CART CATALYST

GENERATION SINGAPORE

GROUP MEMBERS:

ALEX YONG

FAHMY M

HO CHOO GEOK

JANET KEH

CONTENT

EXECUTIVE SUMMARY

Olist, a leading Brazilian e-commerce platform, faced significant challenges due to its fragmented data architecture. With disparate datasets spread across multiple sources—including customer profiles, order details, product catalogs, seller information, and geolocation data—performance tracking and strategic decision-making were severely constrained. The lack of centralized governance and real-time visibility made it difficult for leadership to derive actionable insights and respond swiftly to market dynamics.

To address these limitations, the project will design and implement a modular, scalable data pipeline using Microsoft Fabric, a unified analytics platform that seamlessly integrates data engineering, data science, and business intelligence.

PROBLEM STATEMENT

Olist's leadership faces significant challenges in tracking business performance due to inconsistent, fragmented, and messy data spread across multiple sources. The absence of a centralized data architecture has made it difficult to generate reliable insights, hindering strategic planning and operational efficiency. Existing reporting mechanisms are limited in scope, lacking clarity, drill-down functionality, and real-time visibility, leaving decision-makers without the timely, actionable intelligence they need to steer the business effectively.

PROJECT OBJECTIVES

The primary objective of this project is to establish a unified data foundation by consolidating Olist's fragmented data sources into a Medallion architecture Lakehouse using Microsoft Fabric. This ensures consistent data access and governance across the organization.

Next, we aim to design a scalable and modular data pipeline by building reusable ETL components within Fabric Notebooks. Transformation logic, validation checks, and error handling will be abstracted into modular pipeline blocks to promote flexibility, maintainability, and ease of expansion.

Finally, the project seeks to enable drill-down reporting by delivering real-time dashboards equipped with drill-through capabilities and semantic modeling. These dashboards will empower business leaders with deeper insights and interactive visualizations for informed decision-making.

PROJECT TECH STACK

The project leverages a modern, cloud-native tech stack designed for scalability, modularity, and real-time analytics. Each component plays a distinct role across the data lifecycle—from ingestion to transformation, visualization, and monitoring.

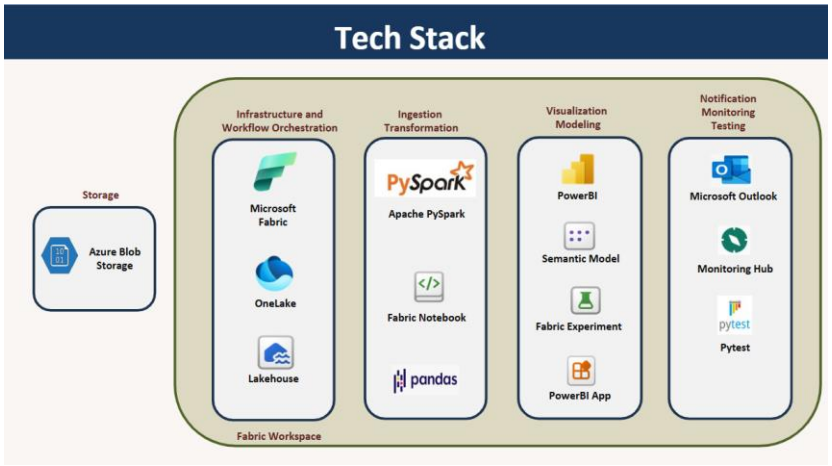
1) **Infrastructure & Storage**
   This foundation ensures scalable, ACID-compliant storage with schema flexibility and version control.
   Azure Blob Storage: Initial landing zone for raw CSVs from Olist.
   OneLake: Unified storage layer enabling seamless access across Microsoft Fabric workloads.
   Lakehouse: Centralized repository for structured Delta tables across Bronze, Silver, and Gold layers.

2) **Workflow Orchestration**
   Enables modular workflows with fabric pipeline design for flexible scheduling and execution.
   Azure Fabric: Execution environment for notebooks and pipeline automation.

3) **Ingestion & Transformation**
   Supports reusable pipeline blocks with embedded validation and error handling logic, enabling rapid iteration and debugging.
   Fabric Notebooks: Modular ETL logic for cleaning, enrichment, and validation.
   Apache PySpark: Distributed processing for scalable transformations.
   Pandas: Lightweight data manipulation for targeted operations.

4) **Visualization & Semantic Modeling**
   Bridges technical outputs with business insights, democratizing access to KPIs and enabling data-driven decisions.
   Power BI: Interactive dashboards for performance monitoring and decision support.
   Semantic Model: Star schema with DAX measures and hierarchies for drill-through analytics.
   Fabric Experiment: ML experimentation environment for market basket analysis.
   Power BI App: Centralized access to curated dashboards for stakeholders.

5) **Notification, Monitoring & Testing**
   Ensures proactive observability, robust error handling, and traceable validation across the Bronze-Silver-Gold pipeline.
   Microsoft Outlook: Email alerts triggered by pipeline failures or validation errors.
   Monitoring Hub: Centralized log repository with Power BI integration for pipeline health tracking.
   Pytest: Unit and integration testing of transformation logic and schema compliance.



PROJECT PIPELINE ARCHITECTURE

The pipeline is designed using a medallion architecture built on Microsoft Fabric. This modular structure ensures scalable ingestion, robust transformation, and ML-ready outputs, all governed by embedded validation and error handling.

1) Data Sources

- Azure Blob Storage serves as the initial landing zone for raw CSVs from Olist (orders, customers, products, payments, reviews, etc.).
- Files are ingested into Lakehouse using copy data, which load files from Azure Blob Storage into the Bronze Lakehouse Files folder.

2) Bronze Lakehouse: Ingestion Layer

Tasks:

- Copy raw CSVs from Blob Storage.
- Convert to Delta format and store in Lakehouse tables.

Validation & Error Handling:

- Schema-on-read validation.
- Overwrite mode ensures clean ingestion each run.
- Errors logged and surfaced via monitoring dashboard.

3) Silver Lakehouse: Cleaning Layer

Tasks:

- Standardized data types (e.g., timestamps, numerical fields).
- Normalize text fields and enrich with derived columns (e.g., purchase_year).
- Remove nulls from critical columns.

Validation & Error Handling:

- Type consistency checks.
- Null value enforcement.
- Schema validation between Bronze and Silver layers.

4) Gold Lakehouse: Transformation Layer

Tasks:

- Read curated tables from Silver Lakehouse.
- Merge into a unified, enriched dataset for ML and reporting.

Validation & Error Handling:

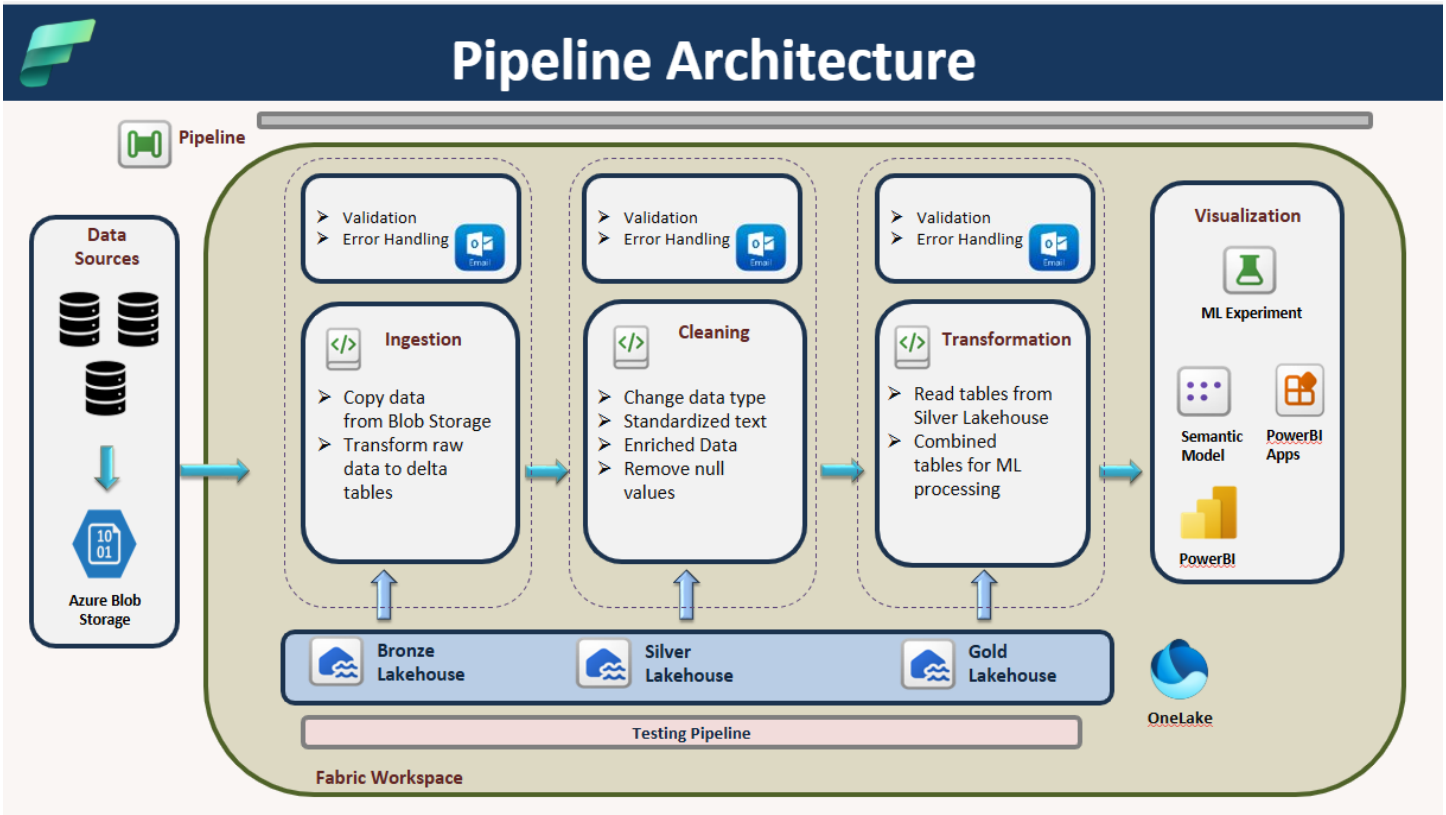
- Row count reconciliation between Silver and Gold.
- Schema alignment checks.
- Error logging and alerting via Outlook integration.

5) Visualization & ML Integration

Outputs:

- ML Experiment: Market Basket Analysis using Apriori algorithm.
- Semantic Model: Star schema with DAX measures and hierarchies.
- Power BI Apps: Dashboards for ingestion health, sales performance, and product insights.

PROJECT ARCHITECTURE DESIGN



MEDALLION ARCHITECTURE

The project adopts Medallion Architecture (Bronze, Silver, and Gold layers) within Microsoft Fabric Lakehouse to ensure data quality, flexibility, and governance across the analytics lifecycle. Each layer serves as a progressive checkpoint, refining data and enabling modular, traceable workflows.

Key Reasons to Adopt Medallion Architecture

1) Improved Data Quality & Integrity

- Each layer progressively refines the data:
  - **Bronze**: Raw, unfiltered source data
  - **Silver**: Cleaned, deduplicated, and standardized
  - **Gold**: Curated, enriched, and business-ready
- This layered refinement ensures consistency, traceability, and trust in downstream analytics.

2) Scalability & Performance

- Supports distributed processing for large datasets.
- Enables optimized queries by isolating transformations and aggregations in the Gold layer.

3) Flexibility & Reusability

- Raw data in Bronze can be reused to regenerate Silver or Gold when business logic changes.
- Modular design makes it easier to plug in new sources or update logic without breaking downstream flows.

4) Clear Separation of Concerns

- Engineers focus on ingestion and transformation (Bronze/Silver).
- Analysts and business users consume curated outputs (Gold).
- This role alignment reduces friction and improves collaboration.

5) Enhanced Data Governance & Lineage

- Each layer provides a checkpoint for auditing, validation, and compliance.
- It is easier to trace errors or anomalies back to their origin.

6) Cost Efficiency

- Only refined data is promoted to higher layers, reducing storage and computing costs.
- Avoids redundant transformations by centralizing logic in Silver.

Lakehouse in Medallion Architecture

1) Bronze Lakehouse: Raw Ingestion Layer

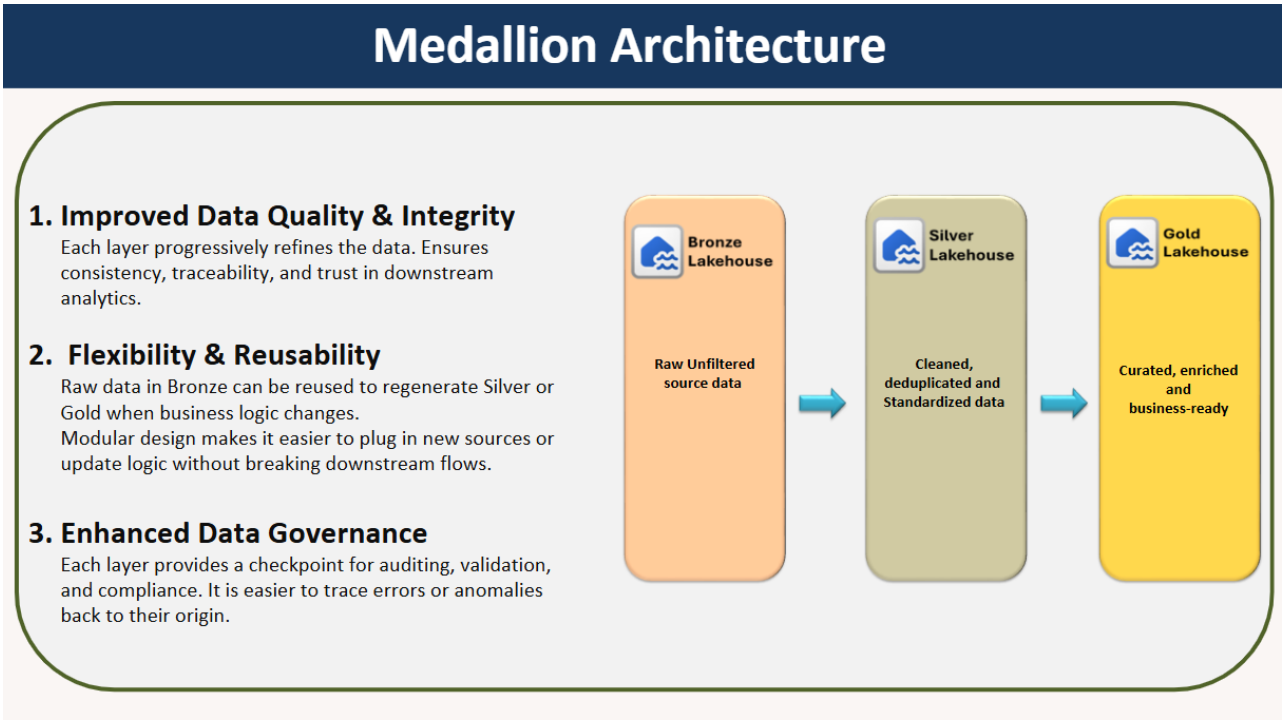
- **Purpose**: Capture raw, unfiltered source data from Azure Blob Storage.
- **Characteristics**:
  - Schema-on-read ingestion using inferred headers.
  - Delta format conversion for ACID compliance.
  - Overwrite mode ensures clean ingestion per run.
- **Role in Architecture**: Acts as the foundational layer for all downstream transformations, preserving original data fidelity.

2) Silver Lakehouse: Cleaning & Standardization Layer

- **Purpose**: Transform raw data into structured, validated datasets.
- **Characteristics**:
  - Type casting for timestamps, numeric, and strings.
  - Null value handling in critical fields.
  - Enrichment with derived columns (e.g., purchase_year).
- **Role in Architecture**: Establishes consistency and prepares data for analytical modeling and ML workflows

3) Gold Lakehouse: Enrichment & Business-Ready Layer

- **Purpose**: Consolidate and curate data for machine learning and reporting.
- **Characteristics**:
  - Merges multiple Silver Lakehouse tables into a unified schema.
  - Validates schema alignment and row count integrity.
  - Supports dimensional modeling for semantic layer integration.
- **Role in Architecture**: Delivers analytics-ready datasets for Power BI dashboards and ML experiments



BRONZE LAKEHOUSE

The Bronze Lakehouse serves as the foundational layer in the project pipeline, designed to ingest raw CSV files and convert them into Delta format for structured, traceable downstream processing. This layer prioritizes flexibility, simplicity, and fidelity to source data.

Process Flow

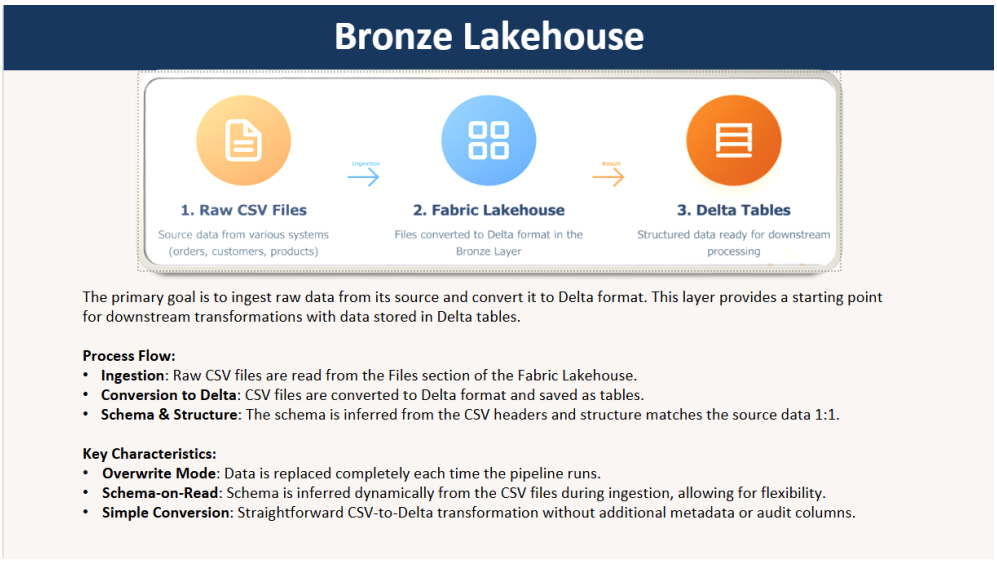
1. **Ingestion**
   - Raw CSV files are loaded from Azure Blob Storage to the Files section of the Bronze Lakehouse using copy operation.
   - Data originates from Olist's transactional systems (orders, customers, products, payments, reviews, etc.).
2. **Conversion to Delta**
   - CSVs are transformed into Delta tables, enabling ACID compliance and version control.
   - Tables are stored in the Bronze Lakehouse for downstream access.
3. **Schema & Structure**
   - Schema is inferred dynamically from CSV headers.
   - Structure mirrors the source data 1:1, preserving original column names and formats.

Key Characteristics

- **Overwrite Mode**
  - Each pipeline run replaces the existing Bronze tables entirely, ensuring clean ingestion and avoiding duplication.
- **Schema-on-Read**
  - Schema is not predefined; instead, it is inferred at runtime from the CSV files, allowing flexibility in handling evolving source formats.
- **Simple Conversion**
  - No enrichment, metadata tagging, or audit columns are added at this stage.
  - Focus remains on raw fidelity and straightforward transformation.

