transformation:** Azure Databricks provides the Spark engine for all transformations, from initial cleansing to final business aggregations. All data is stored in the Delta Lake format.
- **Data Warehousing:** Snowflake serves as the high-performance cloud data warehouse, providing a curated source for all analytics.
- **Business Intelligence:** Power BI connects to Snowflake to deliver live, interactive dashboards to business users.

Architecture :-



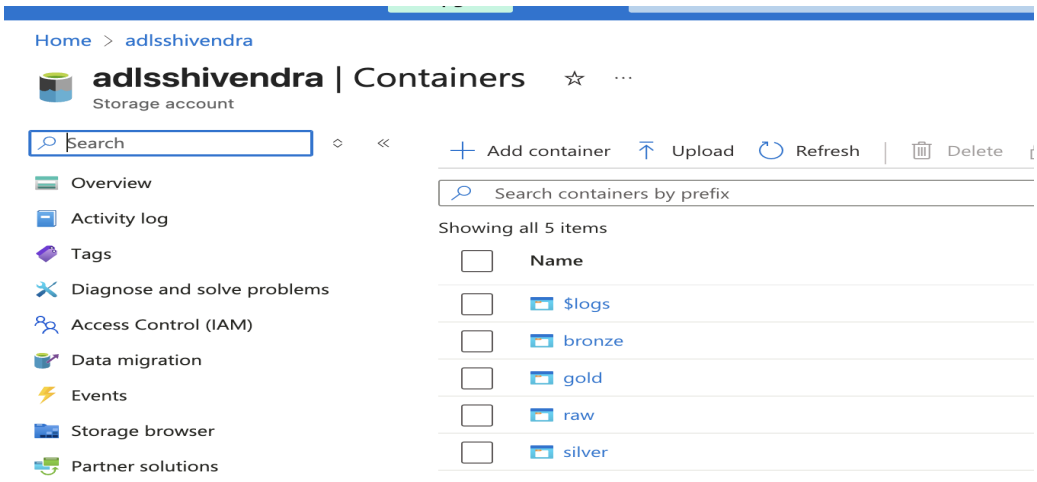
3. Resource Setup

Created a Resource Group “Trial Resource” in my Azure Subscription to host all the resources required.

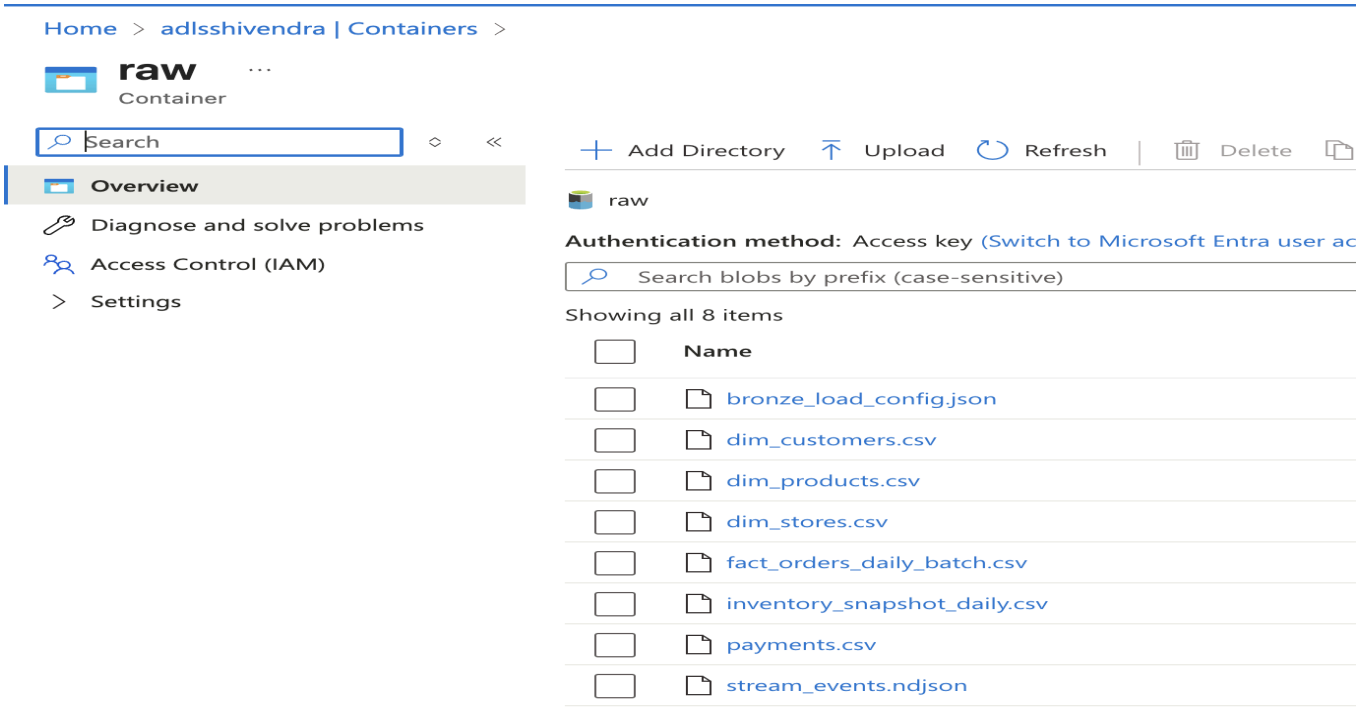


4. Data Ingestion

- **ADLS as a single source of truth, containing raw, bronze, silver and gold containers**



