Project 08

Building a Data Engineering Pipeline for Business Decision Making using Microsoft Fabric

Deepika Mandapalli and Unnathi Shetty

13th January, 2026

Project Overview

- This project builds an end-to-end data engineering pipeline using Microsoft Fabric and the Medallion Architecture.
- The raw e-commerce customer behavior dataset is processed through Bronze, Silver, and Gold layers to create clean, analytics-ready data.
- The final output supports business decision-making through curated fact and dimension tables and interactive Power BI dashboards.

Environment Setup

Component	Name / Details
Platform	Microsoft Fabric
Workspace	Coffee_Shop
OneLake Storage(Lakehouse)	Coffee_shop_LH
Schemas	landing /bronze / silver / gold
Notebooks	NB_01_Bronze_Ingest, NB_02_Silver_DQ_And_Transform NB_03_Gold_Build
Semantic Model	Corner_coffee_shop_semantic_model
File Formats	CSV /Delta
Reporting Tool	Power BI

Dataset

- Dataset name: coffee_shop_sales_2024_01.csv
- Source: Github
- Format: CSV
- Rows: 4139-4200

- Columns: 11

Field Name	Description
transaction_id	Unique identifier for each sales transaction
transaction_qty	Number of units purchased in the transaction
store_id	Unique identifier for the coffee shop store
store_location	Physical location or neighborhood of the store
product_id	Unique identifier for the product sold
unit_price	Price per single unit of the product
product_category	High-level product group (Coffee, Snacks, Drinking Chocolate, etc.)
product_type	Specific product type within the category (Americano, Mocha, Cookie, etc.)
product_detail	Detailed variation of the product (Iced, Classic, Chocolate chip, etc.)
transaction_type	Payment method used (Cash, Credit Card, Debit Card, Apple Pay, etc.)
transaction_datetime	Timestamp when the transaction occurred (date + time)

coffee_shop_sales_2024_01

transaction_id	transaction_qty	store_id	store_location	product_id	unit_price	product_category	product_type	product_detail	transaction_type	transaction_datetime
1	2	1	Astoria	4404	4.47	Drinking Chocolate	Mocha	Classic	Credit Card	2024-01-01 17:23:52
2	3	3	Hell's Kitchen	7250	2.71	Snacks	Granola Bar	Chocolate	Apple Pay	2024-01-01 12:01:59
3	1	1	Astoria	9292	2.98	Coffee	Americano	Iced	Cash	2024-01-01 17:41:27
4	4	3	Hell's Kitchen	4404	4.58	Drinking Chocolate	Mocha	Classic	Debit Card	2024-01-01 15:02:36
5	1	3	Hell's Kitchen	4021	2.53	Snacks	Cookie	Chocolate chip	Credit Card	2024-01-01 16:08:25
6	1	1	Astoria	8101	3.7	Tea	Herbal Tea	Peppermint	Credit Card	2024-01-01 13:43:07
7	1	3	Hell's Kitchen	4404	4.37	Drinking Chocolate	Mocha	Classic	Credit Card	2024-01-01 12:53:41
8	2	2	Brooklyn	2893	6.33	Snacks	Sandwich	Veggie	Cash	2024-01-01 08:09:35
9	1	2	Brooklyn	8830	2.81	Pastries	Croissant	Chocolate	Debit Card	2024-01-01 11:23:25
10	1	1	Astoria	3710	2.78	Pastries	Bagel	Plain	Cash	2024-01-01 19:00:24
11	1	3	Hell's Kitchen	6279	3.28	Coffee	Espresso	Double shot	Credit Card	2024-01-01 17:04:41
12	3	3	Hell's Kitchen	8511	3.2	Tea	Black Tea	Earl Grey	Apple Pay	2024-01-01 14:11:33
13	2	1	Astoria	2893	5.98	Snacks	Sandwich	Veggie	Cash	2024-01-01 11:46:13
14	1	3	Hell's Kitchen	1265	5.05	Coffee	Cold Brew	With milk	Credit Card	2024-01-01 20:38:26
15	1	1	Astoria	9049	3.28	Tea	Green Tea	Mint	Credit Card	2024-01-01 10:31:03
16	1	2	Brooklyn	6851	5.91	Snacks	Sandwich	Chicken	Credit Card	2024-01-01 17:28:35
17	3	1	Astoria	5047	3.56	Drinking Chocolate	Hot Chocolate	Whipped cream	Credit Card	2024-01-01 11:05:27
18	1	2	Brooklyn	7435	5.02	Drinking Chocolate	Mocha	Iced	Debit Card	2024-01-01 15:38:32
19	1	1	Astoria	7899	3.6	Coffee	Cappuccino	Extra foam	Credit Card	2024-01-01 12:07:26
20	1	1	Astoria	4021	2.19	Snacks	Cookie	Chocolate chip	Credit Card	2024-01-01 14:14:58
21	1	2	Brooklyn	3255	3.25	Coffee	Americano	Regular	Debit Card	2024-01-01 13:01:02
22	1	1	Astoria	7505	2.25	Snacks	Cookie	Oatmeal raisin	Credit Card	2024-01-01 07:01:01
23	1	2	Brooklyn	4021	2.49	Snacks	Cookie	Chocolate chip	Credit Card	2024-01-01 12:31:54
24	1	3	Hell's Kitchen	4404	4.4	Drinking Chocolate	Mocha	Classic	Credit Card	2024-01-01 12:23:04

Top 10 Business Objectives

Objective 1 – Analyze sales performance by store location to compare revenue, and transaction volume across stores.

Objective 2 – Identify top-selling products by store location based on revenue and quantity to understand local customer preferences and optimize inventory.

Objective 3 – Determine the product categories that generate the highest revenue and sales volume to support category-level assortment and restocking decisions.

Objective 4 – Assess peak transaction hours by store location to optimize staffing levels and operational efficiency.

Objective 5 – Measure average transaction value (ATV) to evaluate customer spending behavior and upselling effectiveness.

Objective 6 – Compare weekday and weekend sales performance to identify demand patterns and inform promotional and staffing strategies.

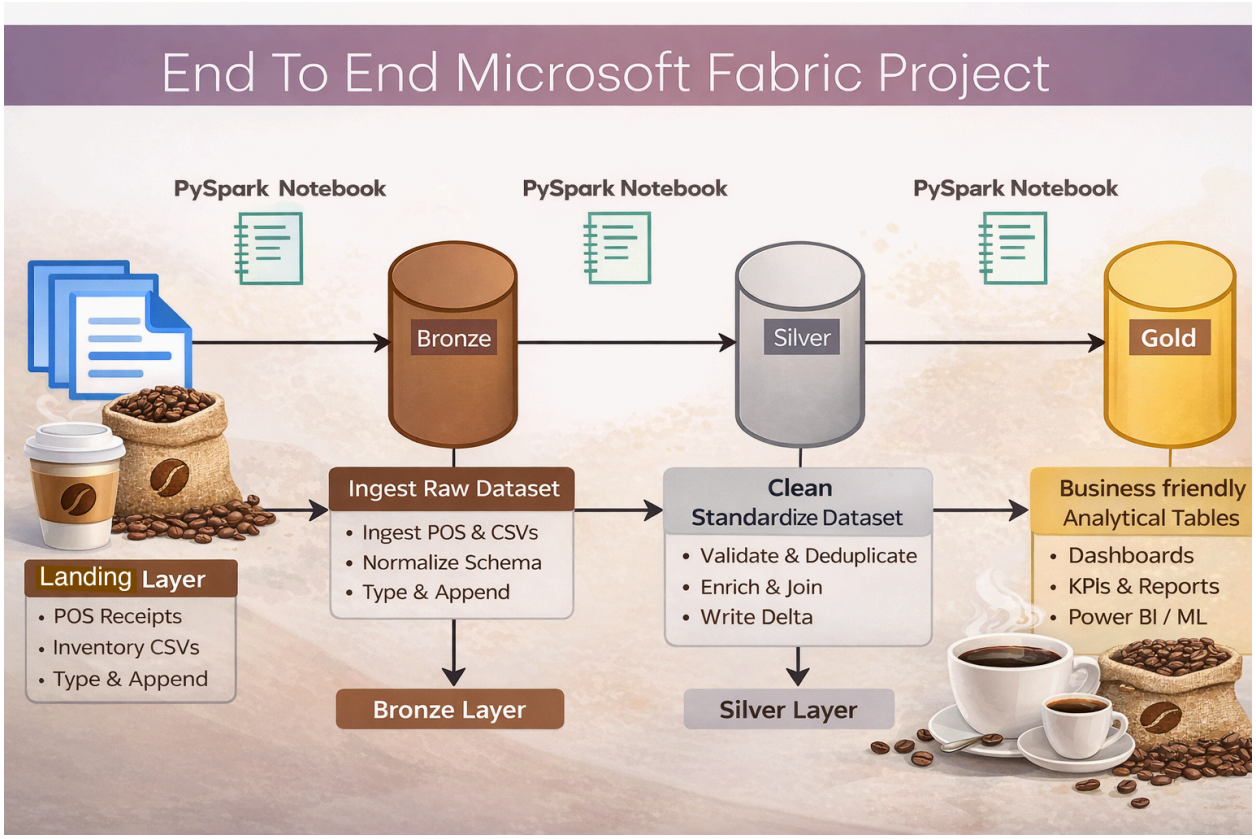
Objective 7 – Identify underperforming products and categories to support pricing adjustments, promotions, or product rationalization.

Objective 8 – Analyze intra-month sales trends to enable timely restocking decisions.