Role in Architecture

- Acts as the single source of truth for raw data.
- Enables reusability by allowing regeneration of Silver and Gold layers when business logic changes.
- Serves as a checkpoint for schema validation and ingestion monitoring.



NOTEBOOK SOURCE CODE FOR BRONZE LAKEHOUSE

Transform Raw CSV Files to Delta Tables for Bronze Lakehouse

1. Data has been loaded from the Azure Blob Storage to LH_Bronze files folder
2. The loaded csv files will be save as Delta tables in the LH_Bronze Tables folder

```python
# To load all the raw csv files in the folder to delta tables

# Import of libraries
import logging
import re


# Configure of logging for monitoring
logging.basicConfig(level=logging.INFO)

# To list the file path
files = notebookutils.fs.ls("Files/")

# To loop through all CSV files in the Files folder
for f in files:
    if f.name.endswith(".csv"):
        try:
            # Sanitize table name
            table_name = re.sub(r"\.csv$", "", f.name)
            table_name = re.sub(r"[^a-zA-Z0-9_]", "_", table_name)

            # To read CSV file
            df = spark.read.option("header", "true").csv(f.path)

            # Write the loaded CSV file to Delta table
            df.write.mode("overwrite").format("delta").saveAsTable(table_name)

        except Exception as e:
                    print("Error {e}")
                    raise Exception("Error in notebook NB_Bronze")
```

✓ - Session ready in 12 sec 636 ms. Command executed in 1 min 31 sec 893 ms by ChooGeok H (JDE07) on 7:44:05 PM, 10/07/25

SILVER LAKEHOUSE

The Silver Lakehouse is the second stage in the project medallion architecture, designed to transform raw ingested data from the Bronze layer into clean, reliable Delta tables. This layer emphasizes type consistency, basic quality checks, and enrichment logic to prepare data for downstream analytics and machine learning.

Process Flow

1. **Data Sourcing**
   - Reads raw Delta tables from the Bronze Lakehouse.
   - Inherits schema and structure directly from ingested CSVs.
2. **Cleaning & Conforming**
   - **Data Type Standardization**: Ensures consistent formatting for timestamps, numeric, and strings across all tables.
   - **Null Value Handling**: Drops or flags of nulls in critical fields (e.g., orders and reviews) based on business rules.
   - **Transformations**:
     - Adds derived columns like purchase_year and purchase_month to enhance temporal analysis.
     - Corrects column name typos (e.g., lenght → length) to ensure schema clarity and semantic alignment

Key Characteristics

1. **Type Consistency**
   a. Enforces uniform data types across all tables to support reliable joins, aggregations, and semantic modeling.
2. **Basic Quality Checks**
   a. Validates the presence of critical fields.
   b. Ensures schema integrity before passing data to the Gold layer.
3. **Enrichment Logic**
   - Introduces business-friendly features to support reporting and ML workflows.

Role in Architecture

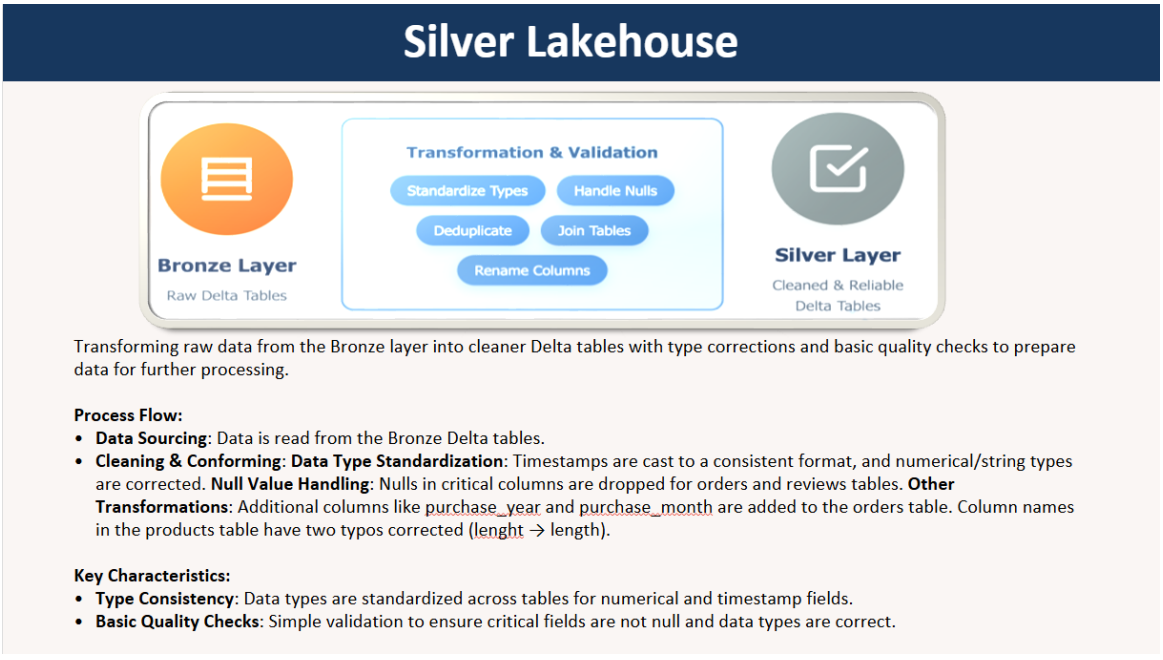
- Acts as the refinement checkpoint between raw ingestion and curated analytics.
- Enables modular validation and transformation logic that can be reused or extended.
- Serves as the foundation for semantic modeling, ensuring clean inputs for Power BI and ML experiments.

Cleaning of tables

Table Name	Column Name	Cleaning
silver_order_items	Shipping_limit_date	Change to date time
silver_order_items	Price, freight_value	Change to float, 2 decimal places
silver_order_items	Order_item_id	Change to integer
Silver_order_payments	Payment_sequential, payment_installments	Change to integer
Silver_order_payments	Payment_value	Change to float, 2 decimal places
Silver_order_reviews	Review_creation_date, review_answer_timestamp	Change to date time
Silver_order_reviews	Review_score	Change to integer, if not number remove rows
Silver_order_reviews	rows	Remove rows if order_id is null
Silver_orders	Order_approved_at, order_delivered_carrier_date, order_delivered_customer_date, order_estimated_delivery_date	Change to date time
Silver_products	Product_name_lenght, product_description_lenght	Change column name lenght to length

Silver Lakehouse Transformation

Table Name	Column(s)	Cleaning Instruction	Code Implementation
silver_order_items	Shipping_limit_date	Change to date time	.withColumn("Shipping_limit_date", to_timestamp(col("Shipping_limit_date")))
silver_order_items	Price, freight_value	Change to float, 2 decimal places	.withColumn("Price", round(col("Price").cast("float"), 2)) .withColumn("freight_value", round(col("freight_value").cast("float"), 2))
silver_order_items	Order_item_id	Change to integer	.withColumn("Order_item_id", col("Order_item_id").cast("integer"))
Silver_order_payments	Payment_sequential, payment_installments	Change to integer	.withColumn("Payment_sequential", col("Payment_sequential").cast("integer")) .withColumn("payment_installments", col("payment_installments").cast("integer"))
Silver_order_payments	Payment_value	Change to float, 2 decimal places	.withColumn("Payment_value", round(col("Payment_value").cast("float"), 2))
Silver_order_reviews	Review_creation_date, review_answer_timestamp	Change to date time	.withColumn("Review_creation_date", to_timestamp(col("Review_creation_date"))) .withColumn("review_answer_timestamp", to_timestamp(col("review_answer_timestamp")))
Silver_order_reviews	Review_score	Change to integer, if not number remove rows	.withColumn("Review_score", col("Review_score").cast("integer")) .na.drop(subset=["Review_score"])
Silver_order_reviews	rows	Remove rows if order_id is null	.na.drop(subset=["order_id"])
Silver_orders	Order_approved_at, order_delivered_carrier_date, order_delivered_customer_date, order_estimated_delivery_date	Change to date time	.withColumn("...", to_timestamp(col("..."))) for each specified date column.
Silver_products	Product_name_lenght, product_description_lenght	Change column name lenght to length	.withColumnRenamed("Product_name_lenght", "product_name_length"). withColumnRenamed("product_description_lenght", "product_description_length")



Transforming raw data from the Bronze layer into cleaner Delta tables with type corrections and basic quality checks to prepare data for further processing.

Process Flow:

- **Data Sourcing**: Data is read from the Bronze Delta tables.
- **Cleaning & Conforming**: **Data Type Standardization**: Timestamps are cast to a consistent format, and numerical/string types are corrected. **Null Value Handling**: Nulls in critical columns are dropped for orders and reviews tables. **Other Transformations**: Additional columns like purchase_year and purchase_month are added to the orders table. Column names in the products table have two typos corrected (lenght → length).

Key Characteristics:

- **Type Consistency**: Data types are standardized across tables for numerical and timestamp fields.
- **Basic Quality Checks**: Simple validation to ensure critical fields are not null and data types are correct.

NOTEBOOK SOURCE CODE FOR SILVER LAKEHOUSE

Cleaning and Transformation Steps

For the orders Table

1. Corrected Data Types: The order_purchase_timestamp column was converted from text into a proper timestamp, making it usable for date-based calculations.
2. Standardized Text: The order_status column was converted to lowercase to ensure consistency (e.g., "Delivered" and "delivered" are treated as the same).
3. Enriched Data: Two new columns, purchase_year and purchase_month, were created by extracting the year and month from the timestamp. This makes filtering and grouping by date much easier.
4. Removed Bad Data: Rows with missing (null) values in critical columns like order_id, customer_id, or order_purchase_timestamp were deleted to improve data integrity.

For All Other Tables

Renamed for Clarity: The primary transformation for the other tables was renaming them from their raw source names (e.g., olist_customers_dataset) to clean, intuitive names for the Silver layer (e.g., silver_customers).

```python
# Import of libraries
import logging
from pyspark.sql.functions import col, to_timestamp, lower, year, month, round

# Configure of logging for monitoring
logging.basicConfig(level=logging.ERROR)

# Configuration mapping Bronze names to clean Silver names
config = {
    "orders":       {"bronze": "olist_orders_dataset",      "silver": "silver_orders"},
    "customers":    {"bronze": "olist_customers_dataset",   "silver": "silver_customers"},
    "order_items":  {"bronze": "olist_order_items_dataset", "silver": "silver_order_items"},
    "products":     {"bronze": "olist_products_dataset",    "silver": "silver_products"},
    "sellers":      {"bronze": "olist_sellers_dataset",     "silver": "silver_sellers"},
    "payments":     {"bronze": "olist_order_payments_dataset", "silver": "silver_order_payments"},
    "reviews":      {"bronze": "olist_order_reviews_dataset",  "silver": "silver_order_reviews"},
    "geolocation":  {"bronze": "olist_geolocation_dataset",    "silver": "silver_geolocation"},
    "translations": {"bronze": "product_category_name_translation", "silver": "silver_product_translations"}
}

```

```python

for key, names in config.items():
    bronze_table_name = names["bronze"]
    silver_table_name = names["silver"]

    try:
        bronze_df = spark.read.table(f"LH_Bronze.{bronze_table_name}")

        df_to_write = None

        if key == "orders":
            # This block now includes all transformations
            cleaned_df = bronze_df \
                .withColumn("order_purchase_timestamp", to_timestamp(col("order_purchase_timestamp"))) \
                .withColumn("order_approved_at", to_timestamp(col("order_approved_at"))) \
                .withColumn("order_delivered_carrier_date", to_timestamp(col("order_delivered_carrier_date"))) \
                .withColumn("order_delivered_customer_date", to_timestamp(col("order_delivered_customer_date"))) \
                .withColumn("order_estimated_delivery_date", to_timestamp(col("order_estimated_delivery_date"))) \
                .withColumn("order_status", lower(col("order_status"))) \
                .withColumn("purchase_year", year(col("order_purchase_timestamp"))) \
                .withColumn("purchase_month", month(col("order_purchase_timestamp"))) \
                .na.drop(subset=["order_id", "customer_id", "order_purchase_timestamp"])
            df_to_write = cleaned_df

        elif key == "order_items":
            cleaned_df = bronze_df \
                .withColumn("Shipping_limit_date", to_timestamp(col("Shipping_limit_date"))) \
                .withColumn("Price", round(col("Price").cast("float"), 2)) \
                .withColumn("freight_value", round(col("freight_value").cast("float"), 2)) \
                .withColumn("Order_item_id", col("Order_item_id").cast("integer"))
            df_to_write = cleaned_df

        elif key == "payments":
            cleaned_df = bronze_df \
                .withColumn("Payment_sequential", col("Payment_sequential").cast("integer")) \
                .withColumn("payment_installments", col("payment_installments").cast("integer")) \
                .withColumn("Payment_value", round(col("Payment_value").cast("float"), 2))
            df_to_write = cleaned_df

```

```
60          elif key == "reviews":
61              cleaned_df = bronze_df \
62                  .withColumn("Review_creation_date", to_timestamp(col("Review_creation_date"))) \
63                  .withColumn("review_answer_timestamp", to_timestamp(col("review_answer_timestamp"))) \
64                  .withColumn("Review_score", col("Review_score").cast("integer")) \
65                  .na.drop(subset=["order_id", "Review_score"])
66              df_to_write = cleaned_df
67
68          elif key == "products":
69              cleaned_df = bronze_df \
70                  .withColumnRenamed("Product_name_lenght", "product_name_length") \
71                  .withColumnRenamed("product_description_lenght", "product_description_length")
72              df_to_write = cleaned_df
73
74          else:
75              df_to_write = bronze_df
76
77
78          df_to_write.write \
79              .mode("overwrite") \
80              .format("delta") \
81              .option("overwriteSchema", "true") \
82              .saveAsTable(f"LH_Silver.{silver_table_name}")
83
84      except Exception as e:
85          print(f"ERROR processing {bronze_table_name}: {e}")
86          raise Exception(f"Silver Lakehouse files failed to load: {bronze_table_name}") from e
```

GOLD LAKEHOUSE

The Gold Lakehouse represents the final refinement stage in the project's pipeline. Its primary goal is to consolidate cleaned data from the Silver Lakehouse into a single, enriched dataset optimized for machine learning workflows and business reporting. This layer ensures schema consistency, data integrity, and analytical readiness.

Transformation Process

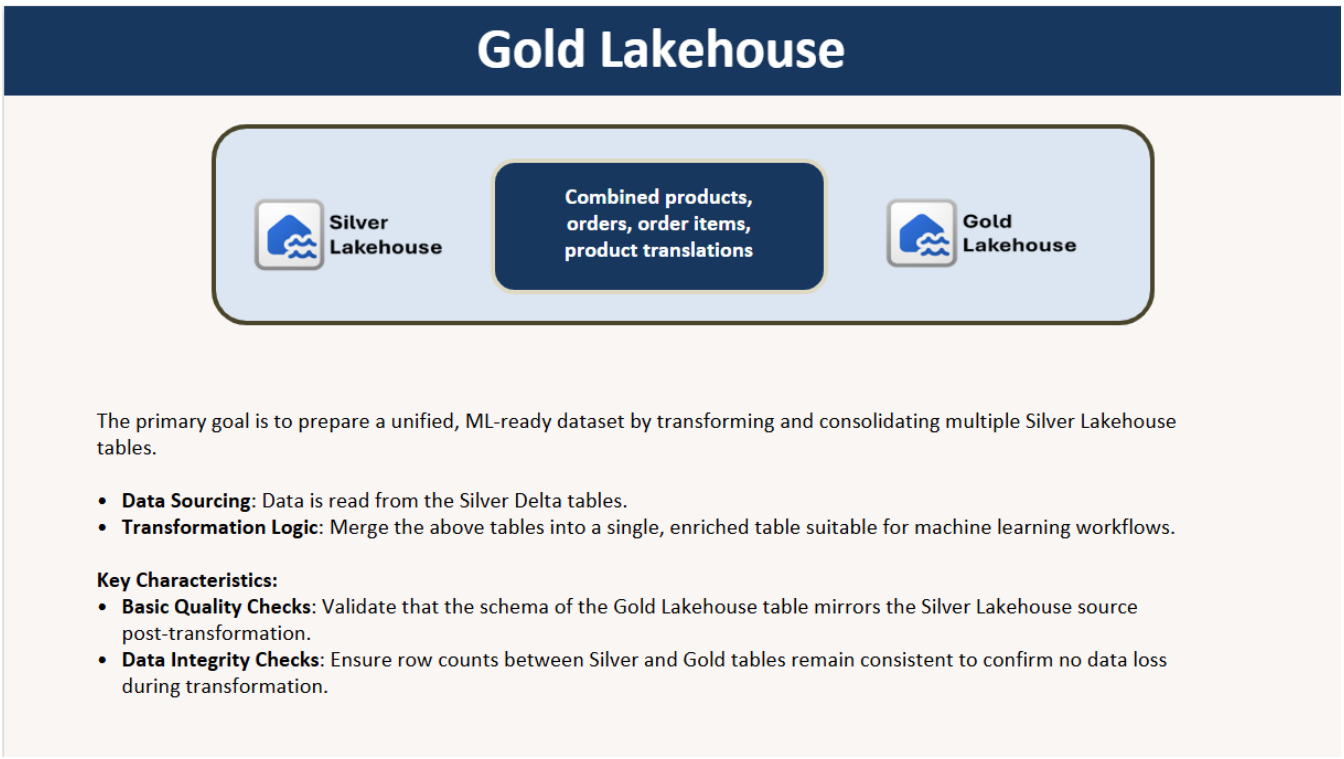
1. **Data Sourcing**
   - Reads curated Delta tables from the Silver Lakehouse.
   - Includes key entities such as products, orders, order items, and product translations.
2. **Transformation Logic**
   - Merges multiple Silver tables into a unified schema.
   - Applies enrichment logic to support downstream ML experiments and semantic modeling.

Key Characteristics

- **Basic Quality Checks**
  - To validate that the schema of the Gold Lakehouse table mirrors its Silver source post-transformation.
  - Ensures column alignment and data type consistency.
- **Data Integrity Checks**
  - Reconciles row counts between Silver and Gold tables to confirm no data loss.
  - Flags anomalies or mismatches for immediate resolution.

Role in Architecture

- Serves as the launchpad for ML experiments, including market basket analysis and recommendation engines.
- Provides a business-ready dataset for semantic modeling and Power BI dashboards.
- Act as a final checkpoint for validation before data enters the reporting and analytics layer



GOLD LAKEHOUSE DIMENSIONAL SCHEMA

Foundation for ML & Analytics

The Gold Lakehouse schema is the culmination of the project pipeline, designed to organize enriched data into a dimensional model that supports advanced analytics, machine learning, and stakeholder reporting. This schema enables efficient querying, slicing, and aggregation across key business domains.