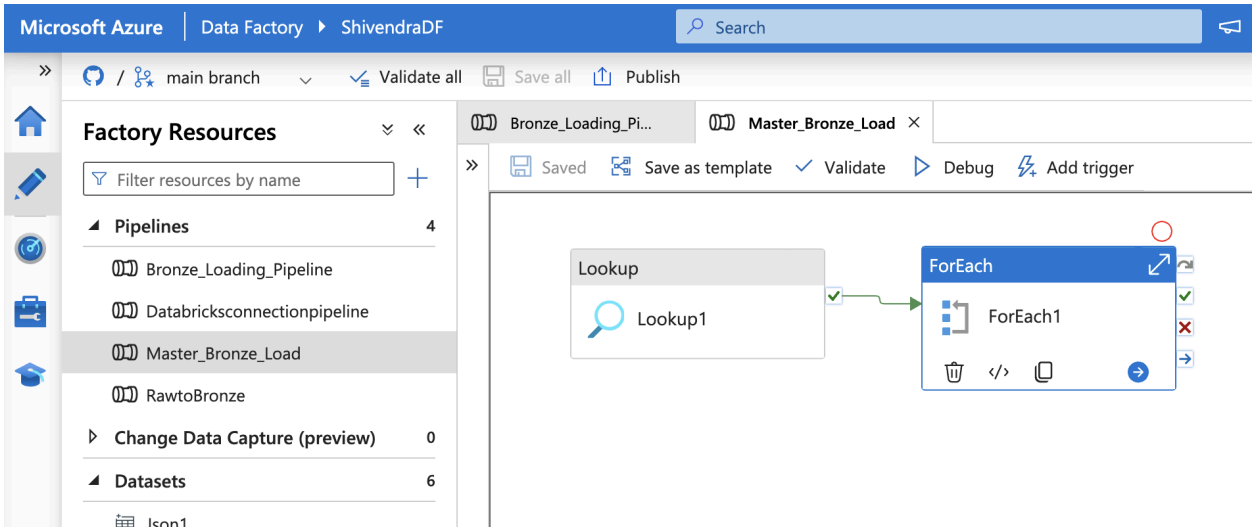
Raw Container:-



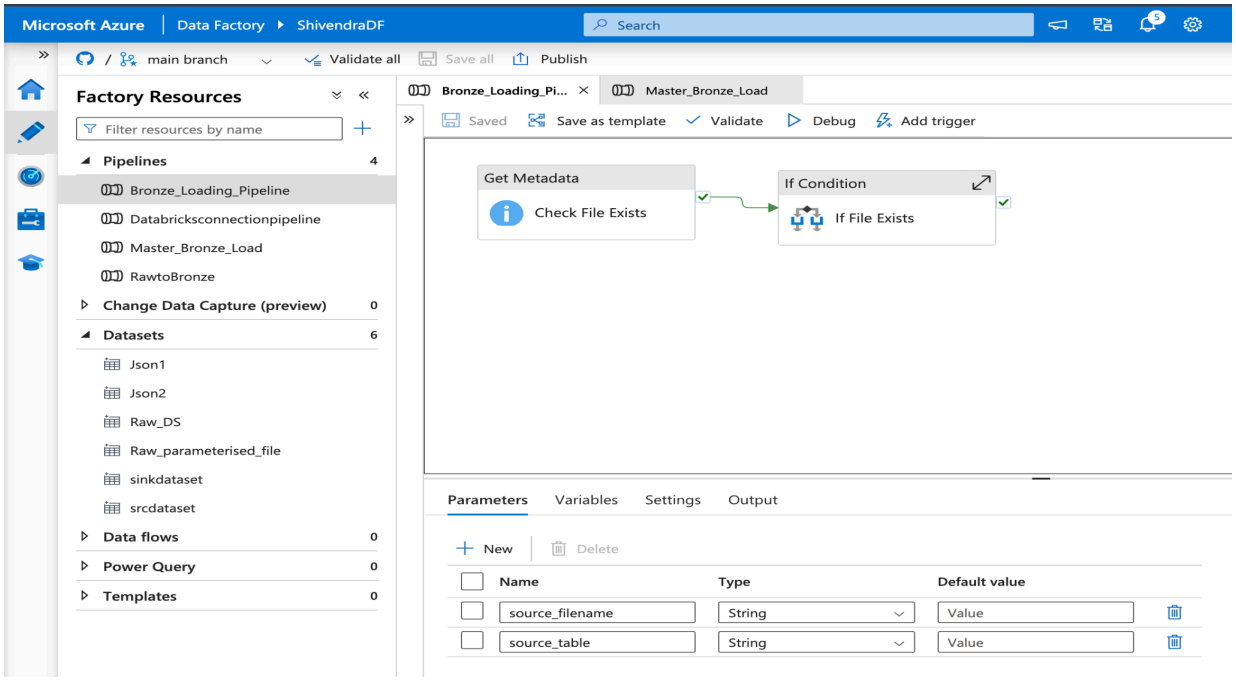
- **Batch Ingestion with Azure Data Factory**

An automated ADF pipeline was built to handle the daily batch ingestion of POS data. A Master/Worker pattern was implemented for reusability: a 'Master' pipeline reads a JSON config file and loops through all tables, calling a 'Worker' pipeline to execute a Databricks notebook for each one.

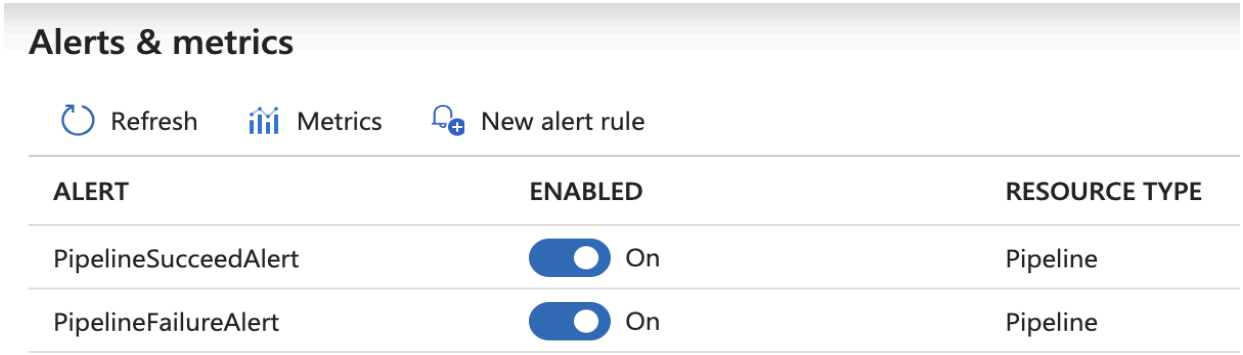
Master Pipeline (Master_Bronze_Load):-



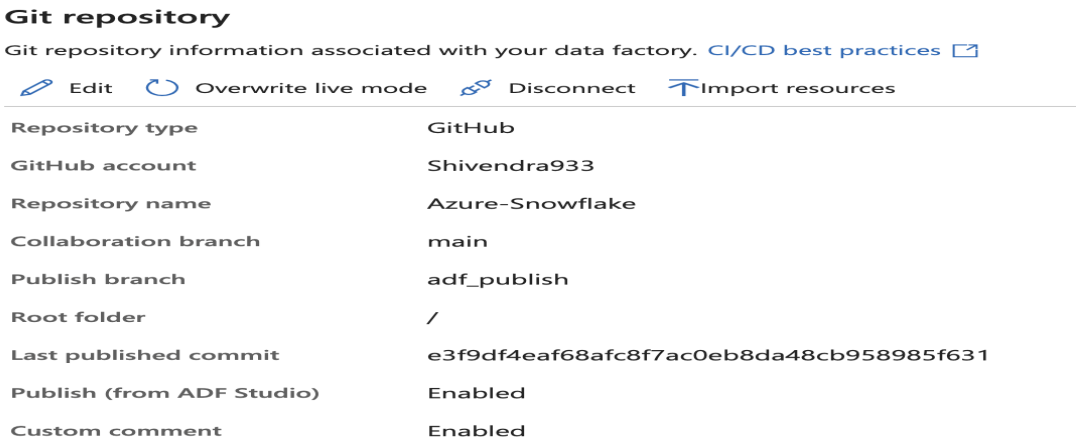
Worker Pipeline(Bronze_Loading_Pipeline):-



Alerts were set up for success or failure of the pipeline.



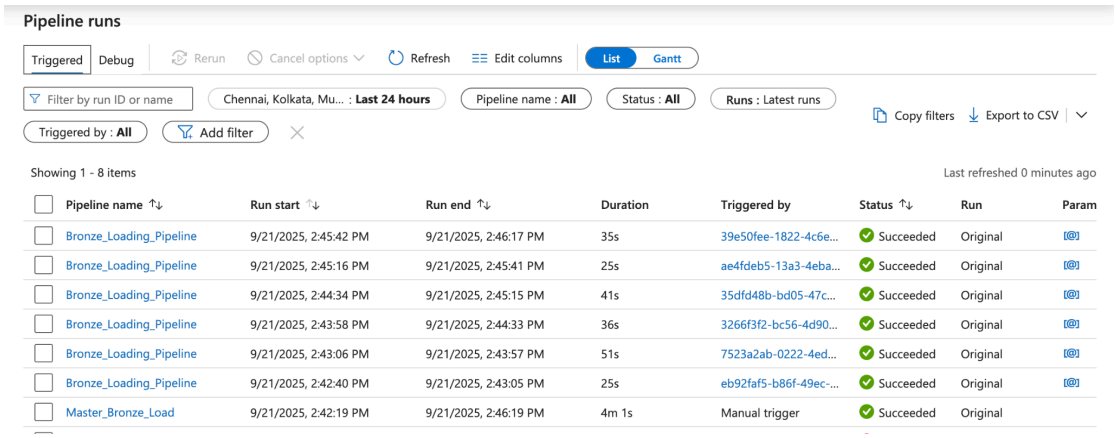
ADF was configured to Github for version control.