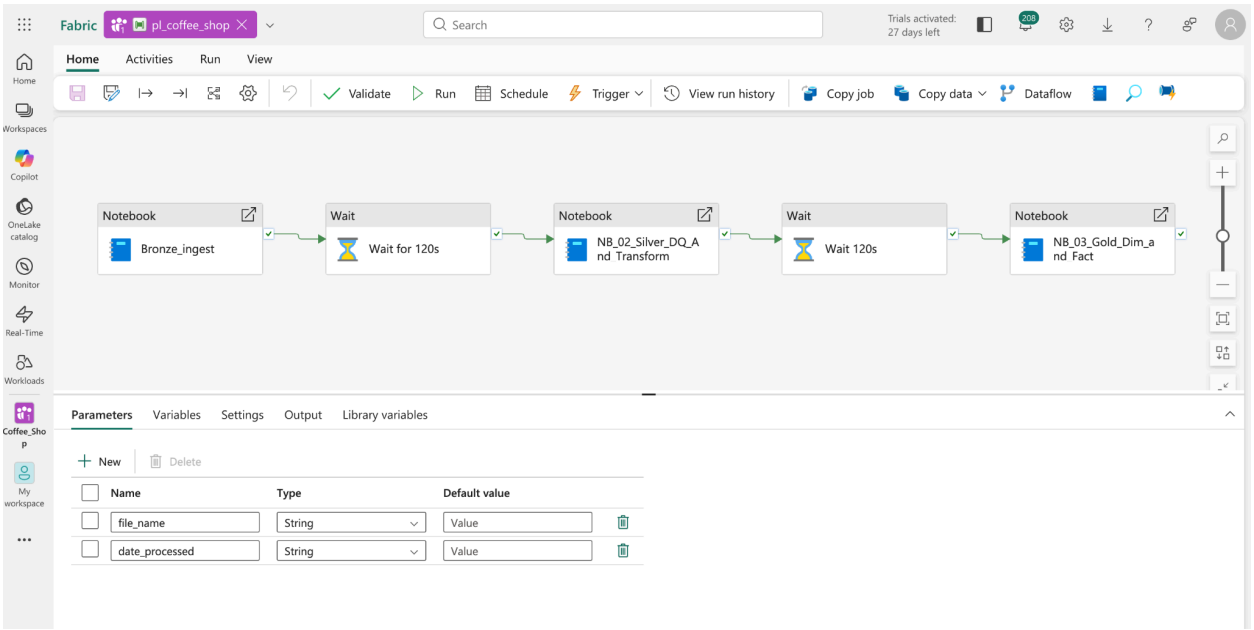
Objective 9 – Evaluate pricing consistency across store locations to detect unit price anomalies that may impact revenue or customer trust.

Objective 10 – Analyze payment method usage to understand customer payment preferences and support payment strategy decisions.

Data Processing Architecture



Pipeline for Business Intelligence Solution

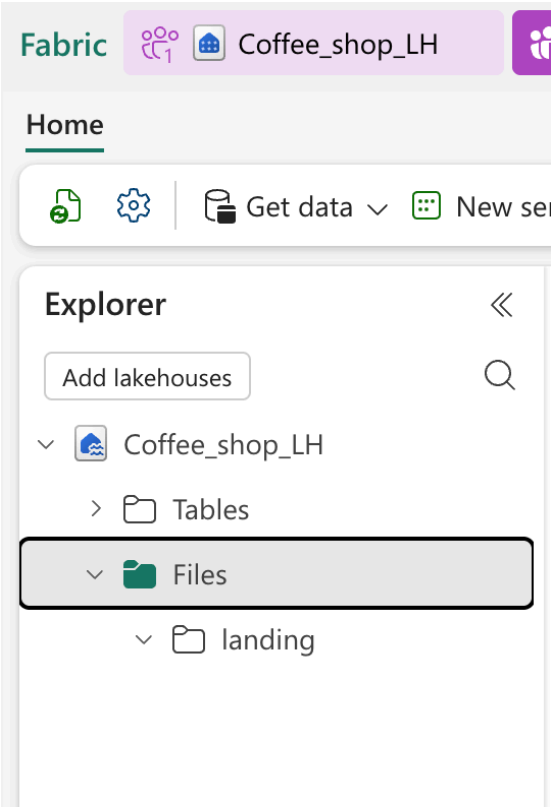


Workspace

- A catalog named Coffee_Shop was created to organize all project-related data objects.

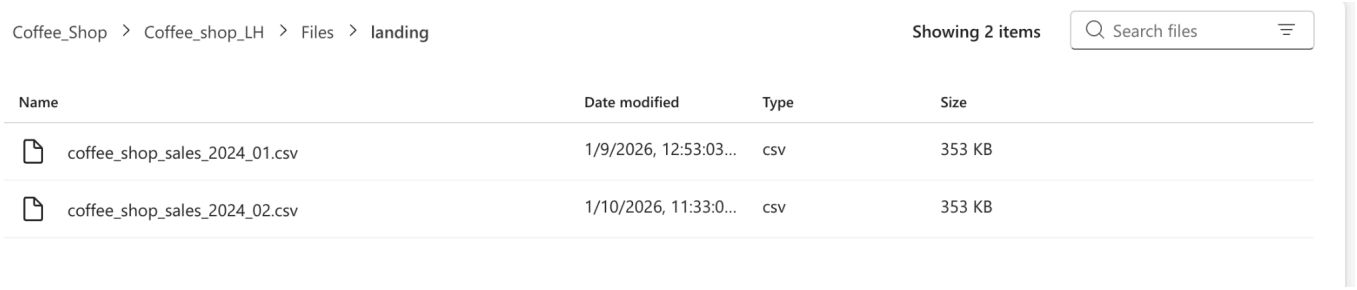
Onelake Catalog - Lakehouse

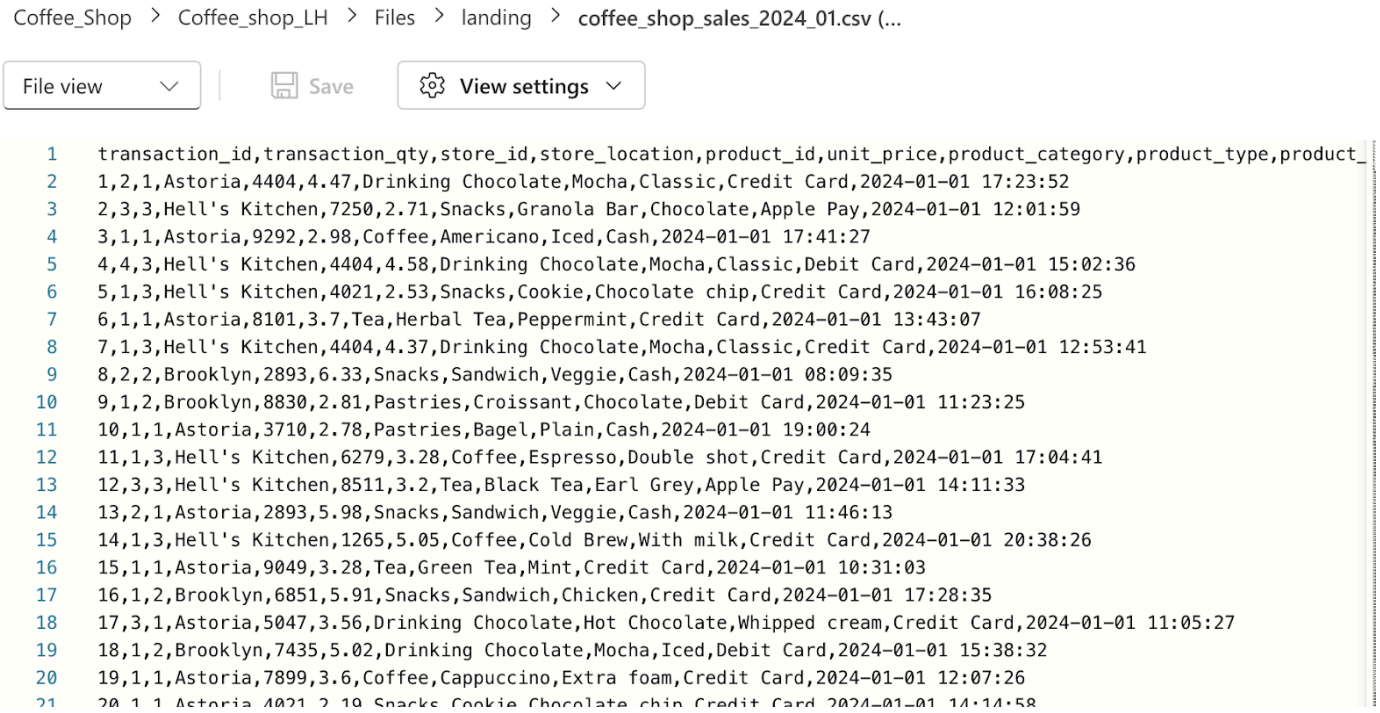
- The catalog provides a logical container for files and tables across different layers.
- The landing folder was created in files.
- Bronze, Silver, and Gold schemas are created under the tables in the lakehouse using notebooks.
- Helps maintain clear separation and governance of data assets within the project.



Landing Layer

- Stores the raw dataset exactly as received from the source.
- Acts as the initial ingestion layer with no transformations applied.
- Data is loaded in its original format (CSV / table).
- Used as a reference point for auditing and reprocessing if needed.
- Ensures raw data is preserved before any cleaning or validation steps.





Bronze Layer – Data Ingestion

Purpose

- Load raw coffee_shop_sales_2024_01.csv data into the Bronze layer in Delta format without transformations.
- Data is ingested from the Landing Layer without business transformations.