Purpose

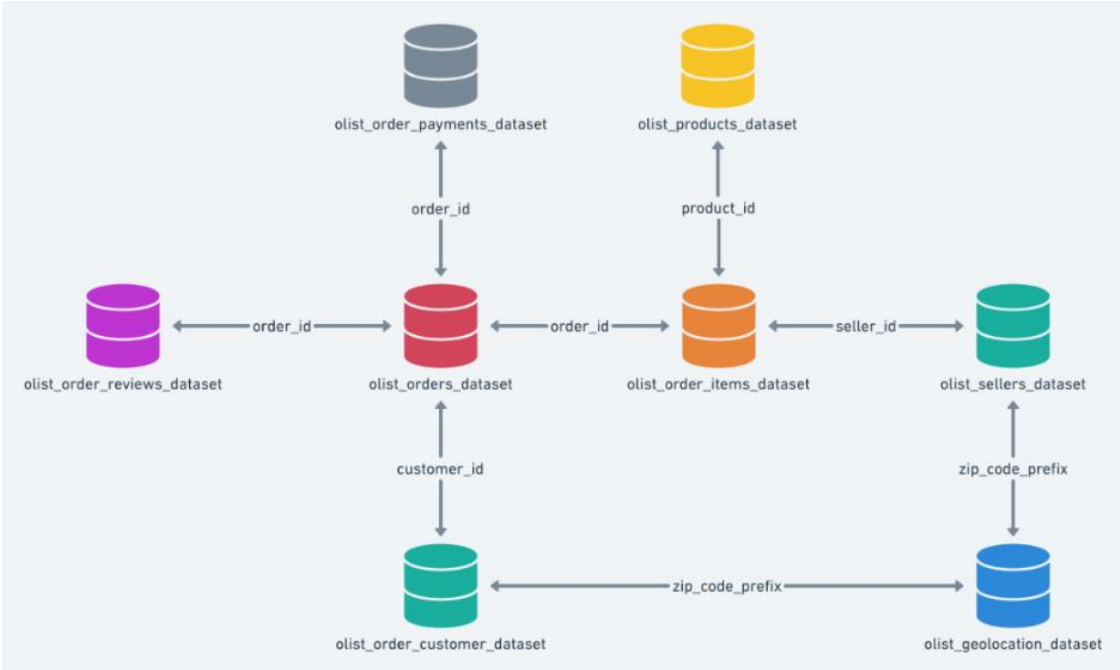
- To deliver a machine learning ready dataset by consolidating cleaned Silver Lakehouse tables into a structured, dimensional format.
- To support semantic modeling and Power BI dashboards with optimized relationships and hierarchies.

Schema Design Highlights

- **Star Schema Structure**
  - Central fact tables (e.g., orders, order_items) linked to dimension tables (e.g., customers, products).
  - Enables efficient joins and aggregations for reporting and ML workflows.
- **Semantic Alignment**
  - Cleaned column names and standardized data types ensure compatibility with Power BI and DAX modeling

Benefits

- **Query Optimization**
  - Dimensional design reduces complexity and improves performance for dashboard visuals and ML feature extraction.
- **Business Readiness**
  - Tables are curated for stakeholder consumption, enabling drill-through insights and KPI tracking.
- **ML Enablement**
  - Unified schema supports feature engineering for market basket analysis and recommendation engines.



PROJECT ARCHITECTURE DESIGN

Dataset Descriptions

Entity Name	Description	IsNull	Unique Values
Table Name: customers			
customer_id	key to the orders dataset. Each order has a unique customer_id.	No	99441
customer_unique_id	unique identifier of a customer.	No	96096
customer_zip_code_prefix	first five digits of customer zip code	No	
customer_city	customer city name	No	4119
customer_state	customer state	No	27
Table Name: geolocation			
geolocation_zip_code_prefix	first 5 digits of zip code	No	
geolocation_lat	latitude	No	96096
geolocation_lng	longitude	No	
geolocation_city	city name	No	8011
geolocation_state	state	No	27
Table Name: order_items			
order_id	order unique identifier	No	98666
order_item_id	sequential no. identifying no. of items included in the same order.	No	
product_id	product unique identifier	No	32951
seller_id	seller unique identifier	No	3095
shipping_limit_date	Shows the seller shipping limit date for handling the order over to the logistic partner.	No	
price	item price	No	
freight_value	item freight value item (if an order has more than one item the freight value is splitted between items)	No	
Table Name: order_payments			
order_id	unique identifier of an order.	No	99440
payment_sequential	customer may pay an order with more than one payment method. If he does so, a sequence will be created to accommodate all payments.	No	
payment_type	method of payment chosen by the customer.	No	5
payment_installments	No. of installments chosen by the customer.	No	
payment_value	transaction value.	No	
Table Name: order_reviews			
review_id	unique review identifier	No	98410
order_id	unique order identifier	No	98673
review_score	Note ranging from 1 to 5 given by the customer on a satisfaction survey.	No	

review_comment_title	Comment title from the review left by the customer, in Portuguese.	87.7k	4527
review_comment_message	Comment message from the review left by the customer, in Portuguese.	58.2k	36.2k
review_creation_date	Shows the date in which the satisfaction survey was sent to the customer.	No	
review_answer_timestamp	Shows satisfaction survey answer timestamp.	No	
Table Name: orders			
order_id	unique identifier of the order.	No	99441
customer_id	key to the customer dataset. Each order has a unique customer_id.	No	99441
order_status	Reference to the order status (delivered, shipped, etc).	No	
order_purchase_timestamp	Shows the purchase timestamp.	No	
order_approved_at	Shows the payment approval timestamp	No	
order_delivered_carrier_date	Shows the order posting timestamp. When it was handled to the logistic partner.	1783	
order_delivered_customer_date	Shows the actual order delivery date to the customer.	2965	
order_estimated_delivery_date	Shows the estimated delivery date that was informed to customer at the purchase moment.	No	
Table Name: products			
product_id	unique product identifier	No	32951
product_category_name	root category of product, in Portuguese.	610	73
product_name_lenght	number of characters extracted from the product name.	610	
product_description_lenght	number of characters extracted from the product description.	610	
product_photos_qty	number of product published photos	610	
product_weight_g	product weight measured in grams.	2	
product_length_cm	product length measured in centimeters.	2	
product_height_cm	product height measured in centimeters.	2	
product_width_cm	product width measured in centimeters.	2	
Table Name: sellers			
seller_id	seller unique identifier	No	3095
seller_zip_code_prefix	first 5 digits of seller zip code	No	
seller_city	seller city name	No	611
seller_state	seller state	No	23
Table Name: product_category			
product_category_name	category name in Portuguese	No	71
product_category_name_english	category name in English	No	71

NOTEBOOK SOURCE CODE FOR GOLD LAKEHOUSE

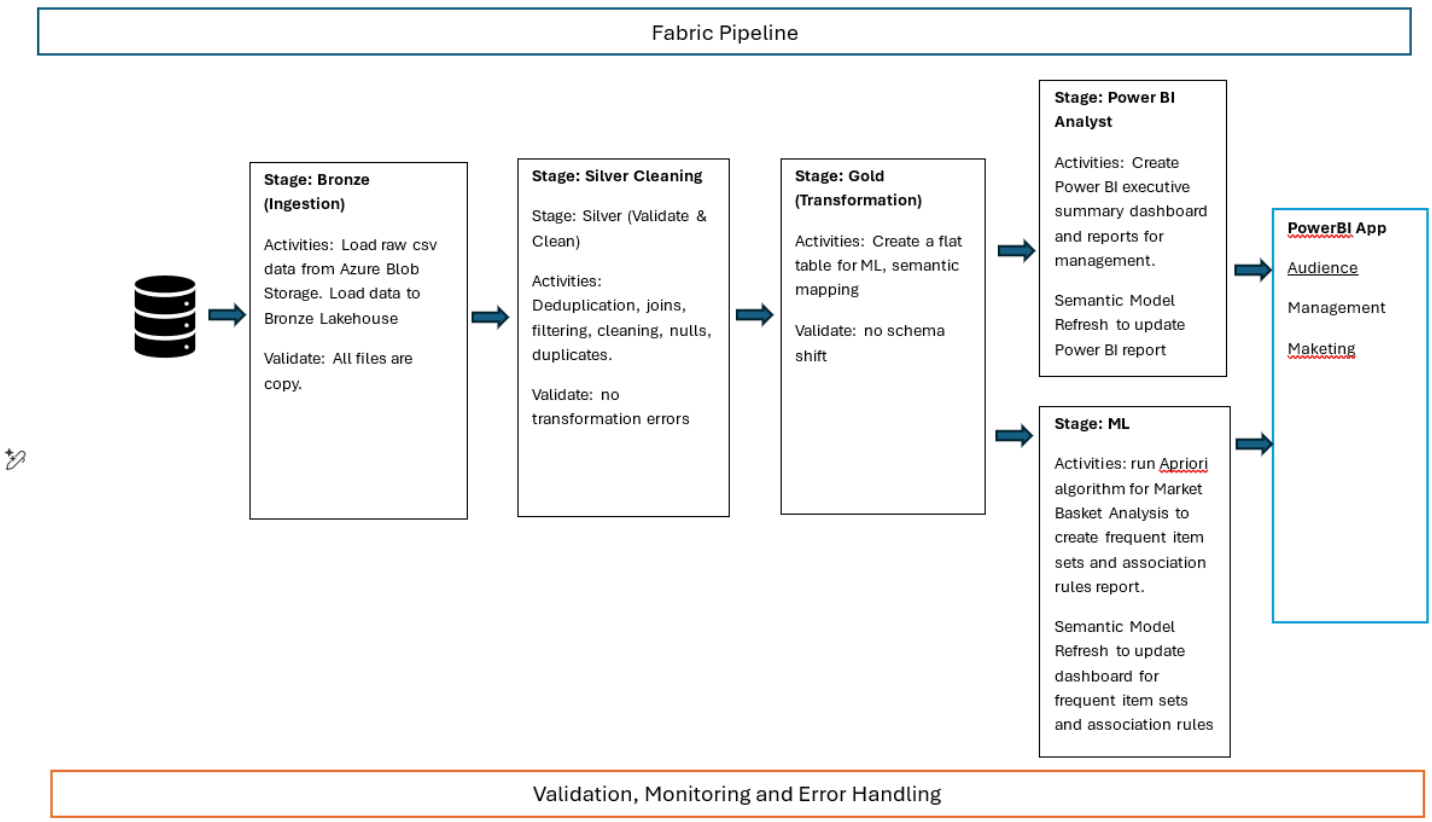
To read tables from Silver Lakehouse and save to Gold Lakehouse

```python
# To read tables from LH_Silver lakehouse and Write to LH_Gold lakehouse
# Use of concurrent.futures to parallelize reads and writes across tables

from pyspark.sql import SparkSession
from concurrent.futures import ThreadPoolExecutor, as_completed
import logging

logging.basicConfig(level=logging.ERROR)

# Start Spark session
spark = SparkSession.builder.getOrCreate()

# Define silver_tables for the executor function
silver_tables = [t.name for t in spark.catalog.listTables('LH_Silver')]

# Function to iterate through the tables, read and rename, then save to the LH_Gold lakehouse
def migrate_table(table):
    try:
        logging.info(f"Processing {table}")
        df = spark.read.table(f"LH_Silver.{table}")
        gold_table = table.replace('silver_', 'gold_', 1) if table.startswith('silver_') else f'gold_{table}'
        df.write.mode("overwrite").format("delta").saveAsTable(f"LH_Gold.{gold_table}")
        logging.info(f"Finished saving {gold_table}")
    except Exception as e:
        # exc_info ensures the full traceback is logged, which is crucial for debugging
        logging.error(f'Error processing table {table}:{e}', exc_info=True)
        raise Exception(f"Gold Lakehouse files failed to load: {f.name}") from e


# To map through the tables in LH_Silver
with ThreadPoolExecutor() as executor:
    futures = {executor.submit(migrate_table, table): table for table in silver_tables}
    for future in as_completed(futures):
        try:
            # This will raise any exception from the thread
            future.result()
        except Exception as e:
            logging.error(f"Notebook failed due to: {e}", exc_info=True)
            raise Exception(f"Gold Lakehouse files failed to load: {f.name}") from e

```

FABRIC PIPELINE DESIGN

This structure of the end-to-end workflow for our project pipeline design is as shown below. The pipeline is segmented into Five key phases, Medallion stages, Validation, Market Basket Analysis, Error Handling, and Power BI Analysis, each representing a critical stage in transforming raw data into actionable insights.

Pipeline Architecture Design



1) Medallion Stages

- Data Input & Extraction: Gathering raw datasets from various sources.
- Validation & Cleaning: Identifying and correcting inconsistencies, missing values, and formatting issues.
- Transformation: Structuring the data into a usable format for downstream processes.

2) Validation

This initial phase validates the data integrity and schema shift of the various stages. It includes:

- **Bronze Lakehouse Validation**: Establish a clean ingestion baseline with schema-on-read flexibility and overwrite mode enforcement
- **Silver Lakehouse Validation**: Ensure clean and standardized data maintains structural integrity before enrichment and transformation
- **Gold Lakehouse Validation**: Guarantee that the final ML-ready dataset is complete, consistent, and trustworthy for semantic modeling and reporting.

3) Market Basket Analysis

This phase applies Apriori association rule mining techniques to uncover patterns in transactional data:

- Data Preparation: Merge datasets to create a table for data mining.
- Analysis Execution: Applying algorithms to identify product affinities.
- Insight Generation: Producing recommendations and strategic findings.

4) Power BI Analysis

The final stage focuses on data visualization and reporting:

- Dashboard Creation: Building interactive visualizations in Power BI.
- Insight Delivery: Presenting findings to stakeholders for informed decision-making.

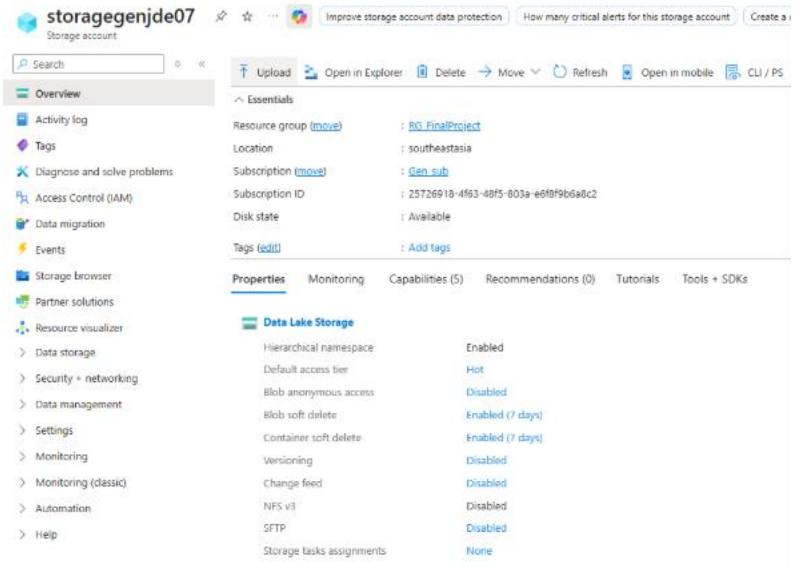
5) Error Handling

Error handling is embedded throughout the pipeline to ensure robustness:

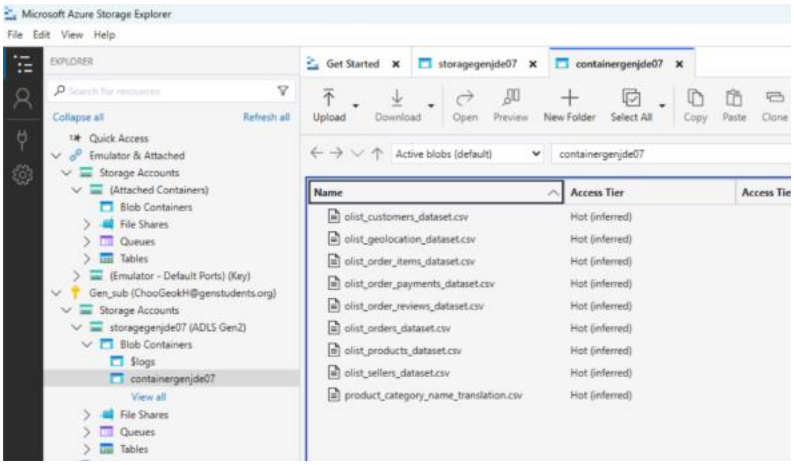
- Validation Phase: Early detection of anomalies.
- Dedicated Module: A separate error-handling loop captures and logs issues for review and resolution.

CREATION OF AZURE BLOB STORAGE

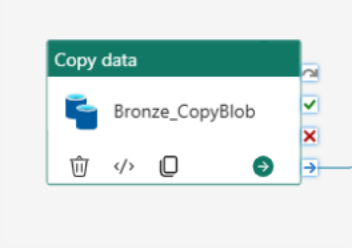
1. Create an azure blob storage to store the raw files



2. Microsoft Azure Storage Explorer
   The raw files are loaded into the container

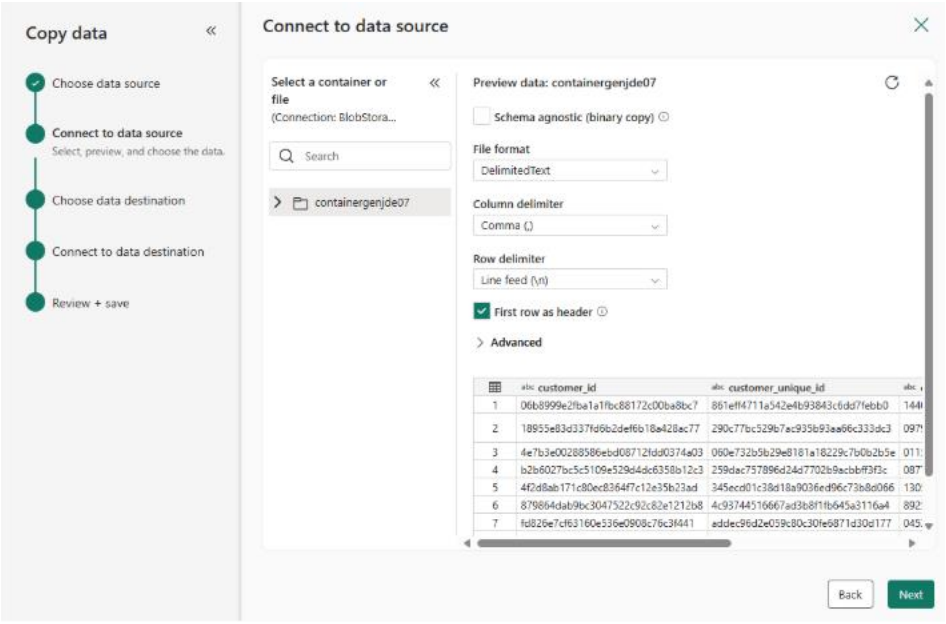


ACTIVITIES IN THE PIPELINE

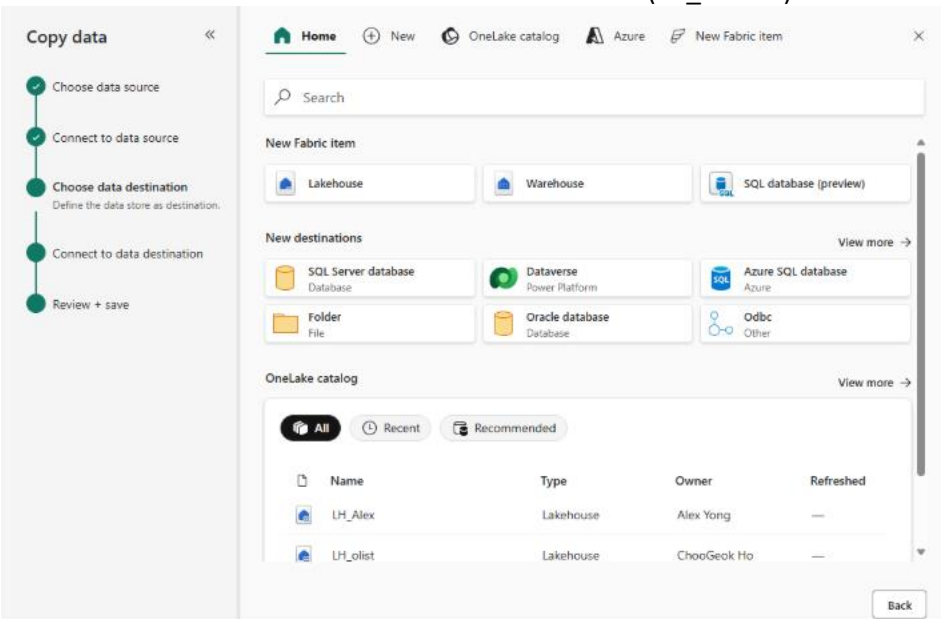


1) To copy data from Azure Blob Storage to Bronze Lakehouse

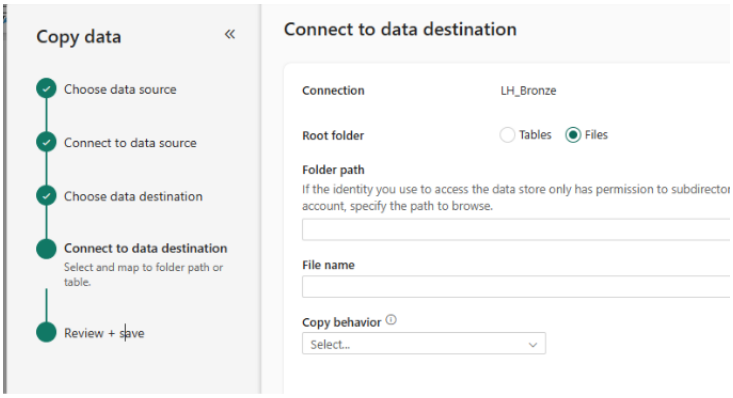
a. Create pipeline and select copy data
Choose data source, select Azure Blob Storage and the container. Connect to the data source. Preview of the data when connection is established.