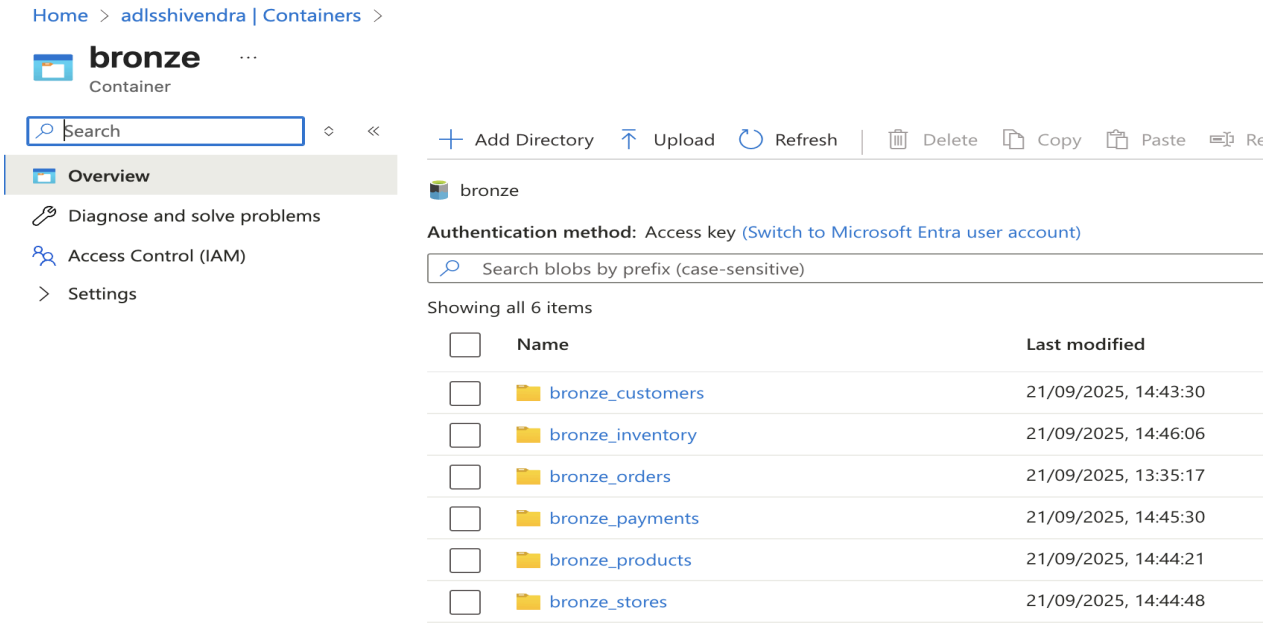
Configured the ADLS through Service Principal Client to connect to databricks

```
23 hours ago (<1s)                                                    2

#Configuring Service Principal

STORAGE_ACCOUNT_NAME = "adlsshivendra"
ADLS_CONTAINER_NAME = "raw"
CLIENT_ID = "4069d864-2ea2-4354-a634-71a8488be89e"
TENANT_ID = "4ac50105-0c66-404e-a107-7cbd8a9a6442"
CLIENT_SECRET = "o.O8Q~ARpL1hfZzcjC_yJZTPhhc-jTCDlI_BRdif"


#Setting the Spark configuration for this session
spark.conf.set(f"fs.azure.account.auth.type.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", "OAuth")
spark.conf.set(f"fs.azure.account.oauth.provider.type.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", "org.apache.hadoop.fs.azurebfs.oauth2.
ClientCredsTokenProvider")
spark.conf.set(f"fs.azure.account.oauth2.client.id.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", CLIENT_ID)
spark.conf.set(f"fs.azure.account.oauth2.client.secret.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", CLIENT_SECRET)
spark.conf.set(f"fs.azure.account.oauth2.client.endpoint.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", f"https://login.microsoftonline.com/
{TENANT_ID}/oauth2/token")
print("Spark config set for Service Principal.")

Spark config set for Service Principal.
```

Pipeline Run:-



Bronze container after loading:-

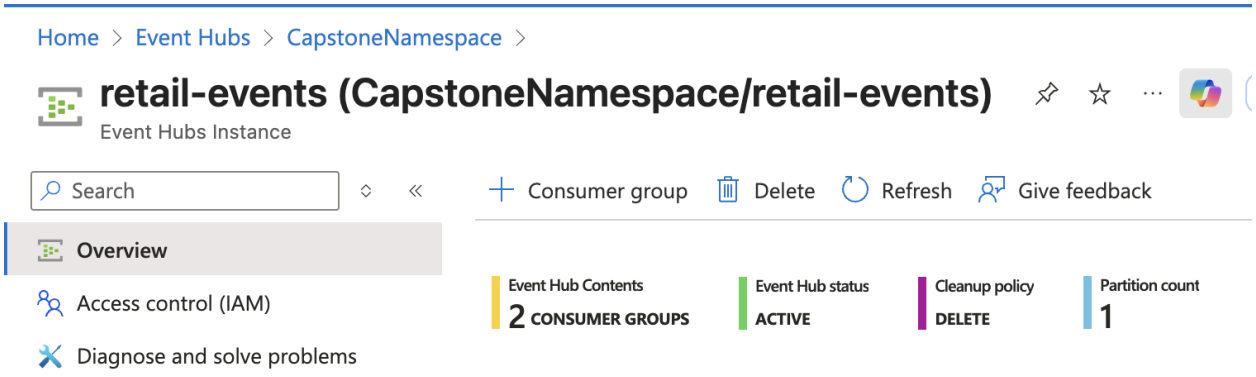


Stored the raw files as delta tables.

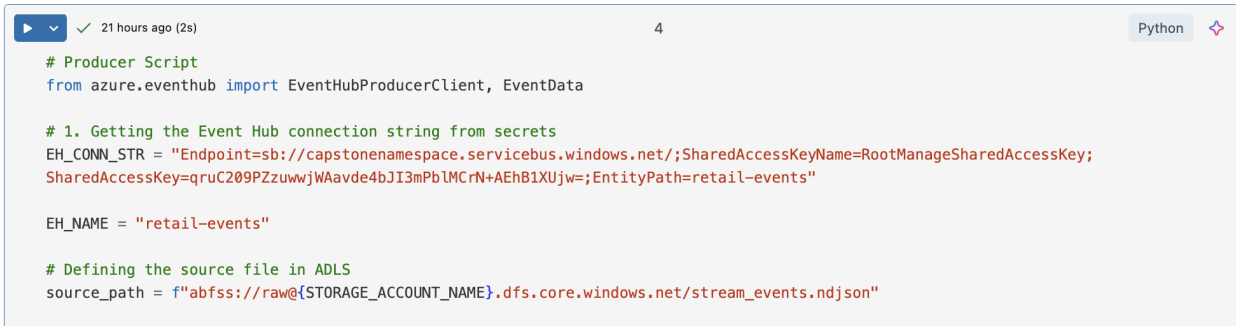
- **Real-Time Ingestion with Azure Event Hub**

To simulate a live feed of online orders and inventory updates, a 'Producer' notebook was created. This script reads the `stream_events.ndjson` file and publishes each event as a message to an Azure Event Hub named `retail-events`.

Create an Azure Event Hub Namespace and Event Hub



Configured the Eventhub to databricks using Primary Connection String



Sent all the events to Event hub using the below python script.

```
    # Reading the NDJSON file into a list
    print(f"Reading events from {source_path}...")
    try:
        events = spark.read.text(source_path).collect()
        print(f"Found {len(events)} events to send.")
    except Exception as e:
        print(f"Error reading from ADLS. Check Cell 2 config and file path.")
        raise e

    # Creating the producer client
    producer = EventHubProducerClient.from_connection_string(EH_CONN_STR, eventhub_name=EH_NAME)

    # 5. Sending the events in batches
    print("Connecting to Event Hub and sending events...")
    batch = producer.create_batch()
    total_sent = 0

    for i, event_row in enumerate(events):
        try:
            # Add the event (which is a JSON string) to the batch
            batch.add(EventData(event_row.value))
            total_sent += 1
        except ValueError:
            # Batch is full, send it and create a new one
            producer.send_batch(batch)
            print(f"  ...sent batch {i // 100}...")
            batch = producer.create_batch()
            batch.add(EventData(event_row.value))

    # Sending any remaining events in the last batch
    if len(batch) > 0:
        producer.send_batch(batch)

    producer.close()
    print("-" * 30)
    print(f"All {total_sent} events sent successfully.")
```

▸ (1) Spark Jobs

```
Reading events from abfss://raw@adlsshivendra.dfs.core.windows.net/stream_events.ndjson...
Found 2160 events to send.
Connecting to Event Hub and sending events...
------------------------------
All 2160 events sent successfully.
```

4. Data Transformation: The Medallion Architecture

- **Bronze Layer: Raw Data Foundation**

The `bronze_ingestion` Databricks notebook reads the raw source files (e.g., CSVs) and writes them as Delta tables into the `bronze` container in ADLS Gen2. This provides a permanent, raw backup of all source data. The `.option("path", ...)` command was used to ensure the physical data resides in our ADLS container.

```
#Read, Drop, and Write Table

if not p_file_name or not p_table_name:
    print("Parameters are empty. Skipping main logic.")
else:
    print(f"Processing {p_file_name}...")
    try:
        # 1. Read the source CSV
        df = spark.read.format("csv") \
            .option("header", "true") \
            .option("inferSchema", "true") \
            .load(source_path)

        # 2. Drop any old, broken registration (THIS IS A KEY FIX)
        print(f"    Dropping old registration for {p_table_name}...")
        spark.sql(f"DROP TABLE IF EXISTS {p_table_name}")

        # 3. Write to the correct ADLS location AND register the table
        print(f"    Writing to {target_path} and registering table...")
        df.write.mode("overwrite") \
          .format("delta") \
          .option("path", target_path) \
          .saveAsTable(p_table_name)

        print(f"  SUCCESS: '{p_table_name}' created at {target_path}.\n")

    except Exception as e:
        print(f"  ERROR processing '{p_file_name}': {e}\n")
        raise e
```

- **Silver Layer: Cleansing and Enrichment**

The `silver_gold_transformations` notebook reads the Bronze tables from ADLS bronze using service principal client and performs key transformations:

- **PII Masking:** Customer `Email` and `Phone` were masked to meet security requirements.