Bronze Code

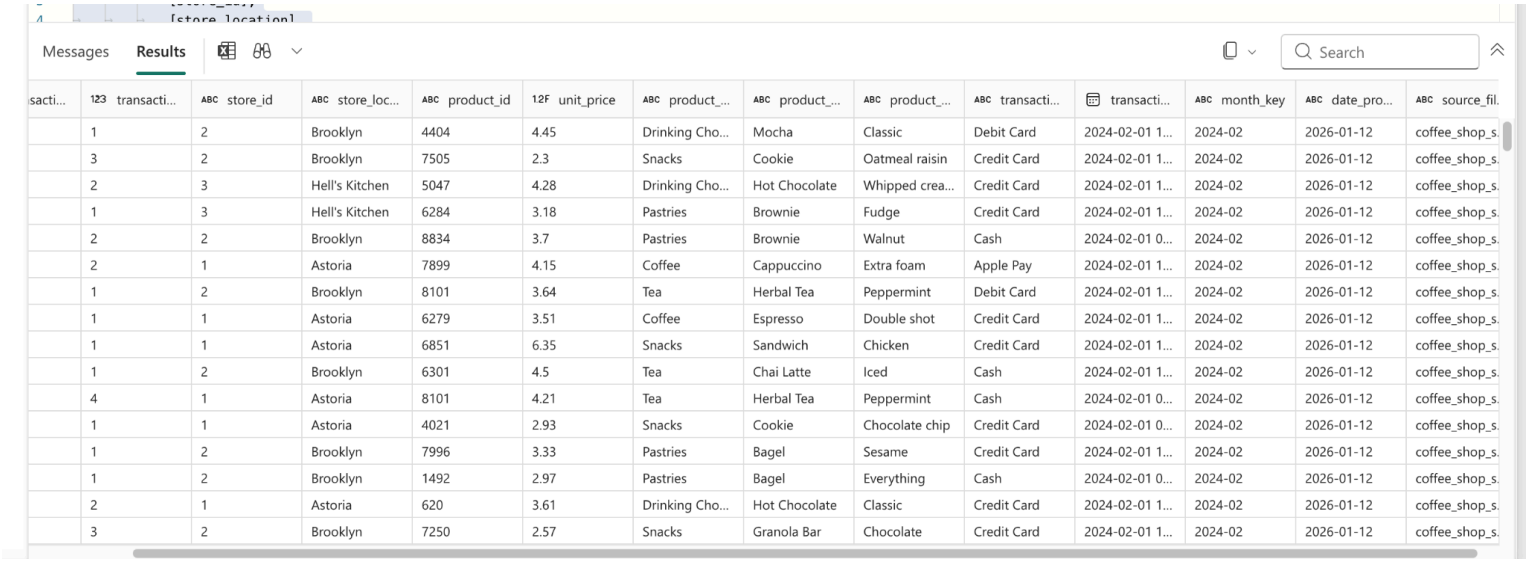
```
# Create Bronze Schema

#spark.sql("DROP TABLE IF EXISTS bronze.coffee_sales")

spark.sql("CREATE SCHEMA IF NOT EXISTS bronze")

print("Reset done: dropped bronze.coffee_sales (if it existed).")

from pyspark.sql import functions as F

import re

# ----------------------------

# Params (pipeline OR manual) - Fabric safe

# ----------------------------

# file_name and date_processed are injected by the pipeline runtime-parameters cell

if not file_name:

  raise ValueError("file_name is missing from pipeline parameters")

if not date_processed:

  raise ValueError("date_processed is missing from pipeline parameters")
```

```
print("file_name:", file_name)

print("date_processed:", date_processed)

# ----------------------------

# Paths

# ----------------------------

landing_base_path =
"abfss://Coffee_Shop@onelake.dfs.fabric.microsoft.com/Coffee_shop_LH.Lakehouse/Files/landi
ng"

landing_file_path = f"{landing_base_path}/{file_name}"

print("landing_file_path:", landing_file_path)

# ----------------------------

# month_key from filename (YYYY_MM)

# ----------------------------

m = re.search(r"(\d{4})_(\d{2})", file_name)

if not m:

  raise ValueError(f"Cannot derive month_key from file_name: {file_name}")

month_key = f"{m.group(1)}-{m.group(2)}"

print("month_key:", month_key)
```

```
# ----------------------------
# Read Landing
# ----------------------------
df_raw = (spark.read.format("csv")
        .option("header", "true")
        .load(landing_file_path))

df_raw = df_raw.drop(*[c for c in df_raw.columns if c.startswith("_c")])
# ----------------------------
# Cast + select
# ----------------------------
df_bronze = df_raw.select(
  F.col("transaction_id").cast("string").alias("transaction_id"),
  F.col("transaction_qty").cast("int").alias("transaction_qty"),
  F.col("store_id").cast("string").alias("store_id"),
  F.col("store_location").cast("string").alias("store_location"),
  F.col("product_id").cast("string").alias("product_id"),
  F.col("unit_price").cast("double").alias("unit_price"),
  F.col("product_category").cast("string").alias("product_category"),
  F.col("product_type").cast("string").alias("product_type"),
```

```
    F.col("product_detail").cast("string").alias("product_detail"),
    F.col("transaction_type").cast("string").alias("transaction_type"),
    F.col("transaction_datetime").cast("timestamp").alias("transaction_datetime"),

    F.lit(month_key).alias("month_key"),
    F.lit(date_processed).alias("date_processed"),
    F.lit(file_name).alias("source_file_name")
)
# ----------------------------
# Write: month partition overwrite
# ----------------------------
target_table = "bronze.coffee_sales"
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")
(df_bronze.write
 .format("delta")
 .mode("overwrite")
 .partitionBy("month_key")
 .saveAsTable(target_table))

print("LANDING → BRONZE done:", target_table, "| month_key:", month_key)
```

```
landing_file_path:
abfss://Coffee_Shop@onelake.dfs.fabric.microsoft.com/Coffee_shop_LH.Lakehouse/Files/landing/coffee_shop_sales_
2024_01.csv
date_processed: 2026-01-09
LANDING → BRONZE done: bronze.coffee_sales
```



Messages | Results | Search

sacti...	123 transacti...	ABC store_id	ABC store_loc...	ABC product_id	12F unit_price	ABC product_...	ABC product_...	ABC product_...	ABC transacti...	transacti...	ABC month_key	ABC date_pro...	ABC source_fil.
	1	2	Brooklyn	4404	4.45	Drinking Cho...	Mocha	Classic	Debit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	3	2	Brooklyn	7505	2.3	Snacks	Cookie	Oatmeal raisin	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	2	3	Hell's Kitchen	5047	4.28	Drinking Cho...	Hot Chocolate	Whipped crea...	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	3	Hell's Kitchen	6284	3.18	Pastries	Brownie	Fudge	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	2	2	Brooklyn	8834	3.7	Pastries	Brownie	Walnut	Cash	2024-02-01 0...	2024-02	2026-01-12	coffee_shop_s.
	2	1	Astoria	7899	4.15	Coffee	Cappuccino	Extra foam	Apple Pay	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	2	Brooklyn	8101	3.64	Tea	Herbal Tea	Peppermint	Debit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	1	Astoria	6279	3.51	Coffee	Espresso	Double shot	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	1	Astoria	6851	6.35	Snacks	Sandwich	Chicken	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	2	Brooklyn	6301	4.5	Tea	Chai Latte	Iced	Cash	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	4	1	Astoria	8101	4.21	Tea	Herbal Tea	Peppermint	Cash	2024-02-01 0...	2024-02	2026-01-12	coffee_shop_s.
	1	1	Astoria	4021	2.93	Snacks	Cookie	Chocolate chip	Credit Card	2024-02-01 0...	2024-02	2026-01-12	coffee_shop_s.
	1	2	Brooklyn	7996	3.33	Pastries	Bagel	Sesame	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	1	2	Brooklyn	1492	2.97	Pastries	Bagel	Everything	Cash	2024-02-01 0...	2024-02	2026-01-12	coffee_shop_s.
	2	1	Astoria	620	3.61	Drinking Cho...	Hot Chocolate	Classic	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.
	3	2	Brooklyn	7250	2.57	Snacks	Granola Bar	Chocolate	Credit Card	2024-02-01 1...	2024-02	2026-01-12	coffee_shop_s.

Month_key → Enables **month-based partitioning** for efficient incremental loads
Allows processing and overwriting **only the affected month**, not the full table

Date_processed → Tracks **when the data was ingested** into the system
Helps identify **latest vs older pipeline runs**

Source_file_name → Maintains **data lineage** back to the original file
Helps debug issues when data is incorrect or duplicated, Enables validation of which **input file produced which records**

Silver Layer – Data Cleaning & Transformation

Purpose

- Convert raw data into clean, standardized, analysis-ready data

Data Quality Checks

- **Removed fully duplicated rows**

Exact duplicate records within the batch were removed using a composite key (transaction_id, product_id, transaction_datetime, store_id) to prevent duplicate sales records.

- **Dropped rows with invalid or missing critical values**

  Records missing essential analytical fields were removed to ensure data reliability:

  - transaction_id
  - store_id
  - product_id
  - transaction_datetime
- **Filtered out invalid dates**

Not applicable — date values are derived from a valid transaction_datetime field.

- **Removed records with invalid numeric values**

Rows with invalid numeric measures were filtered out:

- transaction_qty >= 0
- unit_price >= 0

Transformations Applied

Null Handling – Critical Columns

Rows were dropped if any of the following columns contained null values:

- transaction_id
- store_id
- product_id
- transaction_datetime

Data Quality Rules

- transaction_qty must be positive
- unit_price must be positive
- Each store_id must map to a single store_location
- Each product_id must map consistently to:
    - product_category
    - product_type
    - product_detail

Text Standardization

- Trimmed whitespace from all string columns
- Standardized categorical values to uppercase:
    - store_location
    - product_category
    - product_type
    - transaction_type

Business Transformations

- Calculated total_amount as:
    - transaction_qty × unit_price

- Derived time attributes from transaction_datetime:
    - transaction_date
    - transaction_time
    - transaction_hour
    - day_name
    - month_name
    - month_number

- Created time_bucket for time-based analysis:

    - Morning (5–11)
    - Afternoon (12–16)
    - Evening (17–21)