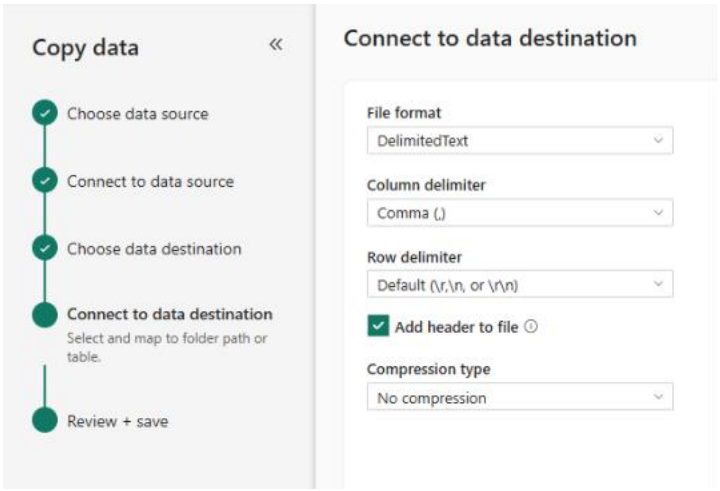


b. Choose data destination. Select the Lakehouse (LH_Bronze)

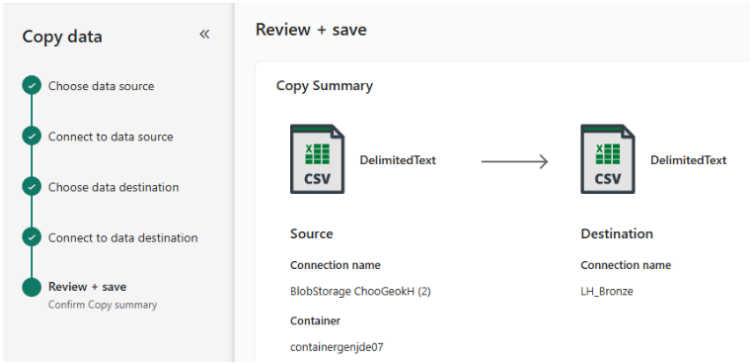


c. Connect to data destination
Select the Root folder as Files to save the raw files into the Files folder

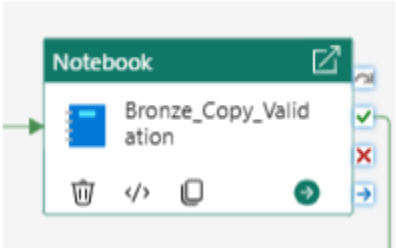




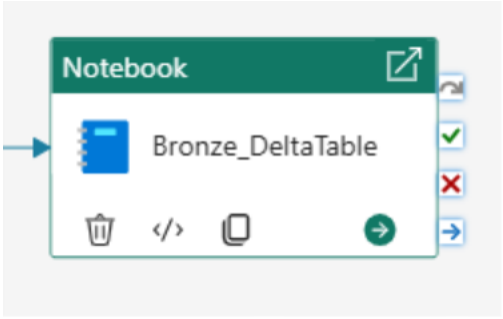
d. Review and save



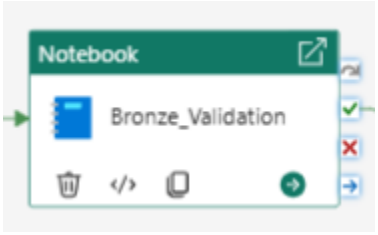
2) **To validate the copy operation from Azure Blob Storage to Bronze Lakehouse**
Run notebook: NB_Bronze_Copy_Validation



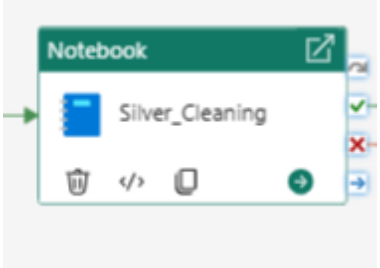
3) **To run notebook to load raw files into Delta Tables**
The notebook will load all csv files in the Files folder to Delta Tables.
Run notebook: NB_Bronze



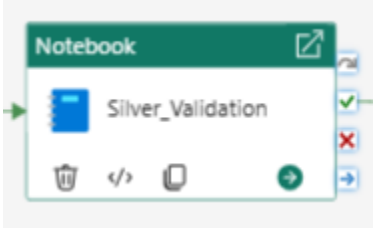
4) **To validate the all csv files have been converted to delta tables in the Bronze Lakehouse**
Run notebook: NB_Bronze_Validation



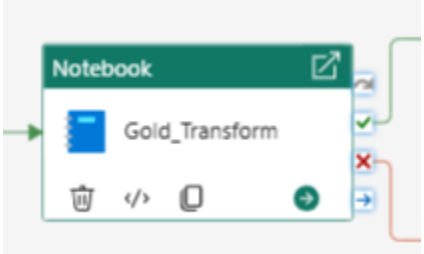
5) **Load data from Bronze Lakehouse and save it to Silver Lakehouse. Perform Cleaning of dataset**
Run Notebook: NB_Silver_Transform



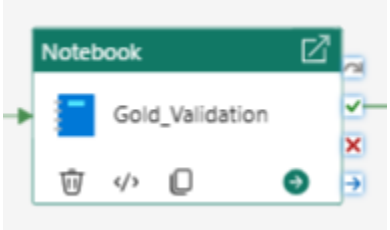
6) **To validation the tables in Silver Lakehouse**
Run Notebook: NB_Silver_Validation



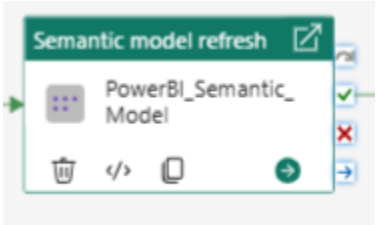
7) **Load dataset from Silver Lakehouse and save into Gold Lakehouse. Perform transformation of dataset.**
Run Notebook: NB_Gold



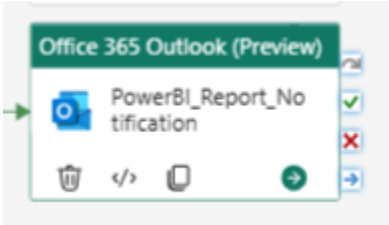
8) **Validation for the tables in LH_Gold Lakehouse**
Run Notebook: NB_Gold_Validation



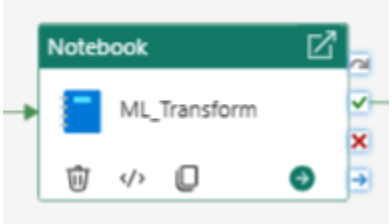
9) **Semantic Model Refresh for Power BI report**
Semantic Model: SM_PowerBI



10) **Email Notification for semantic model refresh and update of report**

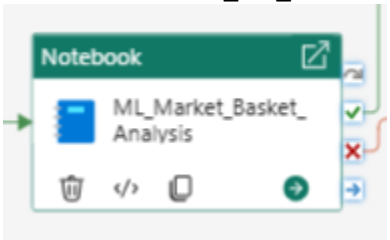


11) **To load Delta tables from LH_Gold Lakehouse and transform into data suitable for ML model training**
Run Notebook: NB_ML_Clean



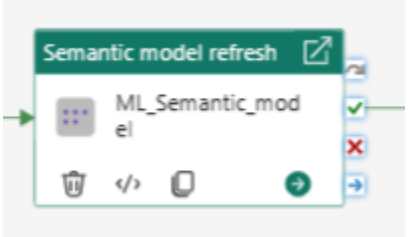
12) To use Apriori Model for Market Basket Analysis

Run Notebook: NB_ML_MarketBasket

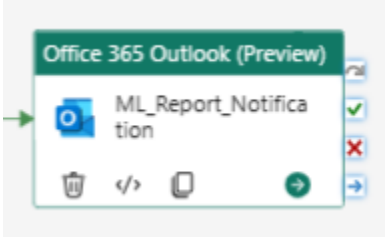


13) Semantic Model refresh for ML report creation

Semantic Model: SM_ML



14) Email Notification for semantic model refresh and update of ML report



MARKET BASKET ANALYSIS

Market Basket Analysis (MBA) is a powerful data mining technique used to uncover associations between items in transactional data. By identifying patterns in customer purchasing behavior, MBA enables businesses to optimize product placement, personalize recommendations, and drive revenue through targeted promotions.

What Is Market Basket Analysis?

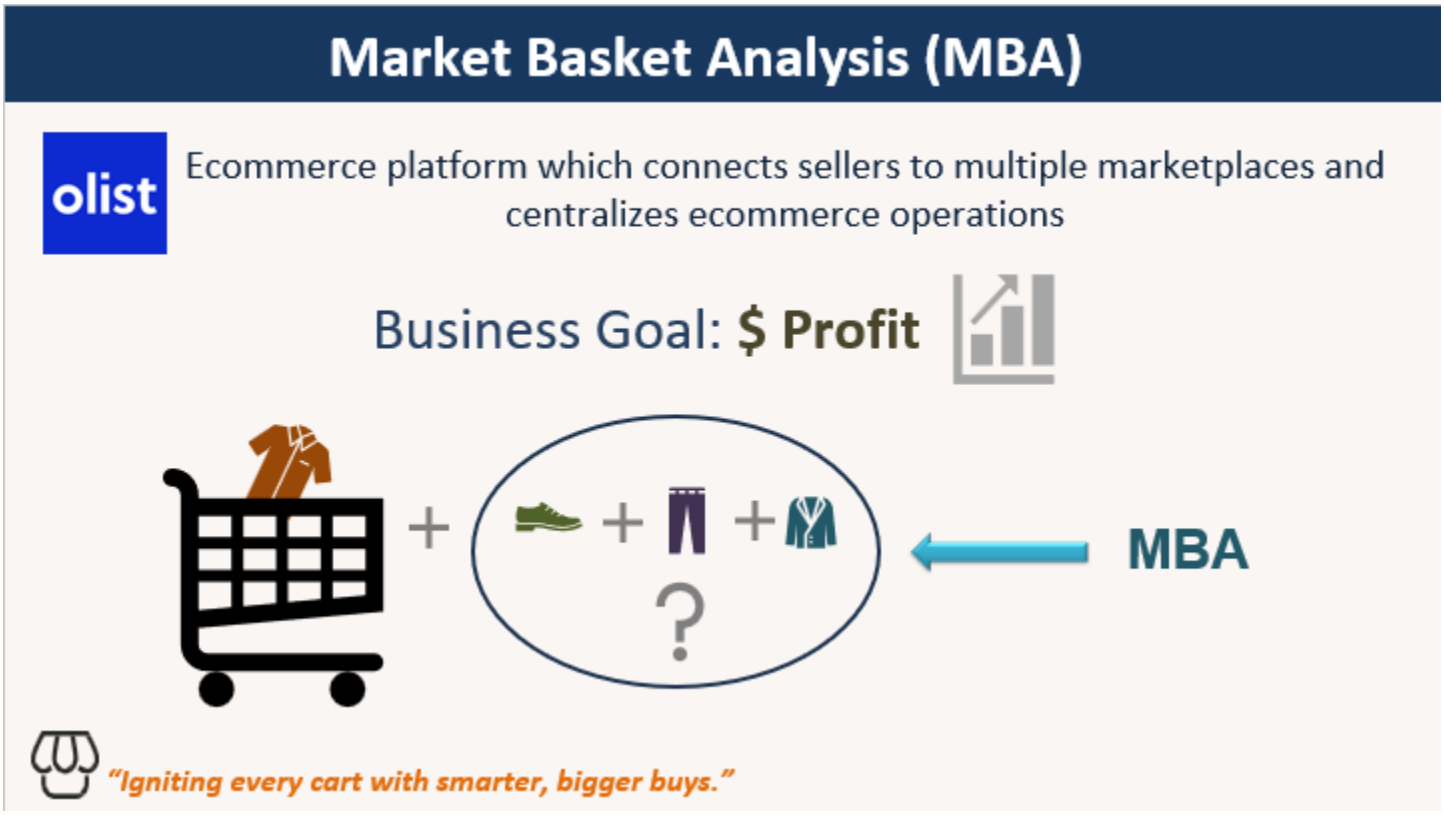
- **Definition**: MBA analyzes transactional data to discover which products are frequently bought together.
- **Goal**: Reveal hidden relationships between items to inform bundling, upselling, and cross-selling strategies

Use Cases & Business Impact

1) **Recommendation Engine**: Suggest complementary products to increase basket size.
2) **Personalized Promotions**: Tailor discounts and bundles based on buying patterns.
3) **Cross-Selling & Upselling**: Identify high-lift product pairs to boost revenue.
4) **Product Strategy**: Inform inventory planning and category expansion.

   Example: A 10% discount on frequently paired items can nudge customers toward higher-value purchases.
   Quote: *"Igniting every cart with smarter, bigger buys."*

5) **Strategic Value**
   - Enhance customer experience through personalization.
   - Enables data-driven decisions for merchandising and marketing.
   - Supports revenue growth via smarter promotions and product bundling.



Market Basket Analysis with Apriori Algorithm

Market Basket Analysis (MBA) is a core component of the project, designed to uncover product associations and drive smarter, personalized shopping experiences. Using the Apriori algorithm, the team mined transactional data to identify frequent item sets and generate actionable association rules

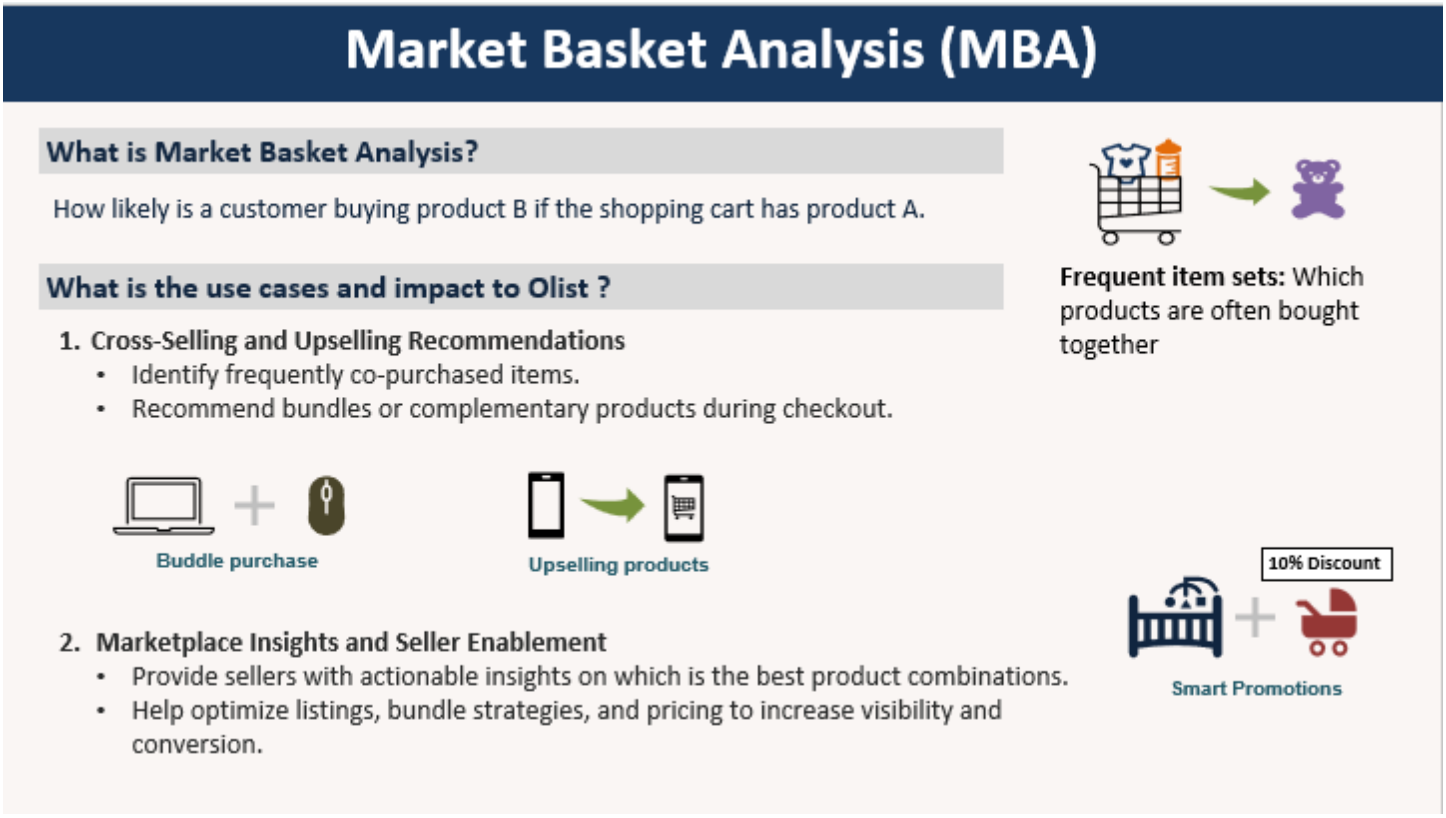
Conceptual Framework

1) Frequent Item Sets

- **Definition**: Groups of products that appear together in transactions more often than a defined threshold.
- **Support**: Measure of how frequently an item set occurs in the dataset.
- Example: Support = 0.10 → Item set appears in 10% of transactions.

2) Association Rules

- **Antecedents → Consequents**: If a customer buys item A, they are likely to buy item B.
- **Confidence**: Probability that item B is purchased when item A is purchased.
  - Example: Confidence = 0.86
- **Lift**: Strength of association beyond random chance.
- Example: Lift = 3.15 → Strong positive correlation between items



Market Basket Analysis (MBA)

How to implement Market Basket Analysis ?