15 hours ago (4s) 11

```
# Masking Email and Phone no. of customers
df_silver_customers = df_customers \
    .withColumn("Email", regexp_replace(col("Email"), r"^(.*)@", "XXXX@")) \
    .withColumn("Phone", regexp_replace(col("Phone").cast("string"), r"\d(?=\d{4})", "X"))

df_silver_customers.show(15, truncate=False)

#ADLS silver path for customer table
silver_customers_path = f"abfss://silver@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/silver_customers"

# Saving the silver customers table into ADLS silver
df_silver_customers.write.mode("overwrite") \
    .format("delta") \
    .option("path", silver_customers_path) \
    .saveAsTable("silver_customers")
```

(2) Spark Jobs

df_silver_customers: pyspark.sql.dataframe.DataFrame = [CustomerID: integer, FirstName: string ... 9 more fields]

CustomerID	FirstName	LastName	Email	Phone	LoyaltyPoints	Tier	SignupDate	City	State	PostalCode
1	Vivaan	Singh	XXXX@example.com	XXXXXXXX7923	547	Silver	2023-12-29	Nagpur	IN	432043
2	Pooja	Verma	XXXX@example.com	XXXXXXXX0525	559	Bronze	2021-05-21	Mumbai	IN	761820
3	Advika	Patel	XXXX@example.com	XXXXXXXX3256	456	Bronze	2021-07-05	Ahmedabad	IN	203225
4	Ananya	Chettri	XXXX@example.com	XXXXXXXX6825	490	Silver	2022-09-06	Jaipur	IN	130353
5	Vihaan	Singh	XXXX@example.com	XXXXXXXX5019	336	Bronze	2021-04-25	Bhopal	IN	475876

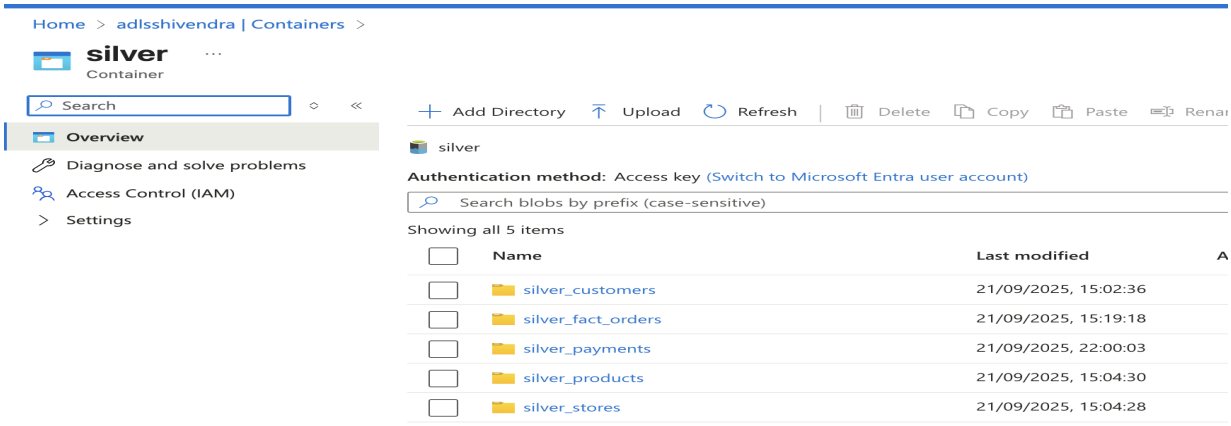
- **Enrichment:** The `silver_fact_orders` table was created by joining 5 separate tables, resolving 5 column-name conflicts (e.g., `City` -> `StoreCity`) in the process.

22 hours ago (5s) 13

```
# Creating Enriched Fact Table (Silver)

# Cleansing and rename the base fact table
df_orders_renamed = df_orders \
    .withColumn("DiscountPct", coalesce(col("DiscountPct"), lit(0))) \
    .withColumnRenamed("UnitPrice", "SaleUnitPrice") \
    .withColumnRenamed("Status", "OrderStatus") \
    .withColumnRenamed("Currency", "OrderCurrency")

# Rename conflicting columns on ALL dimension tables
df_stores_renamed = df_stores \
    .withColumnRenamed("City", "StoreCity") \
    .withColumnRenamed("State", "StoreState")

df_customers_renamed = df_customers \
    .withColumnRenamed("City", "CustomerCity") \
    .withColumnRenamed("State", "CustomerState")

df_products_renamed = df_products \
    .withColumnRenamed("UnitPrice", "ProductListPrice")

df_payments_renamed = df_payments \
    .withColumnRenamed("Status", "PaymentStatus") \
    .withColumnRenamed("Currency", "PaymentCurrency")

# Joining all the renamed tables
df_silver_fact_orders = df_orders_renamed \
    .join(df_payments_renamed, ["OrderID", "PaymentID"]) \
    .join(df_stores_renamed, "StoreID") \
    .join(df_products_renamed, "ProductID") \
    .join(df_customers_renamed, "CustomerID")

# Defining the ADLS path
silver_fact_path = f"abfss://silver@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/silver_fact_orders"


# Writing to ADLS Silver
df_silver_fact_orders.write.mode("overwrite") \
    .format("delta") \
    .option("path", silver_fact_path) \
    .saveAsTable("silver_fact_orders")
```

(5) Spark Jobs

Updated Silver Container:-



- **Gold Layer: Business-Ready Aggregates**

To optimize for BI performance, 7 pre-aggregated Gold tables were created. These tables, such as `gold_channel_payments`, are small and fast to query, allowing Power BI dashboards to load instantly.

22 hours ago (3s) 18

```
# 3. Sales by Channel & Payment

df_gold_channel_payment = df_silver_base \
    .groupBy("Channel", "PaymentMethod") \
    .agg(
        count("OrderID").alias("OrderCount"),
        round(sum("OrderTotal"),2).alias("TotalRevenue")
    )

df_gold_channel_payment.show(truncate=False)

# Defining path and saving to Gold container
gold_path = f"abfss://gold@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/gold_sales_by_channel_payment"
spark.sql("DROP TABLE IF EXISTS gold_sales_by_channel_payment")
df_gold_channel_payment.write.mode("overwrite") \
    .format("delta") \
    .option("path", gold_path) \
    .saveAsTable("gold_sales_by_channel_payment")
```

(4) Spark Jobs

df_gold_channel_payment: pyspark.sql.dataframe.DataFrame = [Channel: string, PaymentMethod: string ... 2 more fields]

```
+-------+-------------+----------+------------+
|Channel|PaymentMethod|OrderCount|TotalRevenue|
+-------+-------------+----------+------------+
|POS    |CARD         |10608     |898448.09   |
|POS    |UPI          |6286      |519853.15   |
|Online |WALLET       |937       |77980.79    |
|Online |CARD         |4492      |393439.34   |
|Online |UPI          |2635      |224738.6    |
|Online |CASH         |911       |72565.02    |
|POS    |CASH         |2042      |180323.42   |
|POS    |WALLET       |2089      |174415.62   |
+-------+-------------+----------+------------+
```

5. Real-Time Streaming Logic

The `streaming_consumer` notebook connects to the Event Hub using Connection string and to the ADLS using Service Principal Client and processes events in real-time.