Silver Code

```
# ============================================================
# Project 8 | BRONZE -> SILVER
# PART 1: INITIAL DQ CHECKS (BRONZE for the batch)
# ============================================================
from pyspark.sql import functions as F
import re
# ----------------------------
# Params (from pipeline)
# ----------------------------
if not file_name:
  raise ValueError("Missing parameter: file_name. Check pipeline Notebook activity Base parameters key must be 'file_name'.")
```

```
m = re.search(r"(\d{4})_(\d{2})", file_name)
if not m:
  raise ValueError(f"Cannot derive month_key from file_name: {file_name}")
month_key = f"{m.group(1)}-{m.group(2)}"

print("Processing month_key:", month_key)

source_table = "bronze.coffee_sales"

df0 = (spark.table(source_table)
    .filter(F.col("month_key") == F.lit(month_key)))

if df0.rdd.isEmpty():
  raise ValueError(f"No Bronze data found for month_key={month_key}")

print("=== INITIAL DATA QUALITY CHECKS (BRONZE for this month) ===")

# 1) Total rows (month)
total_rows = df0.count()
print("Total rows (this month):", total_rows)

# 2) Duplicate transaction_id
dup_txn_df = df0.groupBy("transaction_id").count().filter(F.col("count") > 1)
dup_txn_cnt = dup_txn_df.count()
print("Duplicate transaction_id (number of IDs repeating):", dup_txn_cnt)

print("Sample Duplicate transaction_id (top 20):")
dup_txn_df.orderBy(F.col("count").desc()).show(20, truncate=False)

# 3) Full-row duplicates
```

```
full_dup_rows = total_rows - df0.dropDuplicates().count()
print("Full-row duplicate rows:", full_dup_rows)

# 4) Null checks (critical columns)
critical_cols = ["transaction_id", "store_id", "product_id", "transaction_datetime"]

null_summary = []
for c in critical_cols:
  null_count = df0.filter(F.col(c).isNull()).count()
  null_summary.append((c, null_count))

print("Null counts in critical columns:")
spark.createDataFrame(null_summary, ["Column", "Null_Count"]).show(truncate=False)

# 5) Invalid numeric checks
invalid_qty = df0.filter((F.col("transaction_qty").isNull()) | (F.col("transaction_qty") <=
0)).count()
invalid_price = df0.filter((F.col("unit_price").isNull()) | (F.col("unit_price") <= 0)).count()

print("Invalid transaction_qty rows (null or <= 0):", invalid_qty)
print("Invalid unit_price rows (null or <= 0):", invalid_price)

# 6) store_id -> multiple locations
store_loc_df = (df0.groupBy("store_id")
          .agg(F.countDistinct("store_location").alias("location_count"))
          .filter(F.col("location_count") > 1)
          .orderBy(F.col("location_count").desc()))

store_loc_cnt = store_loc_df.count()
if store_loc_cnt > 0:
```

```
  print(f"DQ ALERT: Found {store_loc_cnt} store_id values mapped to >1 store_location (month={month_key}).")
  store_loc_df.show(truncate=False)
else:
  print("DQ OK: Each store_id maps to a single store_location in this month.")


# 7) product_id -> multiple attributes
prod_attr_df = (df0.groupBy("product_id")
            .agg(
               F.countDistinct("product_category").alias("category_cnt"),
               F.countDistinct("product_type").alias("type_cnt"),
               F.countDistinct("product_detail").alias("detail_cnt")
            )
            .filter((F.col("category_cnt") > 1) | (F.col("type_cnt") > 1) | (F.col("detail_cnt") > 1))
            .orderBy(F.col("category_cnt").desc(), F.col("type_cnt").desc(), F.col("detail_cnt").desc()))

prod_attr_cnt = prod_attr_df.count()
if prod_attr_cnt > 0:
  print(f"DQ ALERT: Found {prod_attr_cnt} product_id values mapped to inconsistent attributes (month={month_key}).")
  prod_attr_df.show(truncate=False)
else:
  print("DQ OK: Each product_id maps consistently to category/type/detail in this month.")
```

```
Processing date_processed: 2026-01-09
=== INITIAL DATA QUALITY CHECKS (BRONZE) ===
Total rows (this batch): 4139
Duplicate transaction_id (number of IDs repeating): 0
Sample Duplicate transaction_id (top 20):
+--------------+-----+
|transaction_id|count|
+--------------+-----+
+--------------+-----+

Full-row duplicate rows: 0
Null counts in critical columns:
+--------------------+----------+
|Column              |Null_Count|
+--------------------+----------+
|transaction_id      |0         |
|store_id            |0         |
|product_id          |0         |
|transaction_datetime|0         |
+--------------------+----------+

Invalid transaction_qty rows (null or <= 0): 0
Invalid unit_price rows (null or <= 0): 0
DQ OK: Each store_id maps to a single store_location in this batch.
DQ OK: Each product_id maps consistently to category/type/detail in this batch.
```

Transformation and Cleaning Code

```
# ============================================================
# Project 8 | BRONZE -> SILVER (Coffee Shop Sales)
# PART 2: SILVER CLEANING + TRANSFORMATION
# ============================================================

from pyspark.sql import functions as F
import re

# ----------------------------
# Params (pipeline)
# ----------------------------

m = re.search(r"(\d{4})_(\d{2})", file_name)
if not m:
  raise ValueError(f"Cannot derive month_key from file_name: {file_name}")
month_key = f"{m.group(1)}-{m.group(2)}"

print("file_name:", file_name)
print("month_key:", month_key)
```

```
print("date_processed:", date_processed)

spark.sql("CREATE SCHEMA IF NOT EXISTS silver")

source_table = "bronze.coffee_sales"
target_table = "silver.coffee_sales"

# ----------------------------
# Read Bronze (only this month)
# ----------------------------
df_bronze = spark.table(source_table).filter(F.col("month_key") == F.lit(month_key))
if df_bronze.rdd.isEmpty():
  raise ValueError(f"No Bronze data found for month_key={month_key}")

# ----------------------------
# Silver cleaning
# ----------------------------
df_clean = (
  df_bronze
  .dropna(subset=["transaction_id", "store_id", "product_id", "transaction_datetime"])
  .filter((F.col("transaction_qty") > 0) & (F.col("unit_price") > 0))
  .withColumn("store_location", F.upper(F.trim(F.col("store_location"))))
  .withColumn("product_category", F.upper(F.trim(F.col("product_category"))))
  .withColumn("product_type", F.upper(F.trim(F.col("product_type"))))
  .withColumn("product_detail", F.trim(F.col("product_detail")))
  .withColumn("transaction_type", F.upper(F.trim(F.col("transaction_type"))))
  .dropDuplicates(["transaction_id", "product_id", "transaction_datetime", "store_id"])
)

# ----------------------------
```

```
# Transformations
# ----------------------------
df_silver = (
  df_clean
  .withColumn("total_amount", F.round(F.col("transaction_qty") * F.col("unit_price"), 2))
  .withColumn("transaction_date", F.to_date(F.col("transaction_datetime")))
  .withColumn("transaction_time", F.date_format(F.col("transaction_datetime"), "HH:mm:ss"))
  .withColumn("transaction_hour", F.hour(F.col("transaction_datetime")))
  .withColumn("day_name", F.date_format(F.col("transaction_datetime"), "EEEE"))
  .withColumn("month_name", F.date_format(F.col("transaction_datetime"), "MMMM"))
  .withColumn("month_number", F.month(F.col("transaction_datetime")))
  .withColumn(
     "time_bucket",
     F.when(F.col("transaction_hour").between(5, 11), "Morning")
      .when(F.col("transaction_hour").between(12, 16), "Afternoon")
      .when(F.col("transaction_hour").between(17, 21), "Evening")
      .otherwise("Night")
  )
  # keep current processing date for lineage
  .withColumn("date_processed", F.lit(date_processed))
)

print("Final rows (this month):", df_silver.count())

# ----------------------------
# WRITE SILVER (rerun-safe overwrite ONLY this month_key)
# ----------------------------
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")

# If table doesn't exist yet, create it once (overwrite full table)
```

```
if not spark.catalog.tableExists(target_table):
  (df_silver.write
   .format("delta")
   .mode("overwrite")
   .partitionBy("month_key")
   .saveAsTable(target_table))
  print("Created silver table:", target_table, "| partitionBy month_key")
else:
  # Overwrite only this month partition
  (df_silver.write
   .format("delta")
   .mode("overwrite")
   .option("replaceWhere", f"month_key = '{month_key}'")
   .saveAsTable(target_table))
  print("Overwrote month partition in silver:", month_key)

# ----------------------------
# Quick validation
# ----------------------------
spark.sql(f"""
SELECT month_key, COUNT(*) AS cnt
FROM {target_table}
GROUP BY month_key
ORDER BY month_key
""").show()
```

```
file_name: coffee_shop_sales_2024_01.csv
month_key: 2024-01
date_processed: 2026-01-11
Final rows (this month): 4139
Created silver table: silver.coffee_sales | partitionBy month_key
+---------+----+
|month_key| cnt|
+---------+----+
|  2024-01|4139|
+---------+----+
```

Final Data Quality Check (After Cleaning)

Validations performed after Silver processing:

- Confirmed unique store_id → store_location mapping
- Confirmed consistent product_id → category/type/detail mapping
- Verified only valid, business-ready records are written to Silver