Apriori Algorithm

1) Generate **Frequent Item Sets** in the database
2) Generate **Association Rules** from frequent item sets

What is Association Rules ?

Order Number					
001	✓			✓	1
002	✓	✓			
003	✓		✓	✓	2
004	✓				
005		✓	✓		
006	✓				
007	✓			✓	3
008	✓	✓			
009			✓		
010	✓			✓	4

LHS	RHS	FREQUENCY	SUPPORT	CONFIDENCE	LIFT
Shirts	Pants	4	0.4	0.5	1

Frequency: No of times shirts and pants occurred in the transaction.
= Freq(LHS, RHS) = 4

Support: Frequency of item sets (LHS + RHS) in total transactions.
= Freq (LHS,RHS) / Total Transactions
= 4/10 = 0.4

Confidence: Likelihood of RHS being bought when item LHS is bought.
= Freq (LHS,RHS) / Freq (LHS)
= 4/8 = 0.5

Lift: Strength of association beyond random chance.
= Support / (Support (LHS) x Support (RHS))
= 0.4 / (8/10 x 4/10) = 1.25

MARKET BASKET ANALYSIS

Implementation Workflow

1) Stage 1: Environment Setup

Purpose: Establish a reproducible and traceable analytics workspace.

- Created a new notebook environment in Microsoft Fabric.
- Imported external module mlxtend for Apriori and association rule mining.
- Configure a Fabric experiment to track execution and enable logging for traceability.

2) Stage 2: Load Tables

Purpose: Prepare high-quality input data for association rule mining.

- Pull relevant datasets from the Gold Lakehouse layer.
- Merge multiple tables into a unified transactional view.
- Clean the dataset by removing unnecessary columns.
- Construct a market basket matrix and filter out low-frequency items to reduce noise.

3) Stage 3: Apriori Algorithm

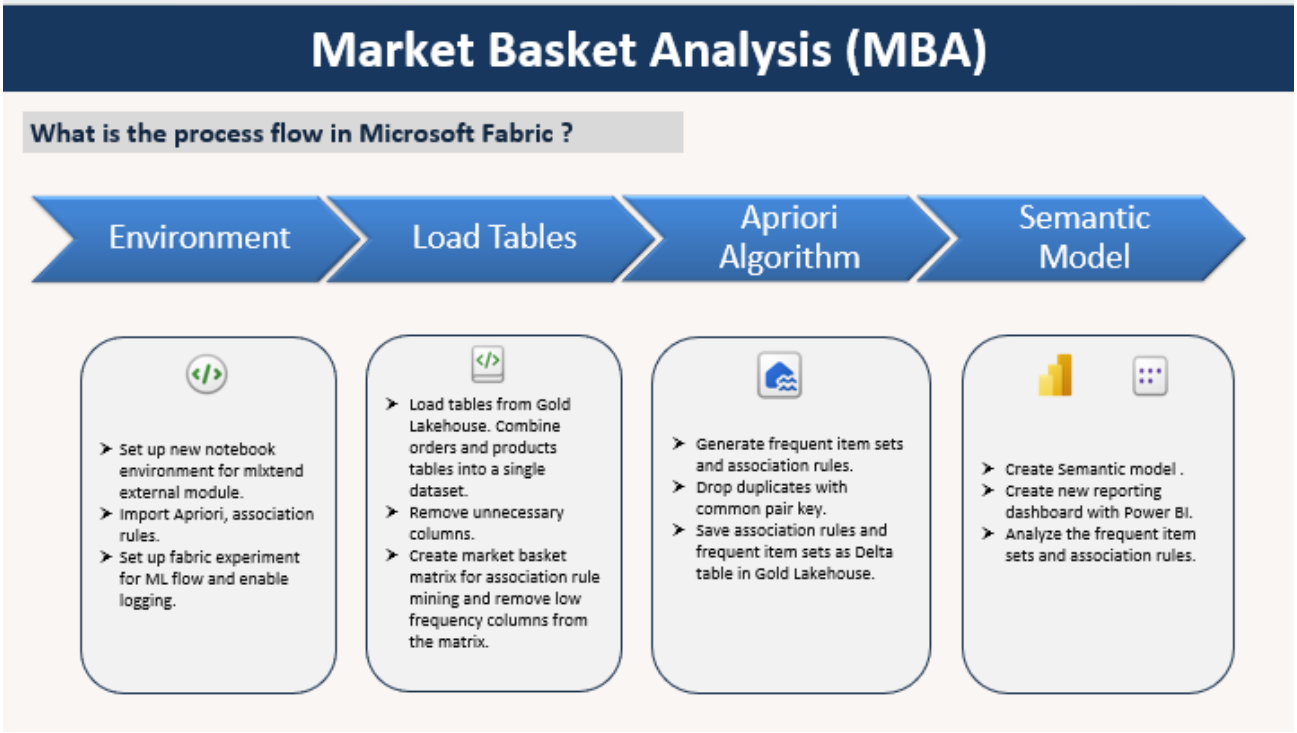
Purpose: Discover actionable product relationships and store them for semantic modeling.

- Apply the Apriori algorithm to extract frequent item sets and generate association rules.
- Persist results as **Delta tables** in the Gold Lakehouse for downstream use.

4) Stage 4: Semantic Model & Reporting

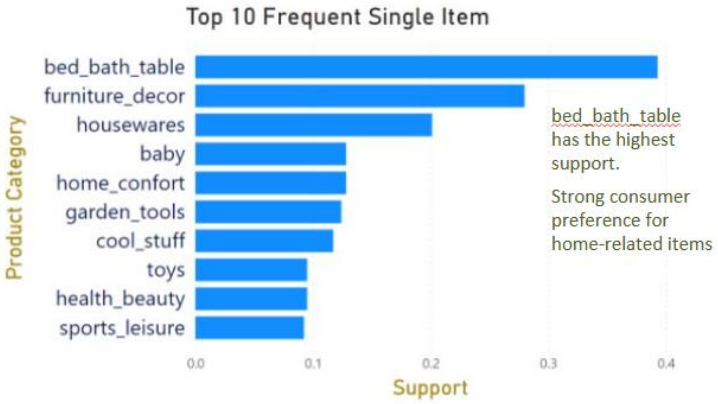
Purpose: Democratize insights and empower data-driven marketing, merchandising, and recommendation strategies.

- Build a semantic model to structure and tag the rules meaningfully.
- Create a Power BI dashboard to visualize frequent item sets and rule metrics (e.g., confidence, lift).
- Enable interactive analysis for business users to explore product affinities and bundling opportunities.



Market Basket Analysis

Frequent Single Item



bed_bath_table has the highest support.

Strong consumer preference for home-related items

IMPLICATIONS

- Categories for cross-selling and bundle promotions.
- Prioritize in home page placement, recommendation engines or email campaigns.

Frequent Item Sets

item_1	item_2	support
bed_bath_table	furniture_decor	0.10
home_confort	bed_bath_table	0.06
housewares	furniture_decor	0.03
cool_stuff	baby	0.03
housewares	bed_bath_table	0.03
toys	baby	0.03
bed_bath_table	baby	0.02
garden_tools	furniture_decor	0.02

bed_bath_table -> 4 out of 8 pairs. It's a Hub item

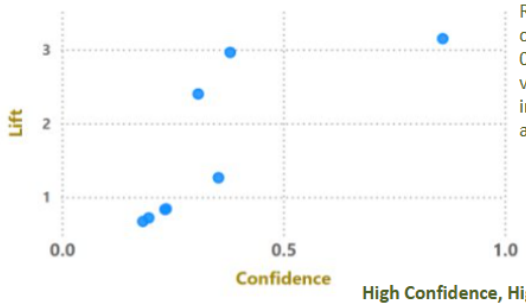
(cool_stuff, baby) and (toys, baby) -> family-oriented purchasing pattern. Target parents or gift buyers.

IMPLICATIONS

- Use these associations to design combo deals or discount bundles.
- Trigger real-time recommendations during check out.
- Inform home page design to group related items.

Market Basket Analysis

Confidence and Lift



Rules cluster in the confidence range of 0.2–0.4, with lift values above 1, indicating meaningful associations.

High Confidence, High Lift represent strong, reliable rules

- Recommendation engines
- Cross-selling strategies
- Personalized promotions

ASSESS THE QUALITY OF RULES

Confidence: Measures how often the rule has been found to be true.
Lift: Indicates how much more likely the consequents is to appear within the antecedent than by random chance.

Association Rules

Association Rules with Lift > 1

antecedents	consequents	confidence	lift	
home_confort	bed_bath_table	0.86	3.15	1
toys	baby	0.38	2.96	2
cool_stuff	baby	0.31	2.40	3
bed_bath_table	furniture_decor	0.35	1.26	4

Rules with **lift greater than 1**, identify item pairings that occur more frequently together than by random chance.

IMPLICATIONS

1. Prioritize pair for bundling, homepage recommendation and inventory planning.
2. Targeted promotions for baby products when toys are purchased.
3. Lower confidence but still a strong lift—this might reflect seasonal or novelty-driven purchases. Consider event-driven campaigns.
4. This rule has the lowest lift indicating a mild positive association. Useful for low-risk cross-selling.

NOTEBOOK SOURCE CODE FOR ML TRANSFORMATION

1) Merge gold_products and gold_product_translations table

1. Load gold_products and gold_product_translations table
2. Remove unnecessary columns from gold_products table
3. Merge both gold_products and gold_product_translations tables
4. The final table contains the product_id and product_category_name in English

```
# Load products and product_translations table
products_df = spark.read.table('gold_products')
translation_df = spark.read.table('gold_product_translations')

# To remove columns that are unnecessary before the merge
products_df = products_df.drop('product_photos_qty','product_weight_g','product_length_cm','product_height_cm','product_width_cm','product_name_length','product_description_length')

# To join both tables to create a new column for english product category name
joined_products_df = products_df.join(
                translation_df,
                products_df['product_category_name'] == translation_df['product_category_name'],
                how='left'
).drop(translation_df['product_category_name'])

# To drop the rows with na
joined_products_df = joined_products_df.na.drop()
```

2) Merge gold_orders and gold_order_items table

1. Load gold_orders and gold_order_items table
2. Drop unnecessary columns from both tables
3. Merge both gold_orders and gold_order_items table

```
# To load the orders, order_reviews and order_items table
orders_df = spark.read.table('gold_orders').alias('o')
order_items_df = spark.read.table("gold_order_items").alias('oi')

# To drop the unnecessary column from the orders table
# orders_df = orders_df.drop('order_estimated_delivery_date','order_delivered_customer_date','order_delivered_carrier_date','order_approved_at')

# To drop the unecessary columns from the order_items table
# order_items_df = order_items_df.drop('seller_id','shipping_limit_date','price','freight_value')

# To rename the order_id column in order_items table as the column name is same as orders table
order_items_df_renamed = order_items_df.withColumnRenamed('order_id','order_items_order_id')

# To merge tables orders and order_items, drop the column for order_items_order_id as it is duplicate with order_id
joined_orders_df = order_items_df_renamed.join(
    orders_df,
    order_items_df_renamed['order_items_order_id'] == orders_df['order_id'],
    how='left'
).drop(order_items_df_renamed['order_items_order_id'])

```

3) Merge orders and products table

1. Rename the product_id of the products table as the column name is duplicated
2. Merge both orders and products table and remove the product_id from the products table
3. Filter rows that have the order status of canceled and unavailable
4. Save the final cleaned table

```
from pyspark.sql.functions import col, to_timestamp

timestamp_cols = ['order_purchase_timestamp','order_approved_at','order_delivered_carrier_date','order_delivered_customer_date','order_estimated_delivery_date','Shipping_limit_date']

# To rename the column name before merge due to duplicated column name
product_cleaned_df_rename = joined_products_df.withColumnRenamed('product_id','product_id_renamed')

# To merge both tables
joined_product_order_df = joined_orders_df.join(
                          product_cleaned_df_rename,
                          product_cleaned_df_rename['product_id_renamed'] == joined_orders_df['product_id'],
                          how='left'
).drop(product_cleaned_df_rename['product_id_renamed'])

# To filter the rows that have the order status of canceled and unavailable
joined_product_order_df = joined_product_order_df.filter(~col('order_status').isin('canceled','unavailable'))

# To convert the data type for order_purchase_timestamp column
for col in timestamp_cols:
    joined_product_order_df = joined_product_order_df.withColumn(col,to_timestamp(col))

# To save the final cleaned table
joined_product_order_df.write.mode('overwrite').saveAsTable('products_orders_cleaned')

```

NOTEBOOK SOURCE CODE FOR ML MARKET BASKET ANALYSIS

Notebook for Olist Market Basket Analysis

About Olist

Founded on July 15, 2015 in Curitiba, Brazil by Tiago Dalvi, Olist was built with a clear mission: to empower small and medium-sized businesses by simplifying their access to major e-commerce marketplaces. Through a unified platform, Olist enables sellers to seamlessly list products, manage orders, and scale their online presence across multiple channels.

Market Basket Analysis

This dataset covers the period from September 2016 to September 2018, offering a snapshot of marketplace activity during Olist's early growth phase. While the timeframe is relatively short, it provides valuable insights into product performance, customer behavior, and operational dynamics.

1) Import required libraries

```
# To import the library from the environment
import mlxtend
```

✓ - Command executed in 297 ms by ChooGeok H (JDE07) on 11:23:37 AM, 10/05/25

```
import mlflow
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm

# Import ML Libraries for Market Basket Analysis
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules

import warnings
warnings.filterwarnings("ignore")
```

✓ - Command executed in 266 ms by ChooGeok H (JDE07) on 11:23:37 AM, 10/05/25

Set up MLflow autolog

Microsoft Fabric extends the MLflow autologging capabilities by automatically capturing the values of input parameters and output metrics of a machine learning model as it's being trained. This information is then logged to the workspace, where it can be accessed and visualized using the MLflow APIs or the corresponding experiment in the workspace.
Experiments in Fabric let you track the results of your AutoML process, providing a comprehensive view of all the metrics and parameters from your trials.

2) Set up experiment tracking for ML flow

```
# Declare experiment name as a variable
# Enable MLflow autologging

EXPERIMENT_NAME = "ML_MarketBasket"
mlflow.set_tracking_uri("file:///lakehouse/default/Files/ML_flow_logs")
mlflow.autolog()
```

✓ - Command executed in 245 ms by ChooGeok H (JDE07) on 11:23:37 AM, 10/05/25 PySpark (Pyt

```
2025/10/05 03:23:37 INFO mlflow.tracking.fluent: Autologging successfully enabled for statsmodels.
```

3) To load the table from LH_Gold Lakehouse and drop columns that are not necessary for Market Basket Analysis

```
# Convert the delta table to pandas dataframe

orders = spark.read.table('products_orders_cleaned')
orders_df = orders.toPandas()

drop_cols = [
            'seller_id',
            'Shipping_limit_date',
            'Price',
            'customer_id',
            'order_status',
            'freight_value',
            'order_approved_at',
            'order_purchase_timestamp',
            'order_delivered_carrier_date',
            'order_delivered_customer_date',
            'order_estimated_delivery_date',
            'product_category_name',
            'purchase_year',
            'purchase_month',
            'product_id'
            ]

orders_df = orders_df.drop(drop_cols,axis=1)
```

MARKET BASKET ANALYSIS

4) To creates a market basket matrix for association rule mining and product affinity analysis

```
# Enables association rule mining and helps identify co-purchase patterns
# For basket analysis encoding the data to 1 and 0, makes the data compatible with mlxtend model
# Enables frequent itemset mining and supports association rule generation

market_basket = pd.pivot_table(data=orders_df,index='order_id',columns='product_category_name_english',values='Order_item_id',fill_value=0)

market_basket_encoded = market_basket.applymap(lambda x: 1 if x >= 1 else 0)
```

✓ - Command executed in 2 sec 297 ms by ChooGeok H (JDE07) on 11:23:43 AM, 10/05/25

5) To remove low-frequency columns from the matrix

```
# To drop columns that have frequency of less than or equal 5 occurrences
# Eliminates single-item orders, which cannot form meaningful associations
# Ensures that association rules (like "If A, then B") are based on co-occurrence
# Improves support and confidence metrics in algorithms like Apriori

low_freq_cols = [col for col in market_basket_encoded.columns if market_basket_encoded[col].sum(skipna=True) <= 5]
market_basket_encoded = market_basket_encoded.drop(low_freq_cols,axis=1)

market_basket_encoded = market_basket_encoded[(market_basket_encoded > 0).sum(axis=1)>=2]
```

✓ - Command executed in 288 ms by ChooGeok H (JDE07) on 11:23:43 AM, 10/05/25

6) Generate frequent itemsets and association rules from your frequent itemsets using the confidence metric

```
# Filters itemsets that appear in at least 2% of all transactions

with mlflow.start_run():
    # Log parameters
    mlflow.log_param("min_support", 0.02)
    mlflow.log_param("metric", "'confidence'")
    mlflow.log_param("min_threshold", 0.1)

    # Run Apriori
    freq_itemset = apriori(market_basket_encoded,min_support=0.02,use_colnames=True)
    # Log metric: number of itemsets
    mlflow.log_metric("num_itemsets", len(freq_itemset))

    # min_threshold of 0.1 to includes rules with confidence greater or equal 10%
    rules_conf = association_rules(freq_itemset,metric='confidence',min_threshold=0.1)
    mlflow.log_metric("num_rules", len(rules_conf))
```