Event Hub Configuration:-

```
21 hours ago (<1s)                                                    3

# Configuring Event Hub Connection

# --- Hardcode your Event Hub connection string ---
EH_CONN_STR = "Endpoint=sb://capstonenamespace.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;
SharedAccessKey=qruC209PZzuwwjWAavde4bJI3mPblMCrN+AEhB1XUjw=;EntityPath=retail-events"
CONSUMER_GROUP = "databricks-consumer"

# #Encrypting the connection string for the Spark workers
encrypted_conn_str = sc._jvm.org.apache.spark.eventhubs.EventHubsUtils.encrypt(EH_CONN_STR)

event_hub_config = {
  # --- Use the ENCRYPTED string ---
  'eventhubs.connectionString' : encrypted_conn_str,
  'eventhubs.consumerGroup' : CONSUMER_GROUP,

  # --- Use the FULL JSON string that provides all keys ---
  'eventhubs.startingPosition' : '{"offset": "-1", "seqNo": -1, "enqueuedTime": null, "isInclusive": true}'
}


print("Event Hub configuration is set.")
```

```
Event Hub configuration is set.
```

ADLS Connection:-

```
21 hours ago (1s)                                                     2

# Configuring Service Principal

STORAGE_ACCOUNT_NAME = "adlsshivendra"
ADLS_CONTAINER_NAME = "raw"
CLIENT_ID = "4069d864-2ea2-4354-a634-71a8488be89e"
TENANT_ID = "4ac50105-0c66-404e-a107-7cbd8a9a6442"
CLIENT_SECRET = "o.O8Q~ARpL1hfZzcjC_yJZTPhhc-jTCDlI_BRdif"

# --- Setting the Spark configs ---
spark.conf.set(f"fs.azure.account.auth.type.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", "OAuth")
spark.conf.set(f"fs.azure.account.oauth.provider.type.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", "org.apache.hadoop.fs.azurebfs.oauth2.
ClientCredsTokenProvider")
spark.conf.set(f"fs.azure.account.oauth2.client.id.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", CLIENT_ID)
spark.conf.set(f"fs.azure.account.oauth2.client.secret.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", CLIENT_SECRET)
spark.conf.set(f"fs.azure.account.oauth2.client.endpoint.{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net", f"https://login.microsoftonline.com/{TENANT_ID}/oauth2/
token")

print("Service Principal configured for ADLS access.")
```

```
Service Principal configured for ADLS access.
```

Reading from Event Hub:-

21 hours ago 4

```
from pyspark.sql.functions import *

# Reading from Event Hub
df_stream_raw = spark.readStream \
    .format("eventhubs") \
    .options(**event_hub_config) \
    .load()

# Parsing the common JSON fields
# The 'body' is the raw data, we cast it to a string
df_parsed = df_stream_raw \
    .withColumn("body", col("body").cast("string")) \
    .withColumn("event_type", get_json_object(col("body"), "$.event_type")) \
    .withColumn("event_time", get_json_object(col("body"), "$.event_time").cast("timestamp")) \
    .withColumn("payload", get_json_object(col("body"), "$.payload")) # Payload is still a JSON string

print("Stream parser defined.")
display(df_parsed) # Run this cell to see the stream (you may need to stop it)
```

(1) Spark Jobs

display_query_11 (id: 99f9a690-e1a1-4caa-ba09-2ff6b555da4e) *Last updated: 21 hours ago*

Table +

	body	partition	offset	sequenceNumber	enqueue
39	{"event_time": "2025-08-15T00:19:00", "event_type": "order_created", "payload": {"OrderID": "S175521714038", "StoreID...	0	10064	38	2025-09-21
40	{"event_time": "2025-08-15T00:19:30", "event_type": "order_created", "payload": {"OrderID": "S175521717039", "StoreID...	0	10360	39	2025-09-21
41	{"event_time": "2025-08-15T00:20:00", "event_type": "order_created", "payload": {"OrderID": "S175521720040", "StoreID...	0	10656	40	2025-09-21
42	{"event_time": "2025-08-15T00:20:30", "event_type": "order_created", "payload": {"OrderID": "S175521723041", "StoreID...	0	10960	41	2025-09-21
43	{"event_time": "2025-08-15T00:21:00", "event_type": "order_created", "payload": {"OrderID": "S175521726042", "StoreID...	0	11256	42	2025-09-21
44	{"event_time": "2025-08-15T00:21:30", "event_type": "order_created", "payload": {"OrderID": "S175521729043", "StoreID...	0	11552	43	2025-09-21

- **Fraud Detection:** The stream is filtered for `order_created` events. A rule (`OrderValue > 1000`) is applied, and any matching events are written to the `gold_fraud_alerts` table.

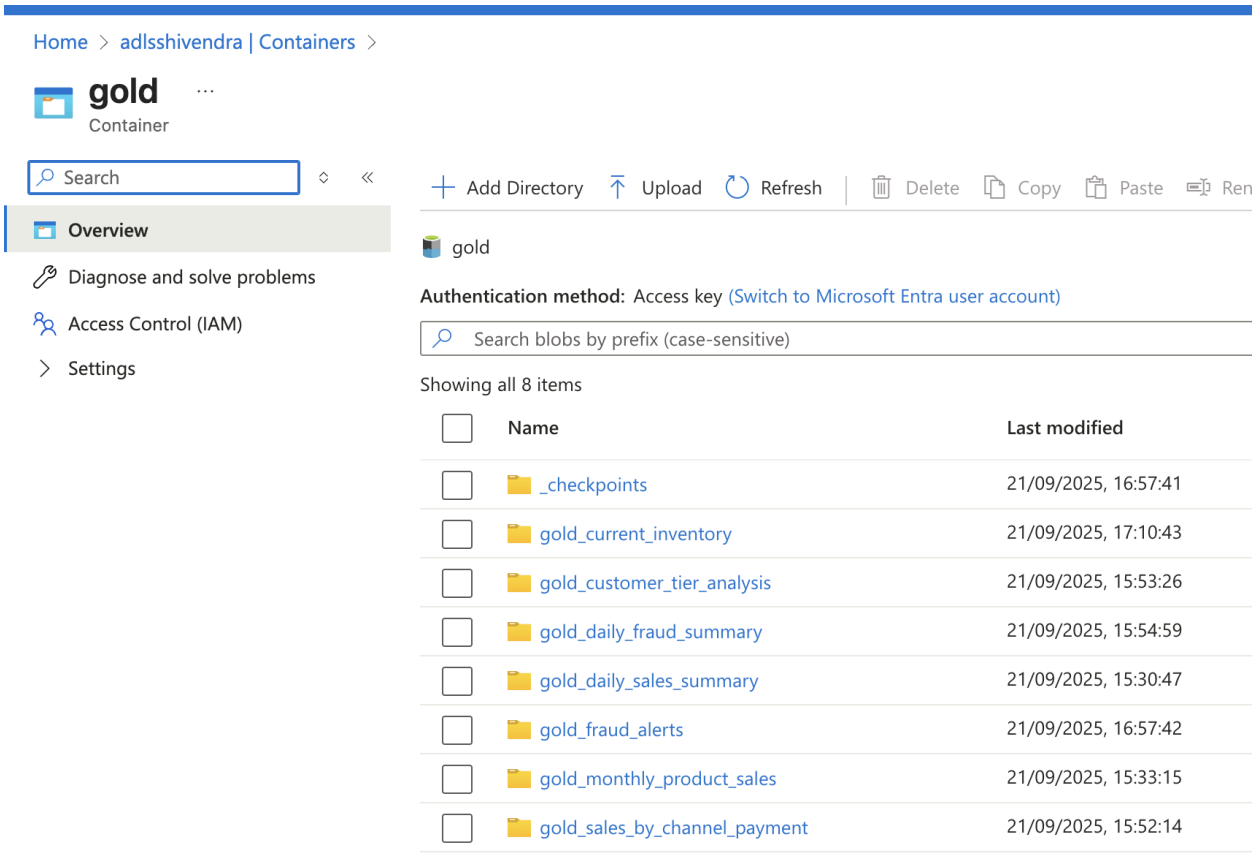
21 hours ago 5

```
# Defining the path for the fraud alerts table
fraud_alerts_path = f"abfss://gold@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/gold_fraud_alerts"
fraud_alerts_checkpoint = f"abfss://gold@{STORAGE_ACCOUNT_NAME}.dfs.core.windows.net/_checkpoints/gold_fraud_alerts"

# Defining the stream
df_orders_stream = df_parsed \
    .filter("event_type = 'order_created'") \
    .withColumn("Quantity", get_json_object(col("payload"), "$.Quantity").cast("int")) \
    .withColumn("UnitPrice", get_json_object(col("payload"), "$.UnitPrice").cast("double")) \
    .withColumn("OrderID", get_json_object(col("payload"), "$.OrderID")) \
    .withColumn("Channel", get_json_object(col("payload"), "$.Channel"))

# Applying the fraud rule (any order over $1000)
df_fraud_alerts = df_orders_stream \
    .withColumn("OrderValue", col("Quantity") * col("UnitPrice")) \
    .filter("OrderValue > 1000")

# Writing this stream to its Gold table
fraud_stream_writer = df_fraud_alerts.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", fraud_alerts_checkpoint) \
    .trigger(processingTime="1 minute") \
    .start(fraud_alerts_path)

print(f"Fraud alerts stream started. Writing to: {fraud_alerts_path}")
```

(1) Spark Jobs

- **Live Inventory Updates:** The stream filters for `inventory_update` events. A `MERGE INTO` command updates the `gold_current_inventory` table by adding the `DeltaQty` from the stream to the current `OnHandQty`. This was a key technical challenge, and the `DELTA_MULTIPLE_SOURCE_ROW_MATCHING_TARGET_ROW_IN_MERGE` error was resolved by pre-aggregating the streaming micro-batch.