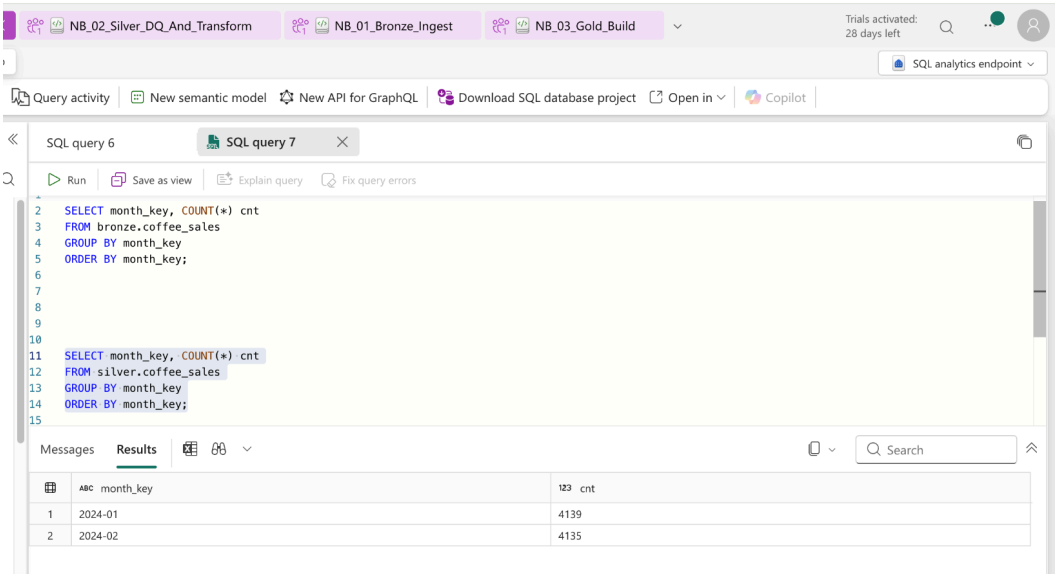
```
print("=== FINAL DATA QUALITY CHECKS (SILVER) ===")

# 1) Row count
final_rows = df_silver.count()
print("Final rows (this month):", final_rows)

# 2) Critical nulls (should be 0)
critical_cols = ["transaction_id", "store_id", "product_id", "transaction_datetime", "month_key"]
for c in critical_cols:
  null_cnt = df_silver.filter(F.col(c).isNull()).count()
  print(f"Null count in {c}:", null_cnt)

# 3) Invalid measures (should be 0)
invalid_qty = df_silver.filter(F.col("transaction_qty") <= 0).count()
```

```
invalid_price = df_silver.filter(F.col("unit_price") <= 0).count()
invalid_total = df_silver.filter(F.col("total_amount") <= 0).count()

print("Invalid transaction_qty (<=0):", invalid_qty)
print("Invalid unit_price (<=0):", invalid_price)
print("Invalid total_amount (<=0):", invalid_total)

# 4) Duplicate check on business grain (should be 0)
dup_cnt = (df_silver
        .groupBy("transaction_id", "product_id", "transaction_datetime", "store_id")
        .count()
        .filter(F.col("count") > 1)
        .count())
print("Duplicate rows on grain (txn_id, product_id, datetime, store_id):", dup_cnt)

# 5) Consistency checks (lightweight)
store_loc_after = (df_silver.groupBy("store_id")
                .agg(F.countDistinct("store_location").alias("location_count"))
                .filter(F.col("location_count") > 1)
                .count())
print("store_id mapped to >1 store_location:", store_loc_after)

prod_attr_after = (df_silver.groupBy("product_id")
                .agg(
                   F.countDistinct("product_category").alias("category_cnt"),
                   F.countDistinct("product_type").alias("type_cnt"),
                   F.countDistinct("product_detail").alias("detail_cnt")
                )
                .filter((F.col("category_cnt") > 1) | (F.col("type_cnt") > 1) | (F.col("detail_cnt") > 1))
                .count())
```

```
print("product_id mapped to inconsistent attributes:", prod_attr_after)

if dup_cnt > 0 or invalid_qty > 0 or invalid_price > 0 or invalid_total > 0:
    raise ValueError("SILVER DQ FAILED for this month. Check duplicates/invalid measures.")
```

```
=== FINAL DATA QUALITY CHECKS (SILVER) ===
Final rows (this month): 4139
Null count in transaction_id: 0
Null count in store_id: 0
Null count in product_id: 0
Null count in transaction_datetime: 0
Null count in month_key: 0
Invalid transaction_qty (<=0): 0
Invalid unit_price (<=0): 0
Invalid total_amount (<=0): 0
Duplicate rows on grain (txn_id, product_id, datetime, store_id): 0
store_id mapped to >1 store_location: 0
product_id mapped to inconsistent attributes: 0
```

	transaction_id	transaction_...	store_id	store_location	product_id	unit_price	product_cat...	product_type	product_det...	transaction_...	transaction
1	1	2	1	ASTORIA	4404	4.47	DRINKING CHO...	MOCHA	Classic	CREDIT CARD	2024-01-01T17:
2	10	1	1	ASTORIA	3710	2.78	PASTRIES	BAGEL	Plain	CASH	2024-01-01T19:
3	100	3	3	HELL'S KITCHEN	7776	2.66	SNACKS	GRANOLA BAR	Honey oat	CREDIT CARD	2024-01-01T11:
4	1000	1	1	ASTORIA	6284	3.89	PASTRIES	BROWNIE	Fudge	CREDIT CARD	2024-01-08T17:
5	1001	2	3	HELL'S KITCHEN	6537	4.99	COFFEE	LATTE	Regular	CREDIT CARD	2024-01-08T14:
6	1002	1	2	BROOKLYN	2893	6.08	SNACKS	SANDWICH	Veggie	DEBIT CARD	2024-01-08T12:
7	1003	1	2	BROOKLYN	1492	3.22	PASTRIES	BAGEL	Everything	CASH	2024-01-08T16:
8	1004	1	3	HELL'S KITCHEN	6279	3.46	COFFEE	ESPRESSO	Double shot	CASH	2024-01-08T10:
9	1005	1	1	ASTORIA	4661	2.98	TEA	BLACK TEA	English Breakfast	CASH	2024-01-08T17:
10	1006	1	3	HELL'S KITCHEN	3710	2.75	PASTRIES	BAGEL	Plain	DEBIT CARD	2024-01-08T09:
11	1007	2	3	HELL'S KITCHEN	9292	3.03	COFFEE	AMERICANO	Iced	DEBIT CARD	2024-01-08T19:
12	1008	1	3	HELL'S KITCHEN	7899	4.02	COFFEE	CAPPUCCINO	Extra foam	CASH	2024-01-08T17:



Gold Layer – Business Modeling

Purpose

- Build an analytics-ready star schema for Power BI.
- **Star schema** design to support efficient querying

Dimension Tables

Dimension Table	Primary Key	Description	Key Attributes
dim_store	store_id	Store dimension for location-based analysis	store_location
dim_product	product_id	Product dimension for menu/category analysis	product_category, product_type, product_detail
dim_date	transaction_date	Date dimension for time-series reporting	day_name, month_name, month_number, week_of_year
dim_time	transaction_hour	Time-of-day dimension for hourly trends	hour_label, time_bucket
dim_payment_method	payment_method	Payment method dimension for payment behavior analysis	payment_method

Fact Table

Fact Table	Description	Measures
fact_sales	Stores transaction-level sales records linked to store, product, date, time, and payment method	transaction_qty, unit_price, total_amount (plus identifiers: transaction_id, transaction_datetime)

Gold Code

```
# ============================================================

# Project 8 | SILVER -> GOLD (Star Schema) - MONTH BASED (Rerun-safe)

# ============================================================

from pyspark.sql import functions as F

# ----------------------------

# Setup

# ----------------------------

spark.sql("CREATE SCHEMA IF NOT EXISTS gold")


# Read full Silver (for dimensions)

df = spark.table("silver.coffee_sales")

if df.rdd.isEmpty():

  raise ValueError("silver.coffee_sales is empty. Cannot build Gold.")


# Ensure payment_method column exists

if "payment_method" not in df.columns and "transaction_type" in df.columns:

  df = df.withColumnRenamed("transaction_type", "payment_method")


# Standardize payment_method

df = df.withColumn("payment_method", F.upper(F.trim(F.col("payment_method"))))
```

```
# ----------------------------
# Decide month_key to process (latest month in Silver)
# ----------------------------
if "month_key" not in df.columns:
    raise ValueError("month_key not found in silver.coffee_sales. Gold expects month_key for incremental loads.")

month_key = spark.sql("SELECT MAX(month_key) AS mk FROM silver.coffee_sales").collect()[0]["mk"]
if month_key is None:
    raise ValueError("No month_key found in Silver.")
print("Building GOLD for month_key:", month_key)

df_batch = df.filter(F.col("month_key") == F.lit(month_key))
if df_batch.rdd.isEmpty():
    raise ValueError(f"No Silver data found for month_key={month_key}")
```

```
# ============================================================
# DIMENSIONS (Natural Keys)  (overwrite is fine; small tables)
# ============================================================
# 1) dim_store (PK: store_id)
dim_store = (
  df.select("store_id", "store_location")
    .dropDuplicates(["store_id"])
)
```

	store_id	store_location
1	1	ASTORIA
2	2	BROOKLYN
3	3	HELL'S KITCHEN

```
# 2) dim_product (PK: product_id)
dim_product = (
  df.select("product_id", "product_category", "product_type", "product_detail")
    .dropDuplicates(["product_id"])
)
```

	product_id	product_category	product_type	product_detail
1	1265	COFFEE	COLD BREW	With milk
2	1395	PASTRIES	MUFFIN	Blueberry
3	1492	PASTRIES	BAGEL	Everything
4	1807	COFFEE	CAPPUCCINO	Regular
5	1891	TEA	HERBAL TEA	Chamomile
6	2481	PASTRIES	MUFFIN	Chocolate chip
7	2893	SNACKS	SANDWICH	Veggie
8	3255	COFFEE	AMERICANO	Regular

```
# 3) dim_date (PK: transaction_date)
dim_date = (
  df.select(
      "transaction_date",
      "day_name",
      "month_name",
      "month_number",
      F.weekofyear("transaction_date").alias("week_of_year")
  )
  .dropDuplicates(["transaction_date"])
)
```

	transaction_date	day_name	month_name	month_number	week_of_year
1	2024-01-01	Monday	January	1	1
2	2024-01-02	Tuesday	January	1	1
3	2024-01-03	Wednesday	January	1	1
4	2024-01-04	Thursday	January	1	1
5	2024-01-05	Friday	January	1	1
6	2024-01-06	Saturday	January	1	1
7	2024-01-07	Sunday	January	1	1
8	2024-01-08	Monday	January	1	2
9	2024-01-09	Tuesday	January	1	2

```
# 4) dim_time (PK: transaction_hour)
dim_time = (
  df.select("transaction_hour", "time_bucket")
    .dropDuplicates(["transaction_hour"])
    .withColumn("hour_label", F.format_string("%02d:00", F.col("transaction_hour")))
    .select("transaction_hour", "hour_label", "time_bucket")
```

)

	transaction_hour	hour_label	time_bucket
1	7	07:00	Morning
2	8	08:00	Morning
3	9	09:00	Morning
4	10	10:00	Morning
5	11	11:00	Morning
6	12	12:00	Afternoon
7	13	13:00	Afternoon
8	14	14:00	Afternoon
9	15	15:00	Afternoon
10	16	16:00	Afternoon
11	17	17:00	Evening
12	18	18:00	Evening
13	19	19:00	Evening
14	20	20:00	Evening