MARKET BASKET ANALYSIS

7) To drop duplicates with common pair_key and convert python sets to string

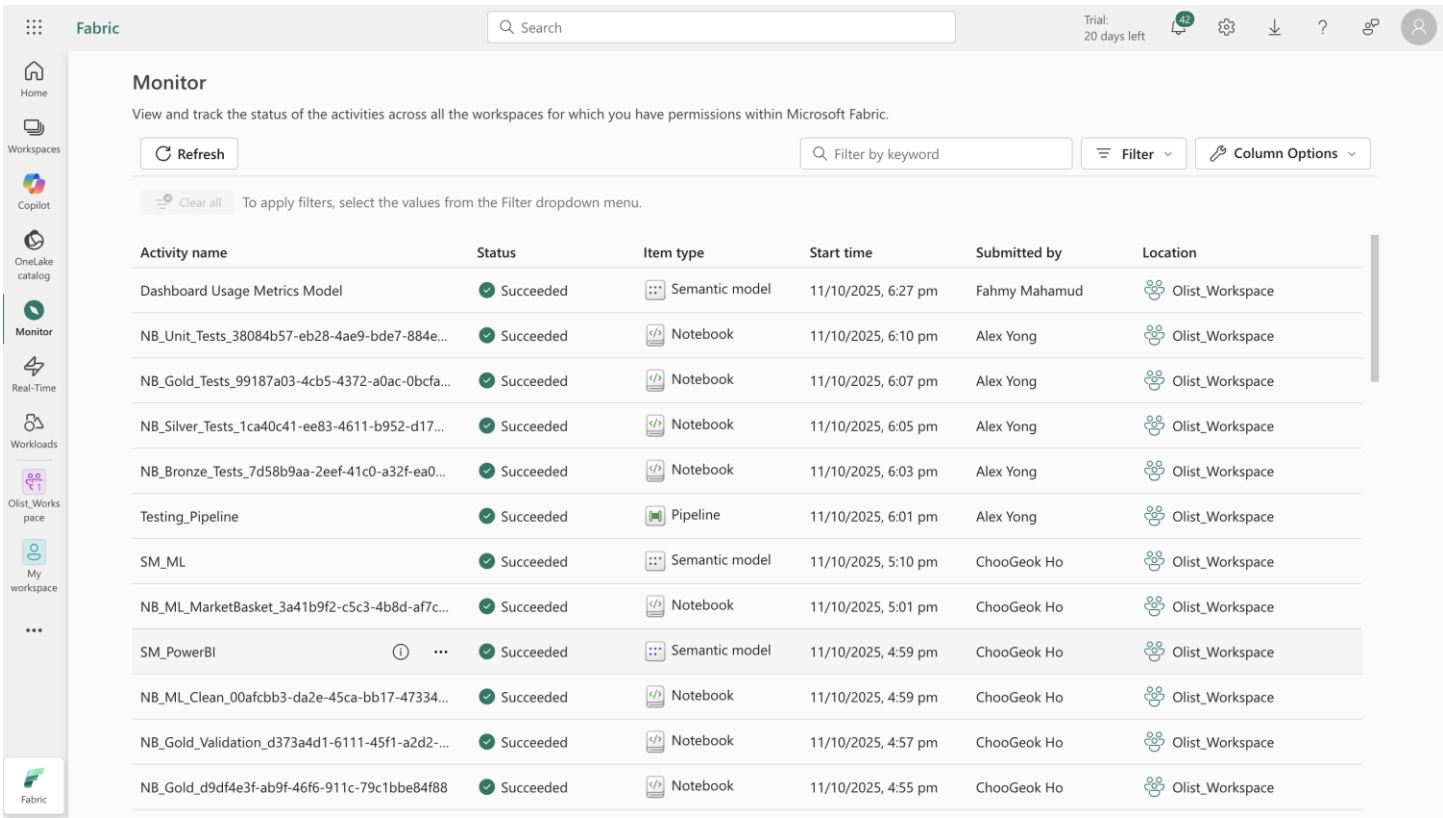
```
# To concatenated both antecedents and consquent column as a new column 'pair_key, to list all the possible combinations to filter the cross pairs
# To drop duplicates rows with common pair_key and to drop the pair_key column

rules_conf_cleaned = rules_conf.copy()

# To convert the python sets to string before saving the rules as spark doesn't natively support python sets
rules_conf_cleaned['antecedents'] = rules_conf_cleaned['antecedents'].apply(lambda x: ', '.join(list(x)))
rules_conf_cleaned['consequents'] = rules_conf_cleaned['consequents'].apply(lambda x: ', '.join(list(x)))

# To rename the columns to remove the space between words
rules_conf_cleaned = rules_conf_cleaned.rename(columns={'antecedent support':'antecedent_support','consequent support':'consequent_support'})

# To save frequent itemset into separate columns and drop the itemsets column which is a python sets
freq_itemset["item_1"] = freq_itemset["itemsets"].apply(lambda x: list(x)[0] if len(x) > 0 else None)
freq_itemset["item_2"] = freq_itemset["itemsets"].apply(lambda x: list(x)[1] if len(x) > 1 else None)
freq_itemset = freq_itemset.drop('itemsets',axis=1)
```

✓ - Command executed in 260 ms by ChooGeok H (JDE07) on 11:23:47 AM, 10/05/25

8) To save association rules and frequent itemsets as Delta Table into the LH_Gold Lakehouse

```
spark.createDataFrame(rules_conf_cleaned).write.format('delta').mode('overwrite').saveAsTable('market_basket_rules')
spark.createDataFrame(freq_itemset).write.format('delta').mode('overwrite').saveAsTable('market_basket_freqitem')
```

MONITORING



Key Capabilities and Information

- The hub allows you to view and track the status of activities across the workspaces you have permission for.
- You can filter the activities by a keyword or use the **Filter** dropdown for advanced options.
- The data can be refreshed using the **Refresh** button.
- It is possible to customize the displayed columns using the **Column Options** menu.

Breakdown of Monitored Activity Fields

The activity log provides the following detailed information for each executed job:

- **Activity name:** This identifies the specific job or item being executed
- **Status:** This indicates the execution result, which is uniformly **Succeeded** for all items visible in the provided log.
- **Item type:** This defines the type of Fabric item, such as a **Notebook**, **Pipeline**, or **Semantic model**.
- **Start time:** This timestamp shows when the activity began
- **Submitted by:** This is the user who started the activity
- **Location:** This specifies the workspace where the item resides, which is Olist_Workspace in all shown examples

Examples of Monitored Project Items

The screenshot displays the monitoring of various data lifecycle stages:

- **Notebooks** are monitored, including unit tests across the data layers: NB_Unit_Tests, NB_Silver_Tests, and NB_Bronze_Tests
- A specific **Pipeline** named Testing_Pipeline is tracked
- **Semantic models** are monitored for refresh status, such as Dashboard Usage Metrics Model, SM_ML, and SM_PowerBI

TESTING

Imports all necessary Python and PySpark modules required for the transformations and testing environment.

```

 # Imports
 import pytest
 from pyspark.sql import SparkSession
 from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType, TimestampType,
 from pyspark.sql.functions import col, to_timestamp, lower, year, month, round as spark_round
 from datetime import datetime
 import re

```

Loads the actual transformation functions, which were presumably extracted from the main pipeline notebooks (NB_Bronze.ipynb, NB_Silver_Transform.ipynb, and NB_Gold.ipynb).

```
# Cell 3: Transformation Functions from Your Notebooks
"""
These are the actual transformation functions extracted from:
- NB_Bronze.ipynb
- NB_Silver_Transform.ipynb
- NB_Gold.ipynb
"""

# ============================================================================
# BRONZE TRANSFORMATIONS
# ============================================================================

def sanitize_table_name(filename: str) -> str:
    """
    From NB_Bronze.ipynb
    Sanitize CSV filename to valid table name
    """
    table_name = re.sub(r"\.csv$", "", filename)
    table_name = re.sub(r"[^a-zA-Z0-9_]", "_", table_name)
    return table_name

```

TESTING

```python
# SILVER TRANSFORMATIONS (extracted from NB_Silver_Transform.ipynb)
# ========================================================================

def transform_orders_timestamps(df):
    """Convert order timestamp columns to timestamp type"""
    return df \
        .withColumn("order_purchase_timestamp", to_timestamp(col("order_purchase_timestamp"))) \
        .withColumn("order_approved_at", to_timestamp(col("order_approved_at"))) \
        .withColumn("order_delivered_carrier_date", to_timestamp(col("order_delivered_carrier_date"))) \
        .withColumn("order_delivered_customer_date", to_timestamp(col("order_delivered_customer_date"))) \
        .withColumn("order_estimated_delivery_date", to_timestamp(col("order_estimated_delivery_date")))

def transform_order_status_lowercase(df):
    """Convert order_status to lowercase"""
    return df.withColumn("order_status", lower(col("order_status")))

def add_purchase_year_month(df):
    """Add purchase_year and purchase_month columns"""
    return df \
        .withColumn("purchase_year", year(col("order_purchase_timestamp"))) \
        .withColumn("purchase_month", month(col("order_purchase_timestamp")))

def remove_order_nulls(df):
    """Remove rows with null critical columns"""
    return df.na.drop(subset=["order_id", "customer_id", "order_purchase_timestamp"])

def transform_order_items(df):
    """Transform order items - timestamps, rounding, casting"""
    return df \
        .withColumn("Shipping_limit_date", to_timestamp(col("Shipping_limit_date"))) \
        .withColumn("Price", spark_round(col("Price").cast("float"), 2)) \
        .withColumn("freight_value", spark_round(col("freight_value").cast("float"), 2)) \
        .withColumn("Order_item_id", col("Order_item_id").cast("integer"))
```

```python
def transform_payments(df):
    """Transform payments - casting and rounding"""
    return df \
        .withColumn("Payment_sequential", col("Payment_sequential").cast("integer")) \
        .withColumn("payment_installments", col("payment_installments").cast("integer")) \
        .withColumn("Payment_value", spark_round(col("Payment_value").cast("float"), 2))

def transform_reviews(df):
    """Transform reviews - timestamps and casting"""
    return df \
        .withColumn("Review_creation_date", to_timestamp(col("Review_creation_date"))) \
        .withColumn("review_answer_timestamp", to_timestamp(col("review_answer_timestamp"))) \
        .withColumn("Review_score", col("Review_score").cast("integer")) \
        .na.drop(subset=["order_id", "Review_score"])

def transform_products(df):
    """Transform products - column renaming"""
    return df \
        .withColumnRenamed("Product_name_lenght", "product_name_length") \
        .withColumnRenamed("product_description_lenght", "product_description_length")
```

TESTING

```python
# ============================================================================
# GOLD TRANSFORMATIONS
# ============================================================================

def gold_table_name(silver_table_name: str) -> str:
    """
    From NB_Gold.ipynb
    Convert silver table name to gold table name
    """
    if silver_table_name.startswith('silver_'):
        return silver_table_name.replace('silver_', 'gold_', 1)
    else:
        return f'gold_{silver_table_name}'

print("✓ Transformation Functions Loaded")
print("=" * 80)

# Cell 4: Unit Tests - Bronze Layer Functions
print("=" * 80)
print("UNIT TESTS - BRONZE LAYER TRANSFORMATIONS")
print("=" * 80)

failed_tests = []
passed_tests = []

# Test 1: sanitize_table_name - basic CSV removal
try:
    result = sanitize_table_name("olist_customers_dataset.csv")
    assert result == "olist_customers_dataset", f"Expected 'olist_customers_dataset', got '{result}'"
    print("✓ PASSED: sanitize_table_name - basic CSV removal")
    passed_tests.append("sanitize_table_name - basic CSV")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
# Test 2: sanitize_table_name - hyphen replacement
try:
    result = sanitize_table_name("product-category.csv")
    assert result == "product_category", f"Expected 'product_category', got '{result}'"
    print("✓ PASSED: sanitize_table_name - hyphen replacement")
    passed_tests.append("sanitize_table_name - hyphen")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 3: sanitize_table_name - space replacement
try:
    result = sanitize_table_name("file with spaces.csv")
    assert result == "file_with_spaces", f"Expected 'file_with_spaces', got '{result}'"
    print("✓ PASSED: sanitize_table_name - space replacement")
    passed_tests.append("sanitize_table_name - spaces")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 4: sanitize_table_name - special characters
try:
    result = sanitize_table_name("special@chars#here.csv")
    assert result == "special_chars_here", f"Expected 'special_chars_here', got '{result}'"
    print("✓ PASSED: sanitize_table_name - special characters")
    passed_tests.append("sanitize_table_name - special chars")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

```

TESTING

```
# Test 5: sanitize_table_name - no extension
try:
    result = sanitize_table_name("table_name")
    assert result == "table_name", f"Expected 'table_name', got '{result}'"
    print("✓ PASSED: sanitize_table_name - no extension")
    passed_tests.append("sanitize_table_name - no extension")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)

# Cell 5: Unit Tests - Silver Order Transformations
print("=" * 80)
print("UNIT TESTS - SILVER ORDERS TRANSFORMATIONS")
print("=" * 80)

spark = SparkSession.builder.getOrCreate()
```

```
# Test 6: Order status lowercase transformation
try:
    test_data = [("ORDER123", "DELIVERED"), ("ORDER124", "PENDING"), ("ORDER125", "CANCELLED")]
    schema = StructType([
        StructField("order_id", StringType(), True),
        StructField("order_status", StringType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = transform_order_status_lowercase(test_df)
    results = [row.order_status for row in result_df.collect()]

    assert results == ["delivered", "pending", "cancelled"], f"Expected lowercase, got {results}"
    print("✓ PASSED: transform_order_status_lowercase")
    passed_tests.append("transform_order_status_lowercase")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 7: Timestamp conversion for orders
try:
    test_data = [("2018-09-04 21:15:22",)]
    schema = StructType([StructField("order_purchase_timestamp", StringType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.withColumn("order_purchase_timestamp", to_timestamp(col("order_purchase_timestamp")

    result_type = str(result_df.schema["order_purchase_timestamp"].dataType)
    assert "timestamp" in result_type.lower(), f"Expected timestamp type, got {result_type}"
    print("✓ PASSED: Timestamp conversion for orders")
    passed_tests.append("Order timestamp conversion")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

```
# Test 8: Add purchase year and month
try:
    test_data = [(datetime(2018, 5, 15, 10, 30, 0),)]
    schema = StructType([StructField("order_purchase_timestamp", TimestampType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = add_purchase_year_month(test_df)

    assert "purchase_year" in result_df.columns, "Missing purchase_year column"
    assert "purchase_month" in result_df.columns, "Missing purchase_month column"

    row = result_df.collect()[0]
    assert row.purchase_year == 2018, f"Expected year 2018, got {row.purchase_year}"
    assert row.purchase_month == 5, f"Expected month 5, got {row.purchase_month}"

    print("✓ PASSED: add_purchase_year_month")
    passed_tests.append("add_purchase_year_month")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

TESTING

```python
# Test 9: Remove order nulls
try:
    test_data = [
        ("ORDER1", "CUST1", datetime(2018, 5, 15)),
        ("ORDER2", None, datetime(2018, 5, 16)),
        (None, "CUST3", datetime(2018, 5, 17)),
        ("ORDER4", "CUST4", datetime(2018, 5, 18))
    ]
    schema = StructType([
        StructField("order_id", StringType(), True),
        StructField("customer_id", StringType(), True),
        StructField("order_purchase_timestamp", TimestampType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = remove_order_nulls(test_df)
    count = result_df.count()

    assert count == 2, f"Expected 2 rows after null removal, got {count}"
    print("✓ PASSED: remove_order_nulls")
    passed_tests.append("remove_order_nulls")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)
```

```python
# Cell 6: Unit Tests - Silver Order Items Transformations
print("=" * 80)
print("UNIT TESTS - SILVER ORDER ITEMS TRANSFORMATIONS")
print("=" * 80)

# Test 10: Order items price rounding
try:
    test_data = [("ITEM1", "12.3456", "5.6789", "1")]
    schema = StructType([
        StructField("order_id", StringType(), True),
        StructField("Price", StringType(), True),
        StructField("freight_value", StringType(), True),
        StructField("Order_item_id", StringType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df \
        .withColumn("Price", spark_round(col("Price").cast("float"), 2)) \
        .withColumn("freight_value", spark_round(col("freight_value").cast("float"), 2))

    row = result_df.collect()[0]
    assert abs(row.Price - 12.35) < 0.01, f"Expected 12.35, got {row.Price}"
    assert abs(row.freight_value - 5.68) < 0.01, f"Expected 5.68, got {row.freight_value}"

    print("✓ PASSED: Order items price rounding")
    passed_tests.append("Order items price rounding")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

TESTING

```python
# Test 11: Order item ID casting to integer
try:
    test_data = [("1",), ("2",), ("3",)]
    schema = StructType([StructField("Order_item_id", StringType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.withColumn("Order_item_id", col("Order_item_id").cast("integer"))

    result_type = str(result_df.schema["Order_item_id"].dataType)
    assert "int" in result_type.lower(), f"Expected integer type, got {result_type}"

    print("✓ PASSED: Order item ID casting to integer")
    passed_tests.append("Order item ID integer cast")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 12: Shipping limit date timestamp conversion
try:
    test_data = [("2018-09-04 21:15:22",)]
    schema = StructType([StructField("Shipping_limit_date", StringType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.withColumn("Shipping_limit_date", to_timestamp(col("Shipping_limit_date")))

    result_type = str(result_df.schema["Shipping_limit_date"].dataType)
    assert "timestamp" in result_type.lower(), f"Expected timestamp, got {result_type}"

    print("✓ PASSED: Shipping limit date timestamp conversion")
    passed_tests.append("Shipping limit timestamp")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)
```

```python
# Cell 7: Unit Tests - Silver Payments Transformations
print("=" * 80)
print("UNIT TESTS - SILVER PAYMENTS TRANSFORMATIONS")
print("=" * 80)

# Test 13: Payment sequential integer casting
try:
    test_data = [("1", "2", "150.50")]
    schema = StructType([
        StructField("Payment_sequential", StringType(), True),
        StructField("payment_installments", StringType(), True),
        StructField("Payment_value", StringType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df \
        .withColumn("Payment_sequential", col("Payment_sequential").cast("integer")) \
        .withColumn("payment_installments", col("payment_installments").cast("integer"))

    seq_type = str(result_df.schema["Payment_sequential"].dataType)
    inst_type = str(result_df.schema["payment_installments"].dataType)

    assert "int" in seq_type.lower(), f"Expected integer for Payment_sequential, got {seq_type}"
    assert "int" in inst_type.lower(), f"Expected integer for payment_installments, got {inst_type}"

    print("✓ PASSED: Payment integer casting")
    passed_tests.append("Payment integer casting")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

TESTING

```python
# Test 14: Payment value rounding
try:
    test_data = [("150.567",)]
    schema = StructType([StructField("Payment_value", StringType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.withColumn("Payment_value", spark_round(col("Payment_value").cast("float"), 2))

    row = result_df.collect()[0]
    assert abs(row.Payment_value - 150.57) < 0.01, f"Expected 150.57, got {row.Payment_value}"

    print("✓ PASSED: Payment value rounding")
    passed_tests.append("Payment value rounding")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)

# Cell 8: Unit Tests - Silver Reviews Transformations
print("=" * 80)
print("UNIT TESTS - SILVER REVIEWS TRANSFORMATIONS")
print("=" * 80)
```

```python
# Test 15: Review timestamp conversion
try:
    test_data = [("2018-01-18 00:00:00", "2018-01-18 21:46:59")]
    schema = StructType([
        StructField("Review_creation_date", StringType(), True),
        StructField("review_answer_timestamp", StringType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df \
        .withColumn("Review_creation_date", to_timestamp(col("Review_creation_date"))) \
        .withColumn("review_answer_timestamp", to_timestamp(col("review_answer_timestamp")))

    create_type = str(result_df.schema["Review_creation_date"].dataType)
    answer_type = str(result_df.schema["review_answer_timestamp"].dataType)

    assert "timestamp" in create_type.lower(), f"Expected timestamp, got {create_type}"
    assert "timestamp" in answer_type.lower(), f"Expected timestamp, got {answer_type}"

    print("✓ PASSED: Review timestamp conversion")
    passed_tests.append("Review timestamp conversion")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

```python
# Test 16: Review score integer casting
try:
    test_data = [("4",), ("5",), ("3",)]
    schema = StructType([StructField("Review_score", StringType(), True)])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.withColumn("Review_score", col("Review_score").cast("integer"))

    result_type = str(result_df.schema["Review_score"].dataType)
    assert "int" in result_type.lower(), f"Expected integer, got {result_type}"

    print("✓ PASSED: Review score integer casting")
    passed_tests.append("Review score integer cast")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))
```

TESTING

```
# Test 17: Review nulls removal
try:
    test_data = [
        ("ORDER1", 4),
        ("ORDER2", None),
        (None, 5),
        ("ORDER4", 3)
    ]
    schema = StructType([
        StructField("order_id", StringType(), True),
        StructField("Review_score", IntegerType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = test_df.na.drop(subset=["order_id", "Review_score"])
    count = result_df.count()

    assert count == 2, f"Expected 2 rows after null removal, got {count}"
    print("✓ PASSED: Review nulls removal")
    passed_tests.append("Review nulls removal")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)

# Cell 9: Unit Tests - Silver Products Transformations
print("=" * 80)
print("UNIT TESTS - SILVER PRODUCTS TRANSFORMATIONS")
print("=" * 80)
```