Updated Gold Container:-



6. Data Warehousing with Snowflake

All 12 Silver (Star Schema) and Gold (Aggregates) tables were loaded from Databricks into a Snowflake data warehouse. Snowflake serves as the central, high-performance 'serving layer' for all BI and analytics users, separating our BI compute workloads from our data engineering workloads.

Created Warehouse, Database and Schema in Snowflake

RETAIL_DB.ANALYTICS Settings

```
CREATE OR REPLACE WAREHOUSE COMPUTE_WH
WITH
WAREHOUSE_SIZE='MEDIUM'
AUTO_SUSPEND=900
AUTO_RESUME=TRUE;

CREATE OR REPLACE DATABASE RETAIL_DB;
CREATE OR REPLACE SCHEMA ANALYTICS;
```

Setted Up the connection between databricks and snowflake for loading

16 hours ago (2s)

```
# Setting up Snowflake Connection

sf_user = "SHIVENDRA933"
sf_password = "Hilux@10000000"
sf_url = "ZBNVGCJ-KR75153.snowflakecomputing.com"
sf_warehouse = "COMPUTE_WH"
sf_database = "RETAIL_DB"
sf_schema = "ANALYTICS"

#Creating the Snowflake options dictionary
snowflake_options = {
  "sfUrl": sf_url,
  "sfUser": sf_user,
  "sfPassword": sf_password,
  "sfDatabase": sf_database,
  "sfSchema": sf_schema,
  "sfWarehouse": sf_warehouse
}

print("Snowflake options are set.")
```

```
Snowflake options are set.
```

Loading silver tables to snowflake

16 hours ago (22s)

```
#Loading 'silver_fact_orders' to Snowflake

table_to_load = "silver_fact_orders"
snowflake_table_name = "FACT_ORDERS"

# 1. Reading the Delta table from Databricks
df = spark.table(table_to_load)

# 2. Writing the DataFrame to Snowflake
df.write \
  .format("snowflake") \
  .options(**snowflake_options) \
  .option("dbtable", snowflake_table_name) \
  .mode("overwrite") \
  .save()

print(f"Successfully loaded {snowflake_table_name}.")
```

(1) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [CustomerID: integer, ProductID: integer ... 43 more fields]

```
Successfully loaded FACT_ORDERS.
```

Loaded Gold Tables to snowflake

16 hours ago (20s)

```
# List of Gold tables to load
tables_to_load = [
    "gold_current_inventory",
    "gold_customer_tier_analysis",
    "gold_daily_fraud_summary",
    "gold_daily_sales_summary",
    "gold_fraud_alerts",
    "gold_monthly_product_sales",
    "gold_sales_by_channel_payment"
]

# Loading using loop
for table in tables_to_load:
    snowflake_table_name = table.upper()  # convert to uppercase for Snowflake


    # Reading the Delta table
    df = spark.table(table)

    # Writing to Snowflake
    df.write \
      .format("snowflake") \
      .options(**snowflake_options) \
      .option("dbtable", snowflake_table_name) \
      .mode("overwrite") \
      .save()

    print(f"Successfully loaded {snowflake_table_name}.")
```

(7) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Channel: string, PaymentMethod: string ... 2 more fields]

```
Successfully loaded GOLD_CURRENT_INVENTORY.
Successfully loaded GOLD_CUSTOMER_TIER_ANALYSIS.
Successfully loaded GOLD_DAILY_FRAUD_SUMMARY.
Successfully loaded GOLD_DAILY_SALES_SUMMARY.
Successfully loaded GOLD_FRAUD_ALERTS.
Successfully loaded GOLD_MONTHLY_PRODUCT_SALES.
Successfully loaded GOLD_SALES_BY_CHANNEL_PAYMENT.
```

Verified if all the tables are loaded correctly

```
12  SHOW TABLES;
13
14
```

Results | Chart

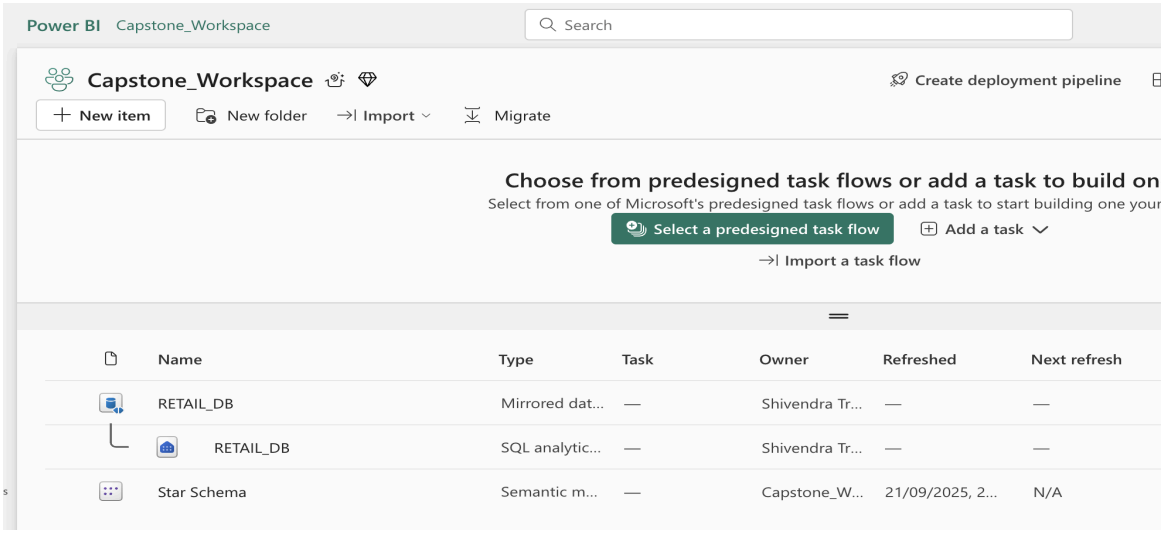
	cre	name	database_name	schema_name	kind
1	2025-	DIM_CUSTOMERS	RETAIL_DB	ANALYTICS	TABLE
2	2025-	DIM_PAYMENTS	RETAIL_DB	ANALYTICS	TABLE
3	2025-	DIM_PRODUCTS	RETAIL_DB	ANALYTICS	TABLE
4	2025-	DIM_STORES	RETAIL_DB	ANALYTICS	TABLE
5	2025-	FACT_ORDERS	RETAIL_DB	ANALYTICS	TABLE
6	2025-	GOLD_CURRENT_INVENTORY	RETAIL_DB	ANALYTICS	TABLE
7	2025-	GOLD_CUSTOMER_TIER_ANALYSIS	RETAIL_DB	ANALYTICS	TABLE
8	2025-	GOLD_DAILY_FRAUD_SUMMARY	RETAIL_DB	ANALYTICS	TABLE
9	2025-	GOLD_DAILY_SALES_SUMMARY	RETAIL_DB	ANALYTICS	TABLE
10	2025-	GOLD_FRAUD_ALERTS	RETAIL_DB	ANALYTICS	TABLE
11	2025-	GOLD_MONTHLY_PRODUCT_SALES	RETAIL_DB	ANALYTICS	TABLE
12	2025-	GOLD_SALES_BY_CHANNEL_PAYMENT	RETAIL_DB	ANALYTICS	TABLE

All tables are loaded in Snowflake

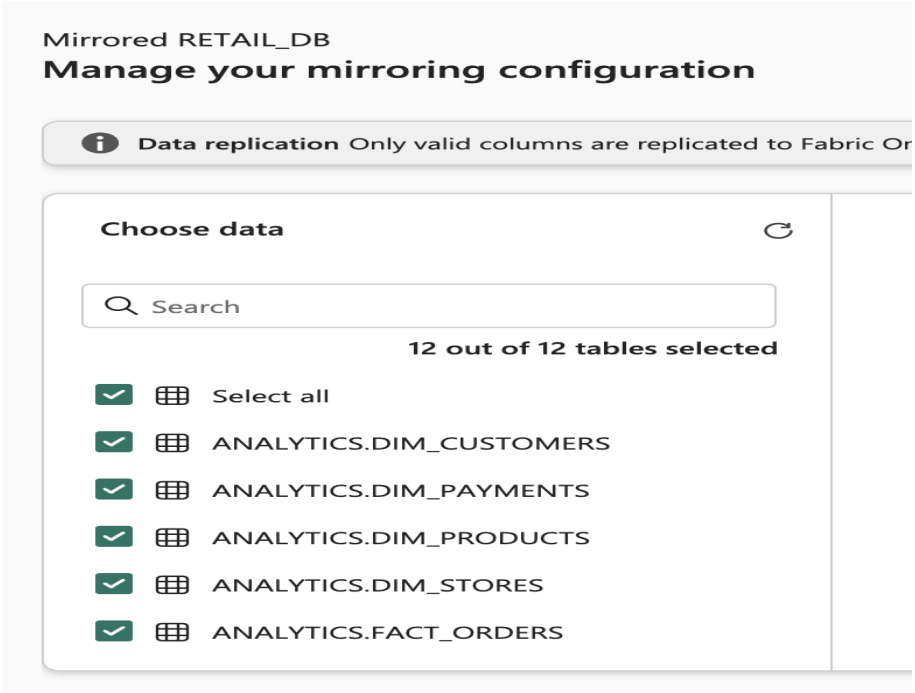
7. Business Intelligence with Power BI

Power BI was connected to Snowflake to deliver the final insights.