5) dim_payment_method (PK: payment_method)

```
dim_payment_method = (
  df.select("payment_method")
    .dropna(subset=["payment_method"])
    .dropDuplicates(["payment_method"])
)
```

	payment_method
1	APPLE PAY
2	CASH
3	DEBIT CARD
4	CREDIT CARD

```
# Write DIM tables
(dim_store.write.format("delta").mode("overwrite").saveAsTable("gold.dim_store"))
(dim_product.write.format("delta").mode("overwrite").saveAsTable("gold.dim_product"))
(dim_date.write.format("delta").mode("overwrite").saveAsTable("gold.dim_date"))
(dim_time.write.format("delta").mode("overwrite").saveAsTable("gold.dim_time"))
(dim_payment_method.write.format("delta").mode("overwrite").saveAsTable("gold.dim_payment_method"))
print("Gold dimensions written.")


# ============================================================
# FACT TABLE (Natural FKs) - overwrite ONLY this month_key
# ============================================================
# Build dynamic column list (so it won't break if metadata changes)
fact_cols = [
  "month_key",
  "transaction_id",
  "transaction_datetime",
  "transaction_date",
  "transaction_hour",
  "store_id",
  "product_id",
  "payment_method",
```

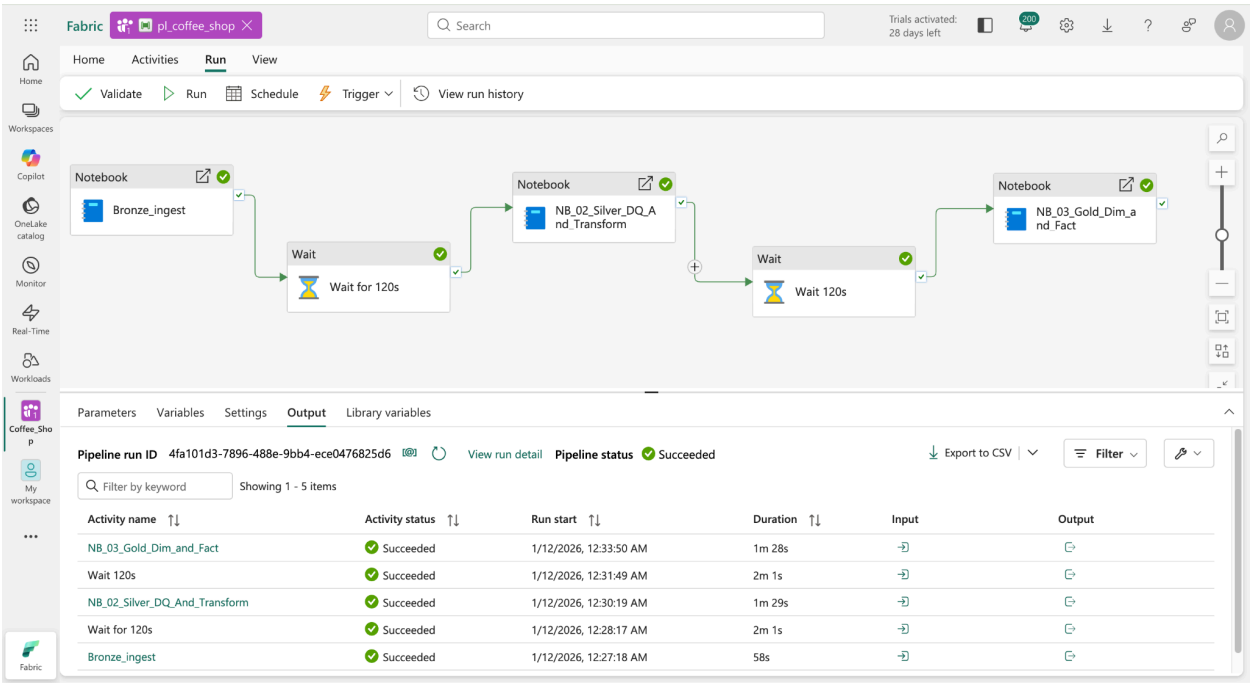
```
    "transaction_qty",
    F.round(F.col("unit_price"), 2).alias("unit_price"),
    F.round(F.col("total_amount"), 2).alias("total_amount"),
]

# Add metadata only if present in Silver
if "date_processed" in df_batch.columns:
    fact_cols.append("date_processed")
if "source_file_name" in df_batch.columns:
    fact_cols.append("source_file_name")

fact_sales = df_batch.select(*fact_cols)

# Duplicate safety (same grain as Silver)
fact_sales = fact_sales.dropDuplicates(
    ["transaction_id", "product_id", "store_id", "transaction_datetime"]
)
```

```
# Write FACT (rerun-safe by month)
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")

if not spark.catalog.tableExists("gold.fact_sales"):
  (fact_sales.write
      .format("delta")
      .mode("overwrite")
      .partitionBy("month_key")
      .saveAsTable("gold.fact_sales"))
  print("Created gold.fact_sales")
else:
  (fact_sales.write
      .format("delta")
      .mode("overwrite")
      .option("replaceWhere", f"month_key = '{month_key}'")
      .saveAsTable("gold.fact_sales"))
  print("Overwrote gold.fact_sales for month_key:", month_key)

print("Gold fact written: gold.fact_sales")
```

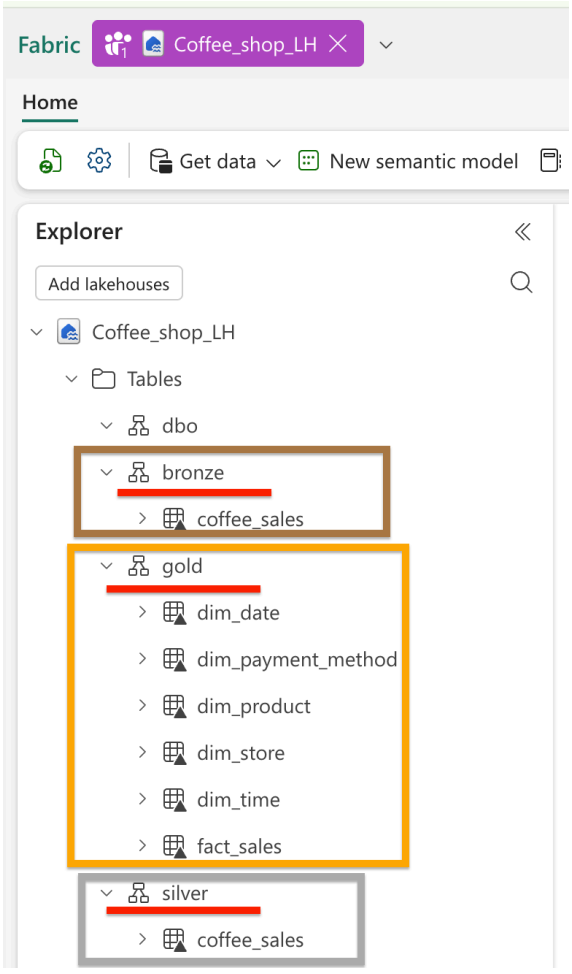
```
# ============================================================

# Quick validation

# ============================================================

spark.sql("""

SELECT month_key, COUNT(*) AS cnt

FROM gold.fact_sales

GROUP BY month_key

ORDER BY month_key

""").show(truncate=False)
```

	month_key	transaction_id	transaction_...	transaction_...	transaction_...	store_id	product_id	payment_m...	transaction_...	unit_price	total_amou
1	2024-02	4140	2024-02-01T17:...	2024-02-01	17	2	4404	DEBIT CARD	1	4.45	4.45
2	2024-02	4141	2024-02-01T12:...	2024-02-01	12	2	7505	CREDIT CARD	3	2.3	6.9
3	2024-02	4142	2024-02-01T10:...	2024-02-01	10	3	5047	CREDIT CARD	2	4.28	8.56
4	2024-02	4143	2024-02-01T14:...	2024-02-01	14	3	6284	CREDIT CARD	1	3.18	3.18
5	2024-02	4144	2024-02-01T09:...	2024-02-01	9	2	8834	CASH	2	3.7	7.4
6	2024-02	4145	2024-02-01T15:...	2024-02-01	15	1	7899	APPLE PAY	2	4.15	8.3
7	2024-02	4146	2024-02-01T13:...	2024-02-01	13	2	8101	DEBIT CARD	1	3.64	3.64
8	2024-02	4147	2024-02-01T11:...	2024-02-01	11	1	6279	CREDIT CARD	1	3.51	3.51

Final Pipeline Execution



After the pipeline ran, all tables were created successfully.