```
# Test 18: Product column renaming
try:
    test_data = [("PROD1", 50, 200)]
    schema = StructType([
        StructField("product_id", StringType(), True),
        StructField("Product_name_lenght", IntegerType(), True),
        StructField("product_description_lenght", IntegerType(), True)
    ])
    test_df = spark.createDataFrame(test_data, schema)

    result_df = transform_products(test_df)

    assert "product_name_length" in result_df.columns, "Missing product_name_length column"
    assert "product_description_length" in result_df.columns, "Missing product_description_length column"
    assert "Product_name_lenght" not in result_df.columns, "Old column name still exists"

    print("✓ PASSED: Product column renaming")
    passed_tests.append("Product column renaming")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)

# Cell 10: Unit Tests - Gold Layer Transformations
print("=" * 80)
print("UNIT TESTS - GOLD LAYER TRANSFORMATIONS")
print("=" * 80)
```

TESTING

```
# Test 19: Gold table name - with silver_ prefix
try:
    result = gold_table_name("silver_orders")
    assert result == "gold_orders", f"Expected 'gold_orders', got '{result}'"
    print("✓ PASSED: gold_table_name - with silver_ prefix")
    passed_tests.append("gold_table_name - with prefix")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 20: Gold table name - without silver_ prefix
try:
    result = gold_table_name("custom_table")
    assert result == "gold_custom_table", f"Expected 'gold_custom_table', got '{result}'"
    print("✓ PASSED: gold_table_name - without silver_ prefix")
    passed_tests.append("gold_table_name - without prefix")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

# Test 21: Gold table name - already has gold prefix
try:
    result = gold_table_name("gold_orders")
    assert result == "gold_gold_orders", f"Expected 'gold_gold_orders', got '{result}'"
    print("✓ PASSED: gold_table_name - already has gold prefix")
    passed_tests.append("gold_table_name - already gold")
except AssertionError as e:
    print(f"✗ FAILED: {e}")
    failed_tests.append(str(e))

print("\n" + "=" * 80)
```

```
# Cell 11: Summary Report
print("=" * 80)
print("UNIT TEST SUMMARY - ACTUAL PIPELINE TRANSFORMATIONS")
print("=" * 80)
print(f"Total Tests Run: {len(passed_tests) + len(failed_tests)}")
print(f"Tests Passed: {len(passed_tests)}")
print(f"Tests Failed: {len(failed_tests)}")
print("\nCoverage:")
print(f"  Bronze Layer: 5 tests")
print(f"  Silver Orders: 4 tests")
print(f"  Silver Order Items: 3 tests")
print(f"  Silver Payments: 2 tests")
print(f"  Silver Reviews: 3 tests")
print(f"  Silver Products: 1 test")
print(f"  Gold Layer: 3 tests")
print("=" * 80)

if len(failed_tests) > 0:
    print("\nFailed Tests:")
    for test in failed_tests:
        print(f"  - {test}")

# Cell 12: Handle Results
if len(failed_tests) > 0:
    print("\n" + "!" * 80)
    print("UNIT TESTS FAILED")
    print("!" * 80)
    raise Exception(f"Unit tests failed: {len(failed_tests)} test(s) failed")
else:
    print("\n" + "=" * 80)
    print("✓ ALL UNIT TESTS PASSED")
    print(f"All {len(passed_tests)} transformation functions validated")
    print(f"Completed: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print("=" * 80)
```

TESTING

```
================================================================================
✓ PASSED: Payment integer casting
✓ PASSED: Payment value rounding

================================================================================
================================================================================
UNIT TESTS - SILVER REVIEWS TRANSFORMATIONS
================================================================================
✓ PASSED: Review timestamp conversion
✓ PASSED: Review score integer casting
✓ PASSED: Review nulls removal

================================================================================
================================================================================
UNIT TESTS - SILVER PRODUCTS TRANSFORMATIONS
================================================================================
✓ PASSED: Product column renaming

================================================================================
================================================================================
UNIT TESTS - GOLD LAYER TRANSFORMATIONS
================================================================================
✓ PASSED: gold_table_name - with silver_ prefix
✓ PASSED: gold_table_name - without silver_ prefix
✓ PASSED: gold_table_name - already has gold prefix

================================================================================
================================================================================

UNIT TEST SUMMARY - ACTUAL PIPELINE TRANSFORMATIONS
================================================================================
Total Tests Run: 21
Tests Passed: 21
Tests Failed: 0

Coverage:
  Bronze Layer: 5 tests
  Silver Orders: 4 tests
  Silver Order Items: 3 tests
  Silver Payments: 2 tests
  Silver Reviews: 3 tests
  Silver Products: 1 test
  Gold Layer: 3 tests
================================================================================

================================================================================
✓ ALL UNIT TESTS PASSED
All 21 transformation functions validated
Completed: 2025-10-07 05:23:27
================================================================================
```

VALIDATION

VALIDATION OF PIPELINE ACTIVITIES

To ensure data quality, schema integrity, and reliable downstream analytics, the project pipeline incorporates structured validation checkpoints across all three Lakehouse layers—Bronze, Silver, and Gold. Each stage includes notebook-driven validation logic that confirms successful data transfer, schema alignment, and row count consistency.

1) Bronze Lakehouse Validation

- Notebook: NB_Bronze_Validation
- Checks Performed:
  - Confirm raw CSV files are successfully converted to Delta format.
  - Validate that no schema shift occurred during ingestion.
- Purpose: Establish a clean ingestion baseline with schema-on-read flexibility and overwrite mode enforcement.

2) Silver Lakehouse Validation

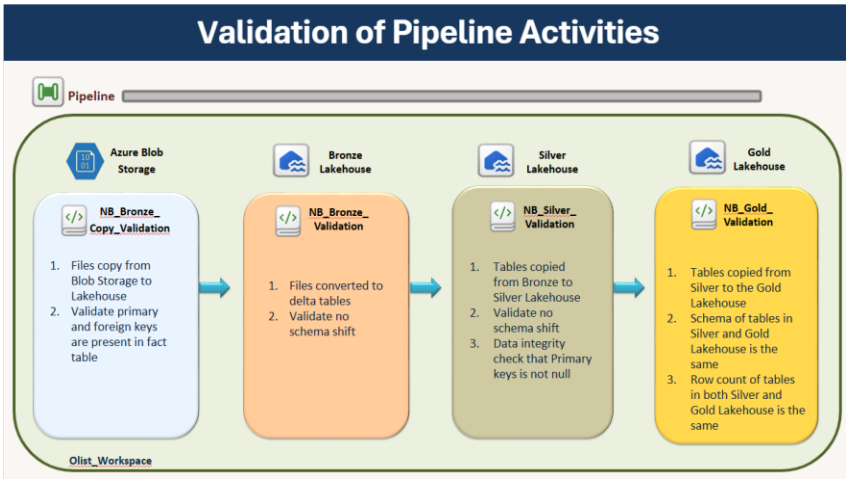
- **Notebook**: NB_Silver_Validation
- **Checks Performed**:
  - Verify that tables are correctly copied from Bronze to Silver.
  - Validate schema consistency between Bronze and Silver layers.
- **Purpose**: Ensure clean and standardized data maintains structural integrity before enrichment and transformation.

3) Gold Lakehouse Validation

- **Notebook**: NB_Gold_Validation
- **Checks Performed**:
  - Confirm successful transfer of tables from Silver to Gold.
  - Validate that the schema in Gold matches the Silver source post-transformation.
  - Reconcile row counts to ensure no data loss during consolidation.
- **Purpose**: Guarantee that the final ML-ready dataset is complete, consistent, and trustworthy for semantic modeling and reporting.

End-to-End Assurance

- Each notebook is integrated into the pipeline flow.
- Failures trigger automated alerts via Outlook, enabling rapid response and root cause analysis.



NOTEBOOK SOURCE CODE FOR COPY DATA FROM AZURE BLOB STORAGE TO BRONZE LAKEHOUSE

Validation for copy of data from Azure Blob Storage to LH_Bronze Lakehouse

```python
# Import libraries
from pyspark.sql.functions import input_file_name
import os
```

✓ - Session ready in 13 sec 660 ms. Command executed in 391 ms by ChooGeok H (JDE07) on 9:00:19 AM, 10/07/25

1) Validate that the expected files have been successfully copied to the LH_Bronze Lakehouse

```python
# Define the expected filenames to be read from the blob storage
file_list = [
            'olist_customers_dataset.csv',
            'olist_geolocation_dataset.csv',
            'olist_order_items_dataset.csv',
            'olist_order_payments_dataset.csv',
            'olist_order_reviews_dataset.csv',
            'olist_orders_dataset.csv',
            'olist_products_dataset.csv',
            'olist_sellers_dataset.csv',
            'product_category_name_translation.csv'
            ]


# Declare the variable for Lakehouse Files folder path
lakehouse_path = "abfss://Olist_Workspace@onelake.dfs.fabric.microsoft.com/LH_Bronze.Lakehouse/Files/"

# Read all CSV files from the Bronze Lakehouse
df = spark.read.option("header", "true").csv(lakehouse_path + "*.csv")

# Extract distinct file names from the Lakehouse Files folder, remove the version with the delimiter of ?
actual_files = [
    os.path.basename(path).split("?")[0]
    for path in df.select(input_file_name()).distinct().rdd.flatMap(lambda x: x).collect()
]

# To compare the distinct set of file names from the actual and declared list
missing_files = list(set(file_list) - set(actual_files))
extra_files = list(set(actual_files) - set(file_list))

if missing_files:
    print(f"Missing files: {missing_files}")
else:
    print("All expected files are present.")

```

VALIDATION

```
35
36  assert not missing_files, f"Missing files in LH_Bronze during the copy operation: {missing_files}"
37  assert not extra_files, f"Extra files in LH_Bronze during the copy operation: {extra_files}"
```

✓ - Command executed in 6 sec 55 ms by ChooGeok H (JDE07) on 9:00:26 AM, 10/07/25

```
All expected files are present.
```

2) Validate that the primary and foreign keys are present in the fact table (orders dataset)

```
1  # Define the primary key and foreign keys in the fact table
2  # Validate the both keys are present in the dataset
3
4  keys = ['order_id','customer_id']
5  df_orders = spark.read.option("header", "true").csv(lakehouse_path + 'olist_orders_dataset.csv')
6  missing_keys = [key for key in keys if key not in df_orders.columns]
7  assert not missing_keys, f"Missing primary and foreign keys in orders table: {missing_keys}"
```

NOTEBOOK SOURCE CODE FOR BRONZE LAKEHOUSE VALIDATION

Validation for Bronze Lakehouse

```
# Import of libraries
import logging
import os
```

1) Validate that the files loaded from the Azure Blob Storage is converted to delta tables and saved in the Bronze Lakehouse

1. Ensures that raw csv files were actually ingested and materialized as Delta tables

2. Prevents silent failures where files are read but not persisted

3. Avoids broken lineage or missing inputs in transformation steps

```
# To compare the filenames of the csv files with the filenames of the delta tables
# To validate that all csv files have been saved as delta tables

# Logging of error messages for monitoring
logging.basicConfig(level=logging.ERROR)

# To list the file path
files = notebookutils.fs.ls("Files/")
expected_tables = []

# To validate and list the csv files are present in the folder
expected_files = [f.name for f in files if f.name.endswith(".csv")]
print("Expected CSV files:", expected_files)

# List actual tables created
actual_tables = [t.name for t in spark.catalog.listTables()]

# To extract the filename with extension
for file in expected_files:
    new_name = os.path.splitext(file)[0]
    expected_tables.append(new_name)

# Compare the list of table name with the list of csv files
missing_tables = [tbl for tbl in expected_tables if tbl not in actual_tables]
if missing_tables:
    print("Missing tables:", missing_tables)
else:
    print("All expected tables loaded successfully.")

```

2) Validate no schema shift and that the schema of the delta tables are the same as the schema declared

1. Guarantees that the structure of ingested data aligns with expectations

2. Prevents mismatches between declared schema and actual data

3. Early detection of schema shift avoids cascading failures in Silver and Gold layers

```
# Read the file using Spark
df = spark.read.text("abfss://Olist_Workspace@onelake.dfs.fabric.microsoft.com/LH_Bronze.Lakehouse/Files/bronze_schemas.py")

# Convert to string and execute
schema_code = "\n".join(row['value'] for row in df.collect())
exec(schema_code)

for table_name, expected_schema in schemas.items():
    try:
        actual_schema = spark.table(f"LH_Bronze.{table_name}").schema
        assert actual_schema == expected_schema, f"Schema mismatch for {table_name}\n"

    except Exception as e:
        print(f"Error validating schema for {table_name}: {e}")
        raise Exception("Schema shifted for {table_name} in Bronze Lakehouse:{e}")
```

SCHEMA FOR BRONZE LAKEHOUSE

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType, DateType, TimestampType

schemas = {
    'silver_customers' : StructType([
                        StructField('customer_id', StringType(), True),
                        StructField('customer_unique_id', StringType(), True),
                        StructField('customer_zip_code_prefix', StringType(), True),
                        StructField('customer_city', StringType(), True),
                        StructField('customer_state', StringType(), True)
                        ]),
    'silver_geolocation' : StructType([
                           StructField('geolocation_zip_code_prefix', StringType(), True),
                           StructField('geolocation_lat', StringType(), True),
                           StructField('geolocation_lng', StringType(), True),
                           StructField('geolocation_city', StringType(), True),
                           StructField('geolocation_state', StringType(), True)
                           ]),

    'silver_order_items' : StructType([
                           StructField('order_id', StringType(), True),
                           StructField('Order_item_id', IntegerType(), True),
                           StructField('product_id', StringType(), True),
                           StructField('seller_id', StringType(), True),
                           StructField('Shipping_limit_date', TimestampType(), True),
                           StructField('Price', FloatType(), True),
                           StructField('freight_value', FloatType(), True)
                           ]),
    'silver_order_payments' : StructType([
                           StructField('order_id', StringType(), True),
                           StructField('Payment_sequential', IntegerType(), True),
                           StructField('payment_type', StringType(), True),
                           StructField('payment_installments', IntegerType(), True),
                           StructField('Payment_value', FloatType(), True)
                           ]),
    'silver_order_reviews' : StructType([
                           StructField('order_id', StringType(), True),
                           StructField('Payment_sequential', IntegerType(), True),
                           StructField('payment_type', StringType(), True),
                           StructField('payment_installments', IntegerType(), True),
                           StructField('Payment_value', FloatType(), True)
                           ]),
```

VALIDATION

```
    'silver_orders' : StructType([StructField('order_id', StringType(), True),
                        StructField('customer_id', StringType(), True),
                        StructField('order_status', StringType(), True),
                        StructField('order_purchase_timestamp', TimestampType(), True),
                        StructField('order_approved_at', TimestampType(), True),
                        StructField('order_delivered_carrier_date', TimestampType(), True),
                        StructField('order_delivered_customer_date', TimestampType(), True),
                        StructField('order_estimated_delivery_date', TimestampType(), True),
                        StructField('purchase_year', IntegerType(), True),
                        StructField('purchase_month', IntegerType(), True)
                    ]),
    'silver_products' : StructType([
                        StructField('product_id', StringType(), True),
                        StructField('product_category_name', StringType(), True),
                        StructField('product_name_length', StringType(), True),
                        StructField('product_description_length', StringType(), True),
                        StructField('product_photos_qty', StringType(), True),
                        StructField('product_weight_g', StringType(), True),
                        StructField('product_length_cm', StringType(), True),
                        StructField('product_height_cm', StringType(), True),
                        StructField('product_width_cm', StringType(), True)
                        ]),
   'silver_sellers' : StructType([
                            StructField('seller_id', StringType(), True),
                            StructField('seller_zip_code_prefix', StringType(), True),
                            StructField('seller_city', StringType(), True),
                            StructField('seller_state', StringType(), True)
                            ]),
    'silver_product_translations' : StructType([
                            StructField('product_category_name', StringType(), True),
                            StructField('product_category_name_english', StringType(), True)
                            ])
}
```

NOTEBOOK SOURCE CODE FOR SILVER LAKEHOUSE VALIDATION

Validation for Silver Lakehouse

1) Validate that the tables in LH_Bronze Lakehouse have been successfully copied to the LH_Silver Lakehouse

1. Ensures no tables were lost, duplicated, or corrupted during the copy
2. Detects silent failures or partial writes due to timeouts, memory issues, or schema drift
3. Validates that the pipeline executed all steps as expected

```
# To load the table names in both LH_Silver and LH_Bronze
# Compare the table names present in LH_Bronze with the table names in LH_Silver
# Assert that there is no missing tables between both Lakehouse after running of the notebook

tables_name_bronze = []
tables_name_silver = []

bronze_tables = [t.name for t in spark.catalog.listTables('LH_Bronze')]
silver_tables = [t.name for t in spark.catalog.listTables('LH_Silver')]
tables_name_bronze = [t.removeprefix('olist_').removesuffix('_dataset') for t in bronze_tables]
tables_name_silver = [t.removeprefix('silver_') for t in silver_tables]
tables_name_silver = ['product_category_name_translation' if t == 'product_translations' else t for t in tables_name_silver]
missing_tables = [t for t in tables_name_silver if t not in tables_name_silver]
assert not missing_tables, f"Missing tables in LH_Silver: {missing_tables}"
```

✓ - Session ready in 13 sec 430 ms. Command executed in 47 sec 45 ms by ChooGeok H (JDE07) on 9:43:21 AM, 10/06/25

2) Validate no schema shift and that the schema of the delta tables are the same as the schema declared

1. Guarantees that the structure of ingested data aligns with expectations
2. Prevents mismatches between declared schema and actual data
3. Early detection of schema shift avoids cascading failures in Gold layers

```
# To read the schema file from the LH_Silver file folders
df = spark.read.text("abfss://Olist_Workspace@onelake.dfs.fabric.microsoft.com/LH_Silver.Lakehouse/Files/silver_schemas.py")

# Convert to string and execute to load the schema
schema_code = "\n".join(row['value'] for row in df.collect())
exec(schema_code)

```

```
# To extract the table name and schema
for table_name, expected_schema in schemas.items():
    try:
        actual_schema = spark.table(f"LH_Silver.{table_name}").schema
        assert actual_schema == expected_schema, f"Schema mismatch for {table_name}\n"

    except Exception as e:
        print(f"Error validating schema for {table_name}: {e}")
        raise Exception(f"Schema shifted for {table_name} in Silver Lakehouse:{e}")
```

✓ - Command executed in 16 sec 96 ms by ChooGeok H (JDE07) on 1:48:14 PM, 10/01/25

3) Row-Level Integrity Checks

1. Ensure critical fields are populated and Null checks on primary keys in the fact table

```
# To validate that the primary key of the fact table is not null
from pyspark.sql.functions import col
orders = spark.read.table(f'LH_Silver.silver_orders')
assert orders.filter(col("order_id").isNull()).count() == 0, "Missing order_id"
```