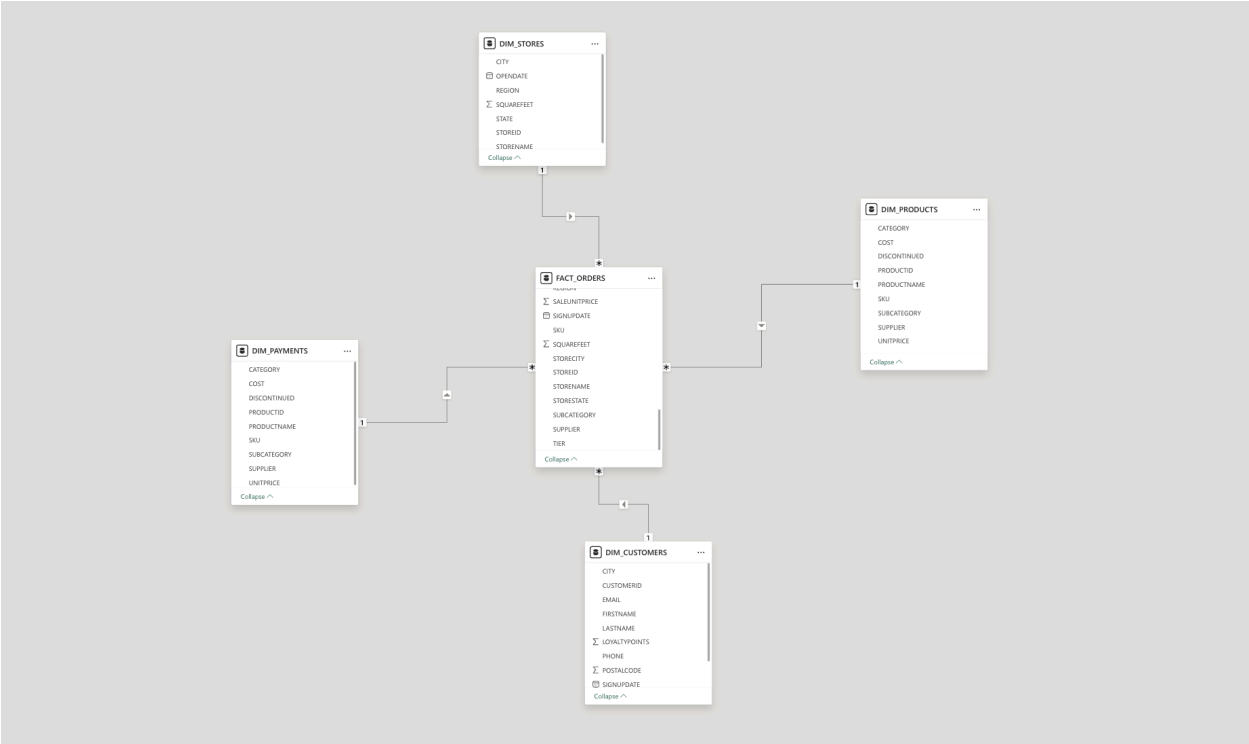
Created a workspace in PowerBI and connected it with Snowflake



Mirrored all the 12 tables in PowerBi



Using semantic model, visualised the Star Schema:-

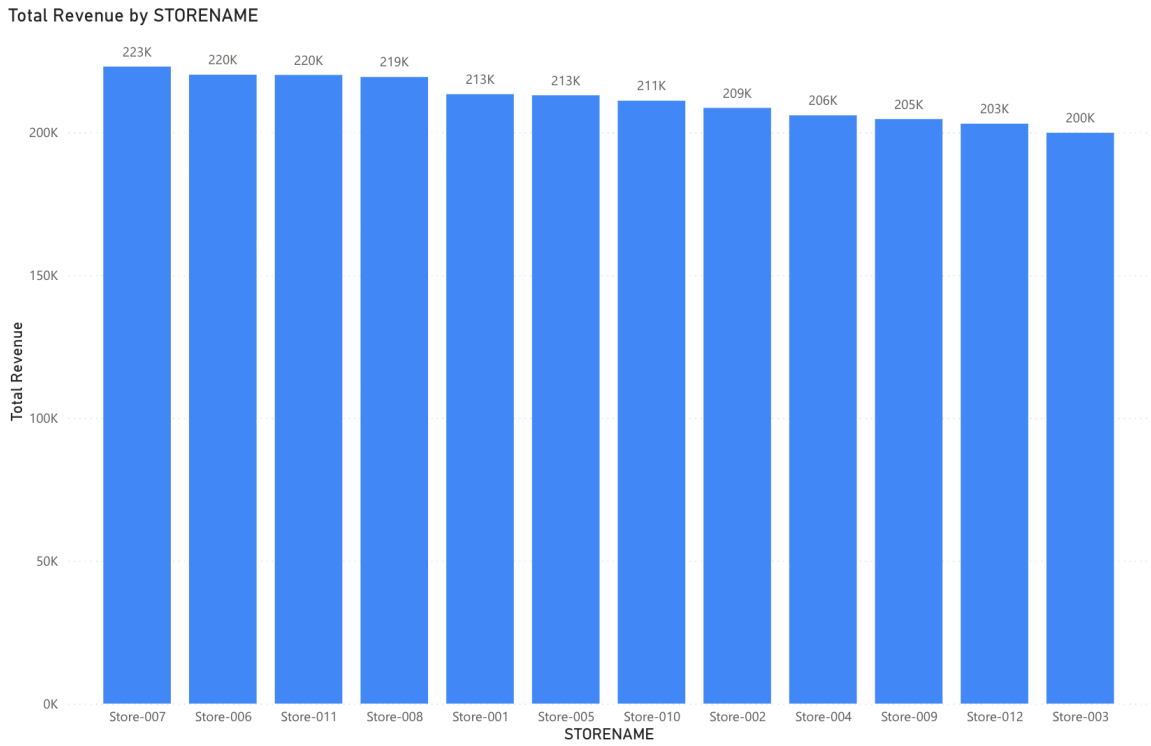


- **Real-Time Operations Dashboard:**
  - An **Inventory Shortages** report. This was built using a DAX Measure (`IsShortage`) to filter the `gold_current_inventory` table to only show rows where `OnHandQty < ReorderPoint`.