Power BI Report

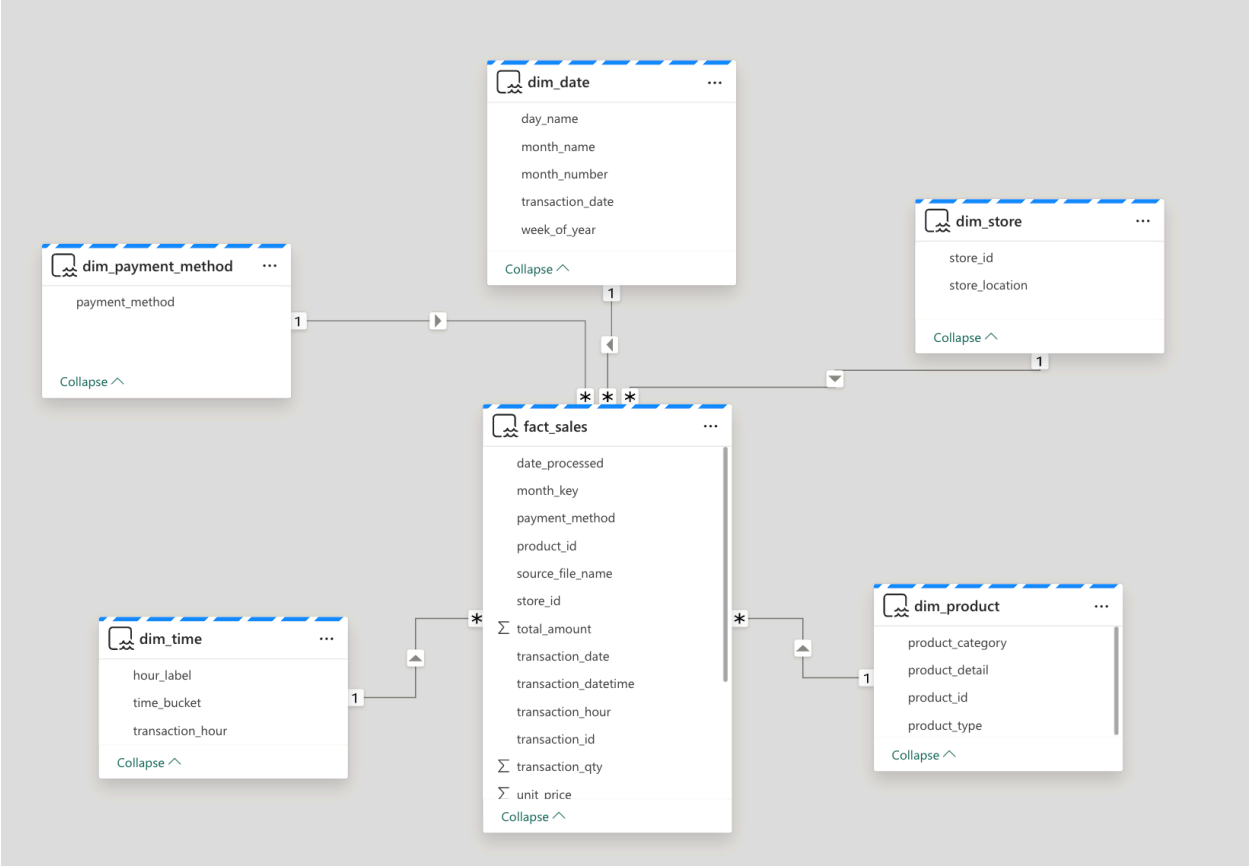
Connection

- Azure Databricks

Data Model

- Gold tables used
- Star schema

Semantic Model



- Dim → Fact : one → many relationship for remaining dimensions

Power BI Dashboard

- Sales performance is strongest in Astoria and Hell's Kitchen, with Brooklyn slightly lagging behind.
- Beverages (coffee and drinks) drive most revenue and quantity, outperforming snacks and pastries.
- Morning and afternoon are peak transaction periods, while evenings see lower activity.
- A few products dominate sales, while several items consistently underperform.
- Customer spending per transaction is moderate, indicating room for upselling and bundles.

Recommendations

- Increase staffing during **morning and afternoon peak hours** to reduce wait times and improve service.
- Prioritize inventory for **top-selling products** (Mocha, Hot Chocolate, Sandwiches) to avoid stockouts.
- Use **bundling and upselling** to increase Average Transaction Value (ATV).
- Run **targeted promotions for underperforming products** or consider removing low-performing items.
- Apply **best-performing store strategies** to improve sales in lower-revenue locations.
- Focus marketing efforts on **high-performing weekdays** and design offers to boost weekend sales.
- Monitor **weekly sales trends** to adjust restocking and promotions mid-month.
- Ensure **pricing consistency across stores** to maintain customer trust.
- Optimize checkout speed by supporting **preferred payment methods** during peak hours.
- Use category-level insights to **expand high-demand beverage offerings** and rationalize low-demand categories.

Conclusion/Errors

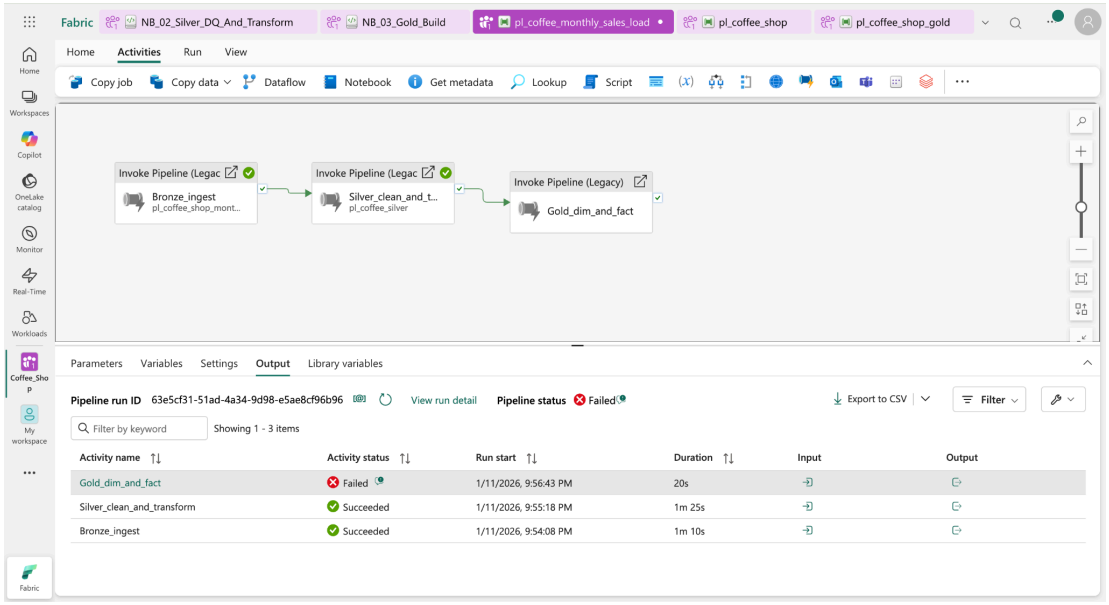
1. Spark Capacity Throttling (HTTP 430 – TooManyRequestsForCapacity)

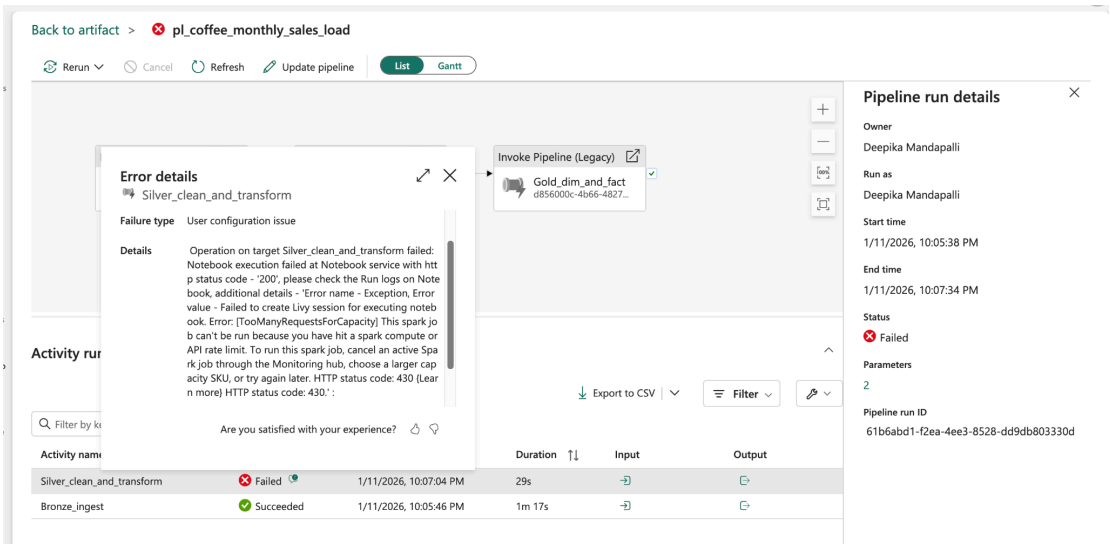
Issue: Pipeline execution failed with HTTP 430 errors when running multiple Spark notebook activities back-to-back. Fabric could not create new Livy sessions due to capacity limits.

Root Cause: Small Fabric capacity and rapid creation of multiple Spark sessions within a single pipeline run.

Resolution:

- Ensured sequential notebook execution (Bronze → Silver → Gold)
- Added **Wait activities** between Spark notebooks
- Limited notebook retries
- Avoided parallel Spark executions





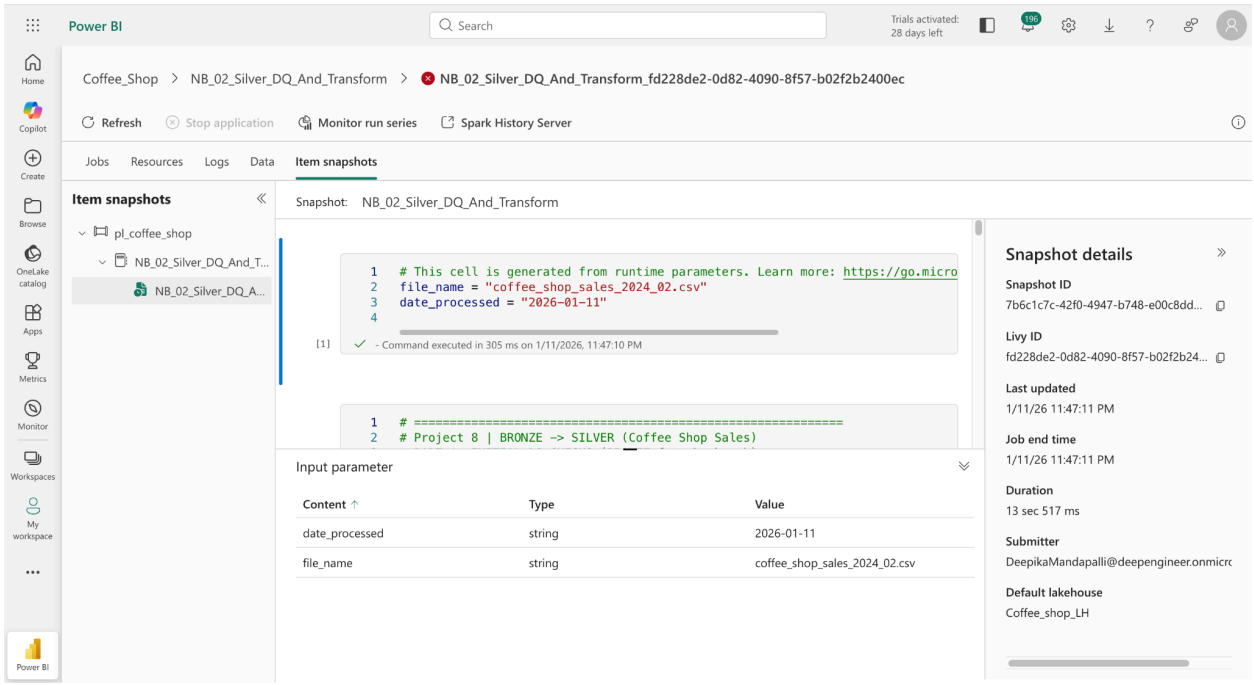
2. Pipeline Parameters Not Read Correctly in Notebooks

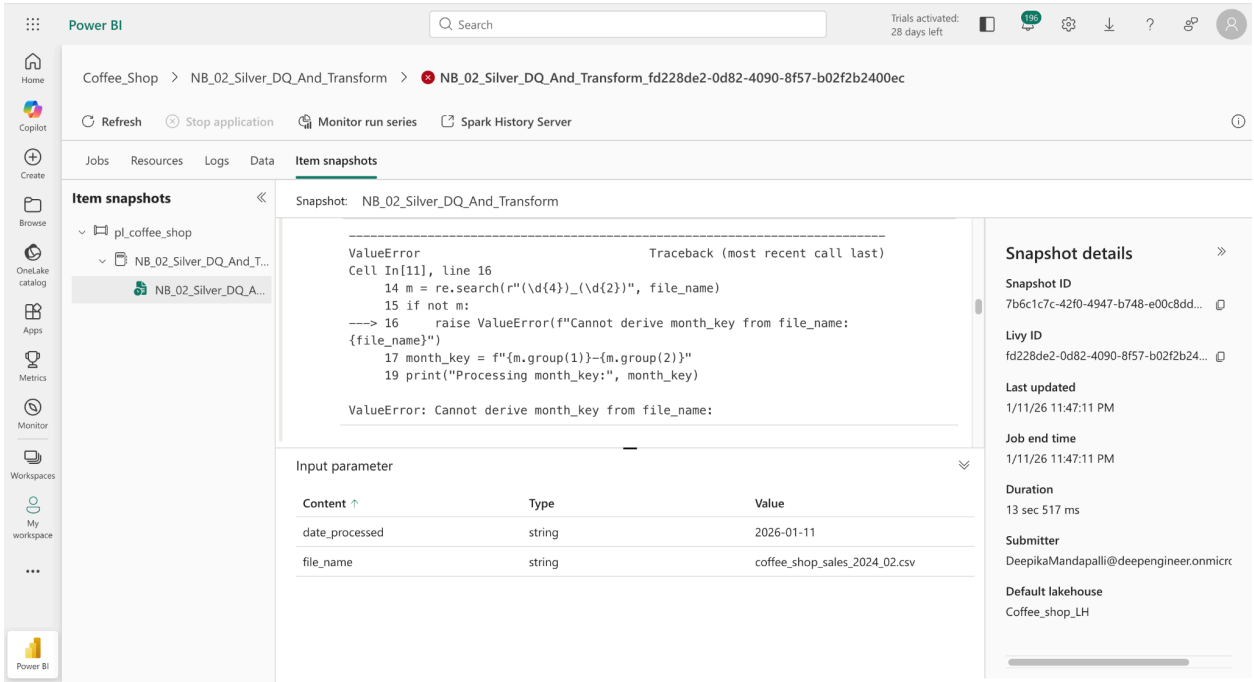
Issue: Bronze and Silver notebooks always processed January data even when February parameters were passed.

Root Cause: Pipeline parameters were injected as runtime Python variables, but notebooks were reading parameters using spark.conf.get(), causing mismatches.

Resolution:

- Check activity snapshot
- Removed spark.conf.get() usage
- Used pipeline-injected variables (file_name, date_processed) directly





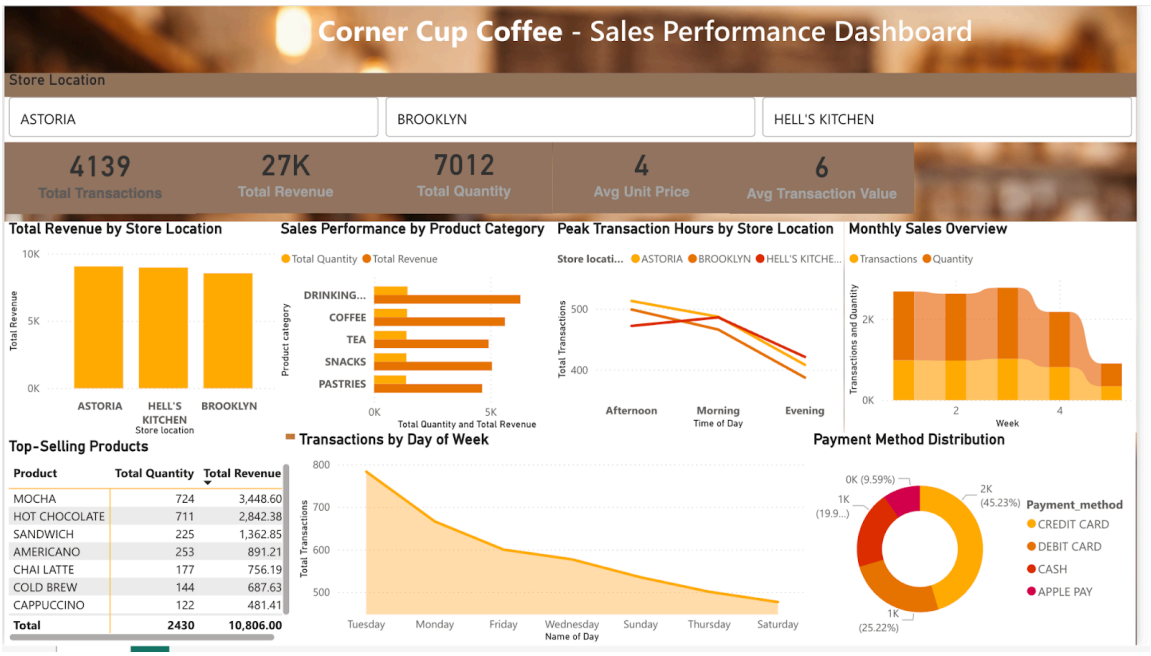
3. Power BI Direct Lake “Parquet File Not Found” Error

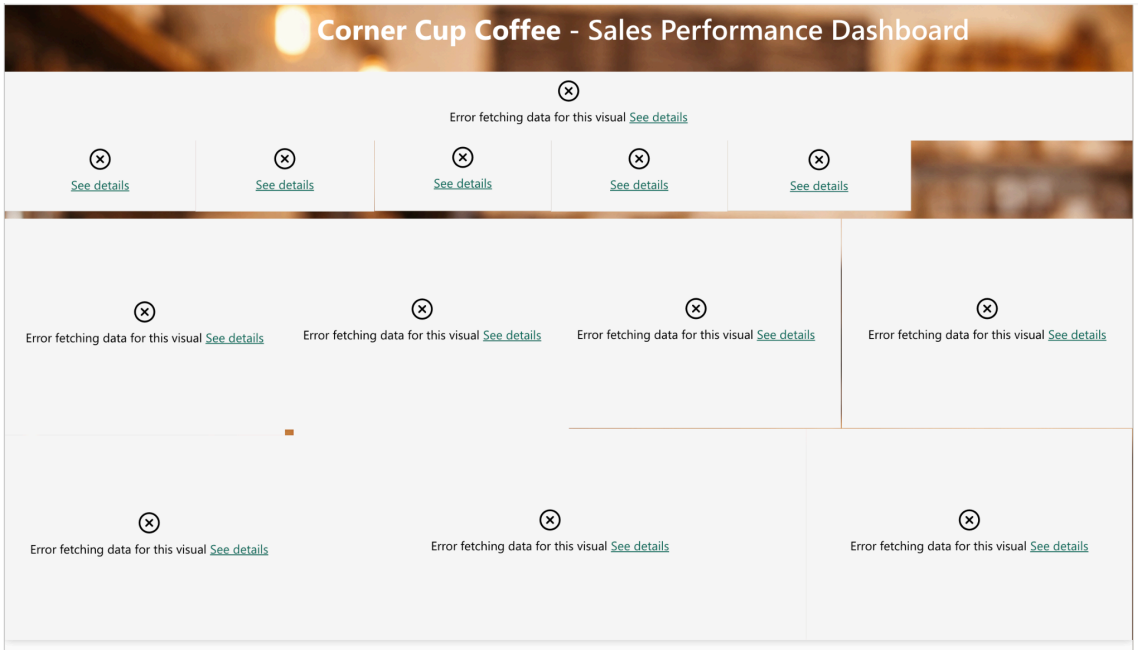
Issue: Power BI visuals failed with *“Parquet file not found in OneLake”* after pipeline completion.

Root Cause: Gold tables were overwritten using Spark, which replaced Parquet files. The direct Lake semantic model held stale file references.

Resolution:

- Refreshed the Semantic Model
- Created a new model





Synapse vs Databricks vs Fabric

Name	**Azure Synapse**	**Databricks**	**Microsoft Fabric**
Difference	Enterprise data warehousing + SQL analytics	Advanced big data processing & ML at scale	End-to-end analytics platform with maximum convenience
Convenience	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐⭐
Cost	Pay-per-query + Spark costs can spike	Can be expensive if clusters are not optimized	Most expensive -Predictable capacity-based pricing
Scenario	➔ Enterprise data warehouse is needed ➔ SQL workloads migrating to cloud	➔ Complex transformations ➔ Advanced Spark optimization is needed ➔ Heavy workloads	➔ When we want end-to-end analytics ➔ Strong Power BI integration is required

Use Fabric for analytics-driven BI solutions, Databricks for large-scale data engineering & AI, and Synapse for enterprise SQL warehousing.