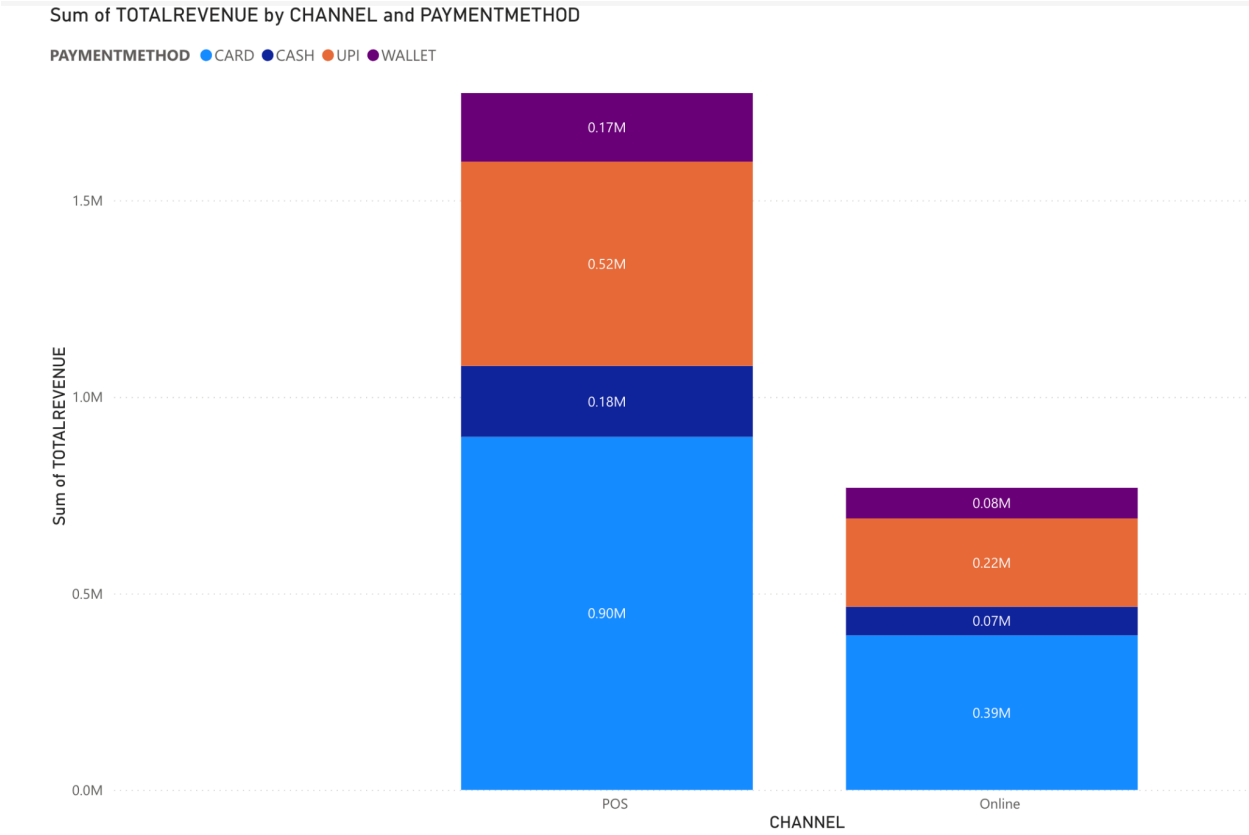
STOREID	PRODUCTID	ONHANDQTY	REORDERPOINT
1.00	46.00	44.00	45.00
1.00	241.00	34.00	41.00
2.00	40.00	9.00	41.00
2.00	46.00	47.00	57.00
2.00	131.00	32.00	41.00
3.00	22.00	43.00	44.00
3.00	61.00	30.00	47.00
3.00	118.00	30.00	47.00
3.00	190.00	29.00	40.00
3.00	245.00	33.00	49.00
4.00	88.00	18.00	38.00
5.00	103.00	42.00	51.00
6.00	136.00	45.00	50.00
6.00	203.00	55.00	56.00
7.00	24.00	40.00	41.00
7.00	161.00	41.00	46.00
8.00	25.00	24.00	33.00
9.00	207.00	22.00	47.00
12.00	52.00	41.00	51.00
12.00	213.00	20.00	46.00

- **Executive Sales Dashboard:** Provided a daily overview of sales, regional performance, and top-selling products.

a. Total Revenue v/s StoreName



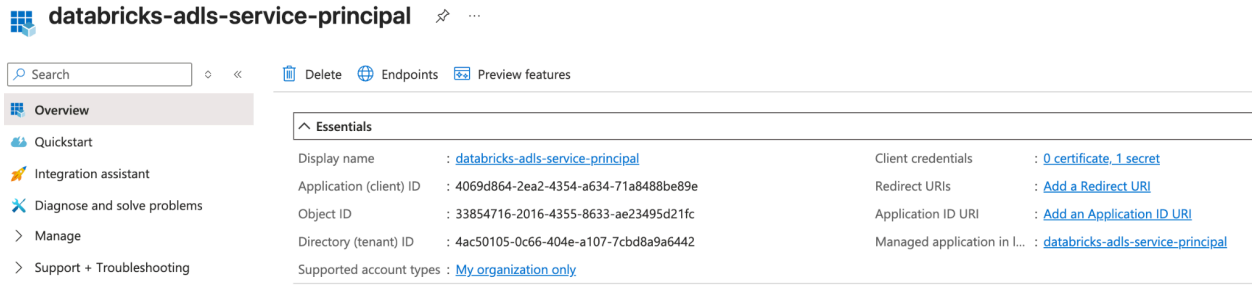
b. Total Revenue v/s Payment Method

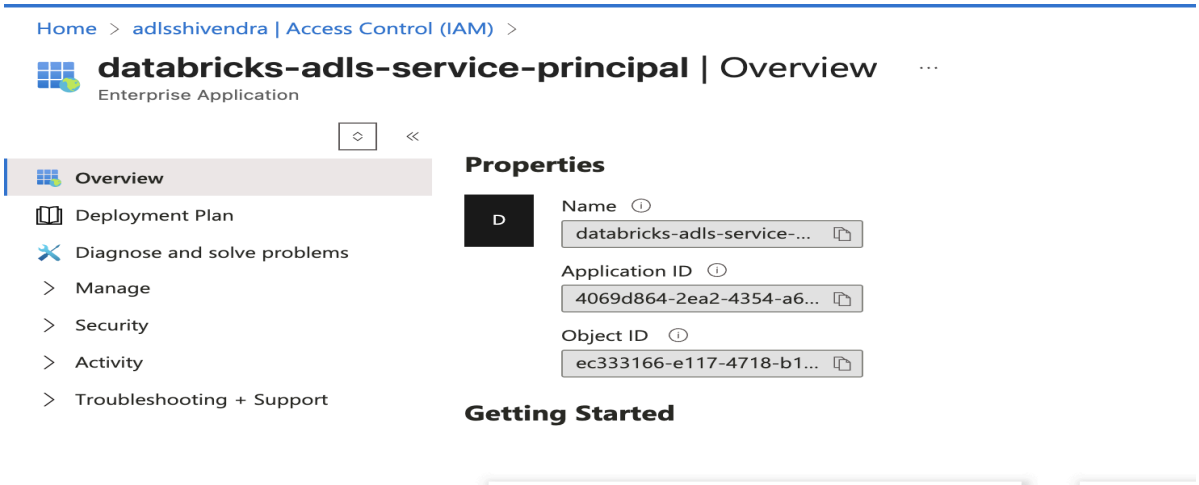


Many more Dashboards can be made using PowerBI.

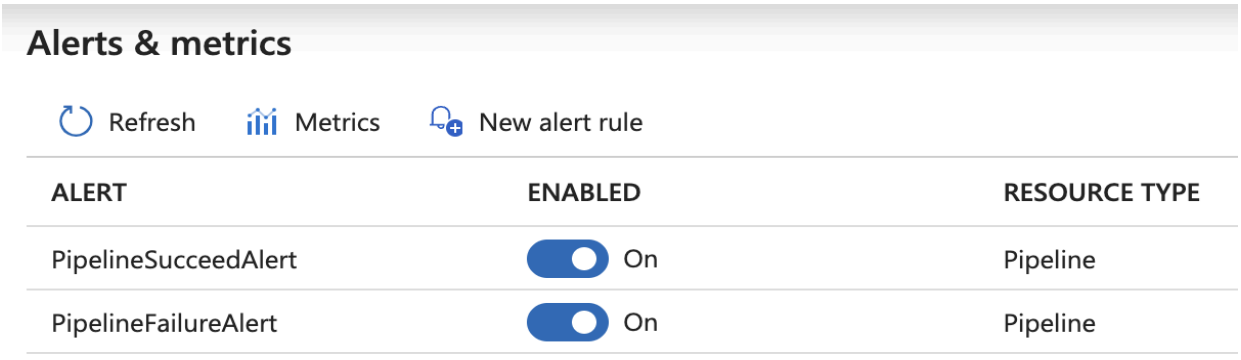
8. Security & Monitoring

- **Security:** All access to ADLS Gen2 from Databricks was secured using an Azure Service Principal with the "Storage Blob Data Contributor" IAM role. PII data was masked in the Silver layer.





- **Monitoring:** An alert rule was configured in Azure Monitor to automatically send an email notification upon any ADF pipeline failure



9. Conclusion & Future Work

This project successfully delivered a complete, end-to-end retail analytics platform. We transformed raw, siloed data from multiple sources into a unified, secure, and real-time source of truth that directly drives business decisions through interactive dashboards.

Future Work:

- Implement a **Demand Forecasting** model using Snowpark ML on the `gold_daily_sales_summary` table.
- Create a detailed **Cost Estimation** playbook for all Azure and Snowflake services.
- Expand monitoring to Databricks logs in Azure Log Analytics.