SCHEMA FOR SILVER LAKEHOUSE

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType,FloatType

schemas = {
    'silver_customers' : StructType([
                        StructField('customer_id', StringType(), True),
                        StructField('customer_unique_id', StringType(), True),
                        StructField('customer_zip_code_prefix', StringType(), True),
                        StructField('customer_city', StringType(), True),
                        StructField('customer_state', StringType(), True)
                        ]),
    'silver_geolocation' : StructType([
                        StructField('geolocation_zip_code_prefix', StringType(), True),
                        StructField('geolocation_lat', StringType(), True),
                        StructField('geolocation_lng', StringType(), True),
                        StructField('geolocation_city', StringType(), True),
                        StructField('geolocation_state', StringType(), True)
                        ]),
    'silver_order_items' : StructType([
                        StructField('order_id', StringType(), True),
                        StructField('Order_item_id', IntegerType(), True),
                        StructField('product_id', StringType(), True),
                        StructField('seller_id', StringType(), True),
                        StructField('Shipping_limit_date', TimestampType(), True),
                        StructField('Price', FloatType(), True),
                        StructField('freight_value', FloatType(), True)
                        ]),
    'silver_order_payments' : StructType([
                        StructField('order_id', StringType(), True),
                        StructField('Payment_sequential', IntegerType(), True),
                        StructField('payment_type', StringType(), True),
                        StructField('payment_installments', IntegerType(), True),
                        StructField('Payment_value', FloatType(), True)
                        ]),
    'silver_order_reviews' : StructType([
                        StructField('review_id', StringType(), True),
                        StructField('order_id', StringType(), True),
                        StructField('Review_score', IntegerType(), True),
                        StructField('review_comment_title', StringType(), True),
                        StructField('review_comment_message', StringType(), True),
                        StructField('Review_creation_date', TimestampType(), True),
                        StructField('review_answer_timestamp', TimestampType(), True)
                        ]),
```

VALIDATION

```
    'silver_orders' : StructType([
                        StructField('order_id', StringType(), True),
                        StructField('customer_id', StringType(), True),
                        StructField('order_status', StringType(), True),
                        StructField('order_purchase_timestamp', TimestampType(), True),
                        StructField('order_approved_at', TimestampType(), True),
                        StructField('order_delivered_carrier_date', TimestampType(), True),
                        StructField('order_delivered_customer_date', TimestampType(), True),
                        StructField('order_estimated_delivery_date', TimestampType(), True),
                        StructField('purchase_year', IntegerType(), True),
                        StructField('purchase_month', IntegerType(), True)
                        ]),
    'silver_products' : StructType([
                        StructField('product_id', StringType(), True),
                        StructField('product_category_name', StringType(), True),
                        StructField('product_name_length', StringType(), True),
                        StructField('product_description_length', StringType(), True),
                        StructField('product_photos_qty', StringType(), True),
                        StructField('product_weight_g', StringType(), True),
                        StructField('product_length_cm', StringType(), True),
                        StructField('product_height_cm', StringType(), True),
                        StructField('product_width_cm', StringType(), True)
                        ]),
   'silver_sellers' : StructType([
                        StructField('seller_id', StringType(), True),
                        StructField('seller_zip_code_prefix', StringType(), True),
                        StructField('seller_city', StringType(), True),
                        StructField('seller_state', StringType(), True)
                        ]),
    'silver_product_translations' : StructType([
                        StructField('product_category_name', StringType(), True),
                        StructField('product_category_name_english', StringType(), True)
                        ])
}
```

NOTEBOOK SOURCE CODE FOR GOLD LAKEHOUSE VALIDATION

Validation for Gold Lakehouse

1) Validate that the tables in LH_Silver Lakehouse have been successfully copied to the LH_Gold Lakehouse

```
# To load the table names in both LH_Silver and LH_Gold
# Compare the table names present in LH_Silver with the table names in LH_Gold
# Assert that there is no missing tables between both Lakehouse after running of the notebook

tables_name_silver = []
tables_name_gold = []

silver_tables = [t.name for t in spark.catalog.listTables('LH_Silver')]
gold_tables_all = [t.name for t in spark.catalog.listTables('LH_Gold')]
gold_tables = [name for name in gold_tables_all if name.startswith('gold_')]

tables_name_silver = [t.removeprefix('silver_') for t in silver_tables]
tables_name_gold = [t.removeprefix('gold_') for t in gold_tables]

missing_tables = [t for t in tables_name_silver if t not in tables_name_gold]
assert not missing_tables, f"Missing tables in LH_Gold: {missing_tables}"
```

✓ - Command executed in 9 sec 783 ms by ChooGeok H (JDE07) on 3:32:34 PM, 9/30/25

+ Code + Markdown

2) Validate that the schema of tables in both LH_Silver and LH_Gold Lakehouse is the same

```
# To extract the schema (column names, types) from both LH_Silver and LH_Gold Lakehouse
# Assert that the schemas in both Lakehouse is the same

for table_name in tables_name_silver:
    df_silver_schema = spark.table(f"LH_Silver.silver_{table_name}").schema
    df_gold_schema = spark.table(f"LH_Gold.gold_{table_name}").schema
    assert df_silver_schema == df_gold_schema
```

3) Validate that the row count of tables in both LH_Silver and LH_Gold Lakehouse is the same

```
# To ensure row-level fidelity with row count comparison
# Assert that the number of rows in the same table name of both Lakehouse is the same

for table_name in tables_name_silver:
    df_silver = spark.table(f"LH_Silver.silver_{table_name}")
    df_gold = spark.table(f"LH_Gold.gold_{table_name}")
    assert df_silver.count() == df_gold.count(), f"Row count mismatch in {table_name}"
```

✓ - Command executed in 7 sec 933 ms by ChooGeok H (JDE07) on 3:32:45 PM, 9/30/25

VALIDATION OF PIPELINE FLOW

To ensure transparency, reliability, and rapid response across the project pipeline, a structured validation framework was implemented alongside automated email notifications. Each Lakehouse layer—Bronze, Silver, and Gold—undergoes rigorous validation, and successful execution triggers targeted alerts to the data engineering team

Layer-by-Layer Validation Summary

1) Bronze Lakehouse

Validation Steps:

- Bronze_DeltaTable: Failed
- Bronze_CopyBlob: Succeeded

Activity:

Email_Notification_Bronze is triggered when the Bronze_DeltaTable fails.

2) Silver Lakehouse

Validation Steps:

- Silver_Cleaning: Failed
- Bronze_Validation: Succeeded
- Bronze_DeltaTable: Succeeded
- Bronze_CopyBlob: Succeeded

Activity:

Email_Notification_Silver is triggered when the Silver_Cleaning fails.

3) Gold Lakehouse

Validation Steps:

- Gold_Transform: Failed
- Silver_Validation: Succeeded
- Silver_Cleaning: Succeeded
- Bronze_Validation: Succeeded
- Bronze_DeltaTable: Succeeded
- Bronze_CopyBlob: Succeeded

Activity:

Email_Notification_Gold is triggered when the Gold_Transform fails.

4) Semantic Model Refresh

Validation Steps:

- ML_Semantic_model: Succeeded
- Gold_Validation: Succeeded
- Gold_Transform: Succeeded
- Silver_Validation: Succeeded
- Silver_Cleaning: Succeeded
- Bronze_Validation: Succeeded
- Bronze_DeltaTable: Succeeded
- Bronze_CopyBlob: Succeeded

Activity:

New_Report_Notification is triggered when the whole pipeline runs is completed.

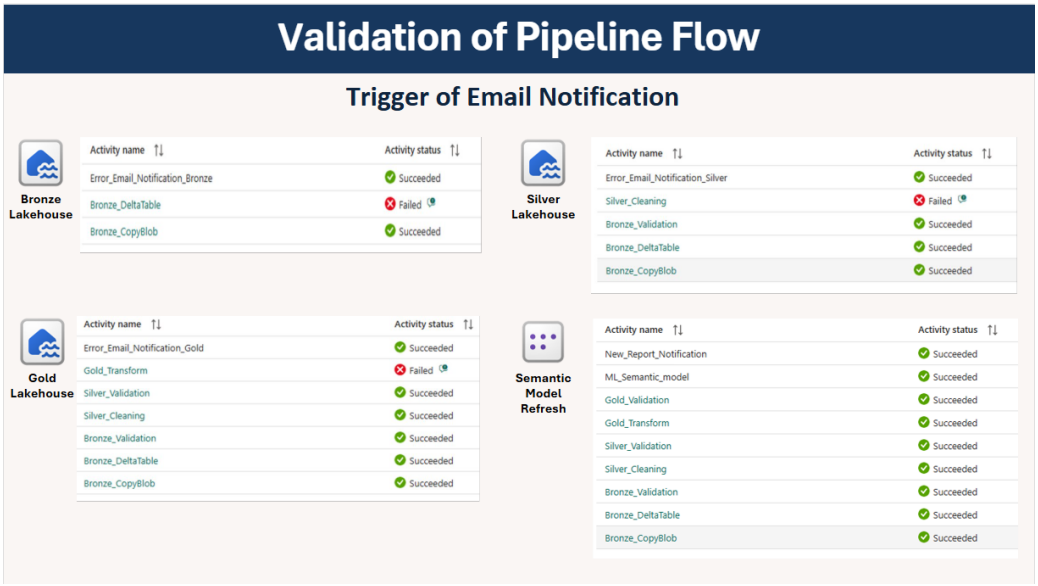
VALIDATION

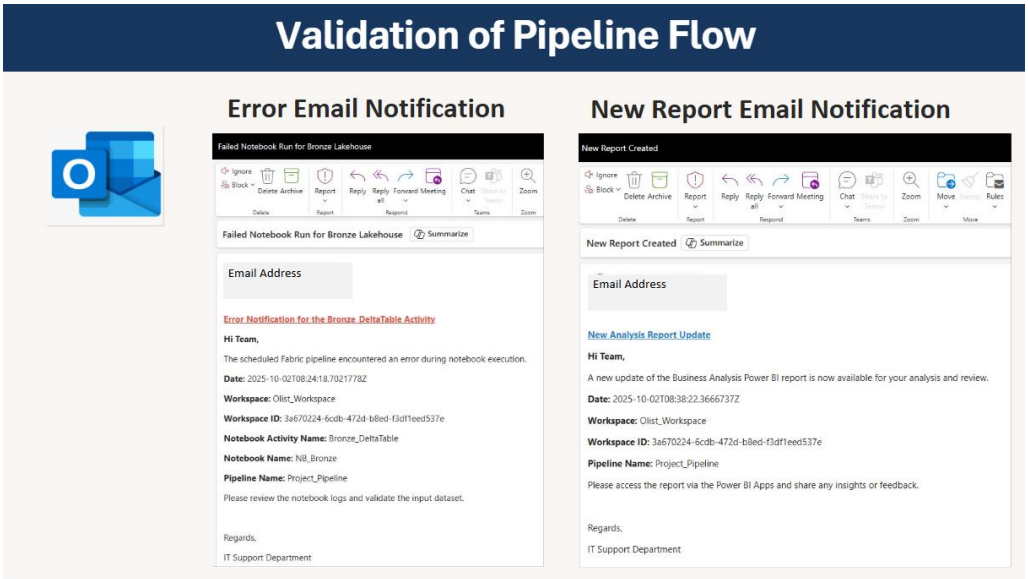
Email Notification Logic

- Each validation notebook is linked to a corresponding notification trigger.
- Upon failed execution of the notebook, an email is sent via Outlook to the designated recipients.
- The email includes:
    - Pipeline stage (Bronze, Silver, Gold, Semantic Model)
    - Execution status
    - Timestamp and activity name
- This ensures the teams involved are informed in real time and can act immediately if any stage fails.

Benefits

- **Operational Transparency**: Clear visibility into pipeline health across all layers.
- **Rapid Response**: Immediate alerts enable faster troubleshooting and resolution.
- **Auditability**: Logged validation outcomes support governance and historical analysis.





Project Deliverable

PROJECT PIPELINE OVERVIEW

The project pipeline is a modular, orchestrated system that spans ingestion, validation, analytics, and alerting. Built on Microsoft Fabric, it integrates notebooks, semantic modeling, and Power BI dashboards with automated email notifications for robust observability and stakeholder communication

Medallion Layer

The Medallion Layer structures the data lifecycle into Bronze, Silver, and Gold stages, enabling progressive refinement from raw ingestion to enrich, analytics-ready datasets.

- Bronze_DeltaTable: Convert CSV files into Delta Table and save them in Bronze Lakehouse.
- Silver_Cleaning: Applies type casting, null handling, and enrichment logic.
- Gold_Transform: Copy delta tables from Silver Lakehouse into Gold Lakehouse and perform transformation.

Validation Layer

This foundational layer ensures data integrity and schema consistency across the Bronze, Silver, and Gold Lakehouse stages**.**

- Bronze_Copy_Validation: Initiates data ingestion from source files.
- Bronze_Validation: Confirms successful conversion to Delta format and schema alignment.
- Silver_Validation: Validates schema and row count consistency between Bronze and Silver.
- Gold_Validation: Ensures schema and data integrity between Silver and Gold.

Market Basket Analysis Layer

This layer applies Apriori-based association rule mining to uncover product affinities and drive personalized recommendations.

- ML_Transform: Consolidates cleaned data into a unified schema for ML and reporting.
- ML_Market_Basket_Analysis: Executes the Apriori algorithm using mlxtend within a Fabric notebook. Generates frequent item sets and association rules.
- ML_Semantic_Model: Refresh of data for dashboard.
- ML_Report_Notification: Sends email notifications with insights and rule summaries to stakeholders.

Power BI Analysis Layer

This layer delivers curated dashboards for leadership decision support.

- ML_Semantic_Model: Refresh of data for Power BI analysis report.
- PowerBI_Report_Notification: Triggers email alerts upon successful semantic model refresh and dashboard updates.

Error Handling Layer

This layer ensures proactive alerting and rapid response to pipeline failures.

- **Outlook**: Automatically send error notifications when validation or transformation notebooks fail. Emails include pipeline stage, error context, and timestamp for root cause analysis.

Architectural Highlights

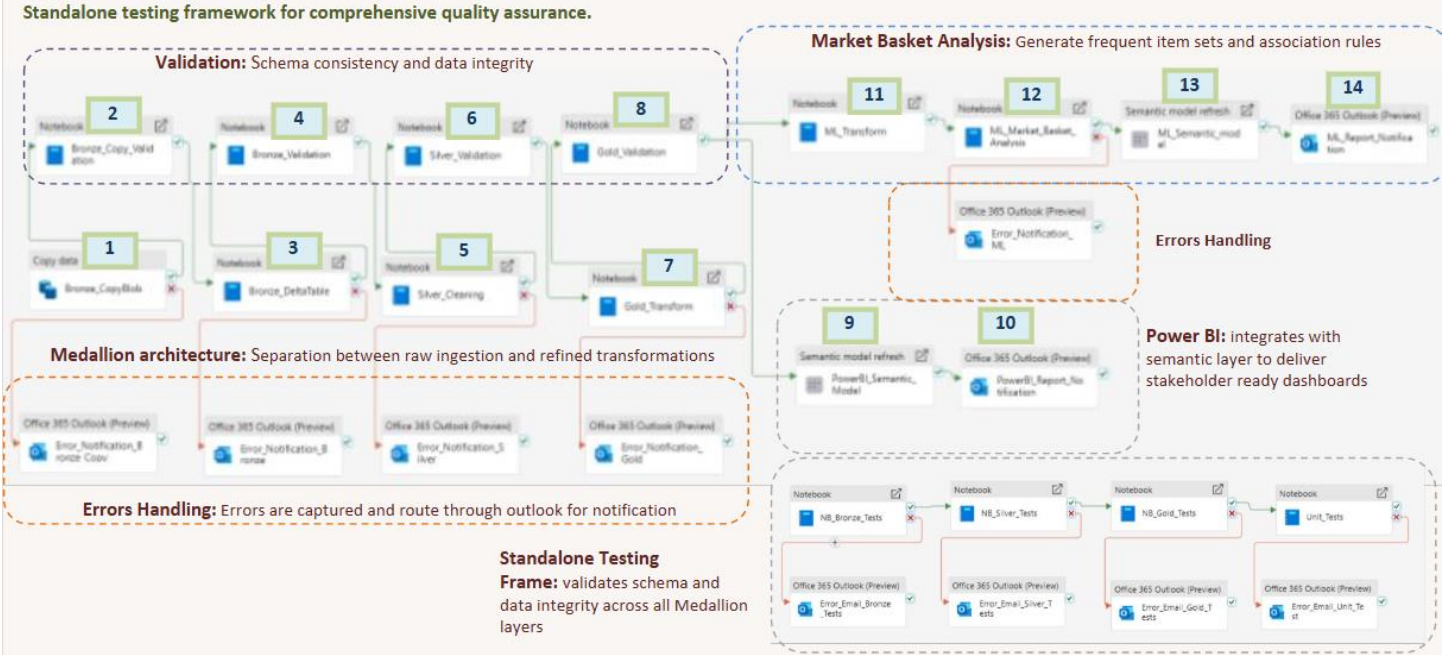
- **Modular Notebook Design**: Each task is encapsulated in reusable notebooks for validation, transformation, and ML logic.
- **Integrated Alerting**: Outlook notifications are triggered at key checkpoints to ensure transparency and accountability.
- **Semantic Modeling**: Power BI semantic layer supports drill-through analytics and stakeholder-friendly reporting.
- **End-to-End Traceability**: Logs and validation results are stored in Delta tables and visualized via monitoring dashboards.

Screen shot of Project Pipeline

PROJECT DELIVERABLE

PROJECT PIPELINE

End-to-end pipeline from validation to reporting, engineered with a robust, modular, and scalable architecture.
It spans the full lifecycle from Medallion-based transformations.
Standalone testing framework for comprehensive quality assurance.



Screen shot of Project Pipeline Status

PROJECT PIPELINE

Screenshot showing the pipeline status with all activities marked as succeeded

Parameters | Variables | Settings | **Output** | Library variables

Pipeline run ID 764453da-711d-4212-b2d5-1a6d112f8abf | View run detail | **Pipeline status** Succeeded

Filter by keyword | Showing 1 - 14 items

Activity name	Activity status	Run start
ML_Report_Notification	Succeeded	10/8/2025, 11:43:09 AM
ML_Semantic_model	Succeeded	10/8/2025, 11:42:25 AM
ML_Market_Basket_Analysis	Succeeded	10/8/2025, 11:33:53 AM
PowerBI_Report_Notification	Succeeded	10/8/2025, 11:33:04 AM
ML_Transform	Succeeded	10/8/2025, 11:32:23 AM
PowerBI_Semantic_Model	Succeeded	10/8/2025, 11:32:23 AM
Gold_Validation	Succeeded	10/8/2025, 11:30:37 AM
Gold_Transform	Succeeded	10/8/2025, 11:28:35 AM
Silver_Validation	Succeeded	10/8/2025, 11:27:23 AM
Silver_Cleaning	Succeeded	10/8/2025, 11:25:09 AM
Bronze_Validation	Succeeded	10/8/2025, 11:24:13 AM
Bronze_DeltaTable	Succeeded	10/8/2025, 11:21:58 AM
Bronze_Copy_Validation	Succeeded	10/8/2025, 11:21:12 AM
Bronze_CopyBlob	Succeeded	10/8/2025, 11:20:49 AM

VISUALIZATION AND REPORTING

Setting the context right…

Persona 4: Matthew Taylor – Strategy Director

As Strategy Director, Matthew 's role is to ***drive marketplace growth, profitability, and customer trust***. He needs a single, consolidated view *of* ***business performance across sales, customer retention, delivery efficiency, and seller reliability.***

Problem Statement

- Leadership **lacks a consolidated view** of sales, customer retention, delivery, and seller performance on one page.

Project Objective

- To deliver an **executive-level summary** that brings together all marketplace KPIs into a single, intuitive dashboard.

1. **Are we growing in the right places?**

"As Strategy Director, I need to quickly see whether our marketplace is expanding in the right categories and regions — not just topline revenue."

2. **Are customers staying with us?**

"Our growth is hollow if customers don't return. I need clarity on churn vs. loyalty."

3. **Are our goods delivered timely as promise?**

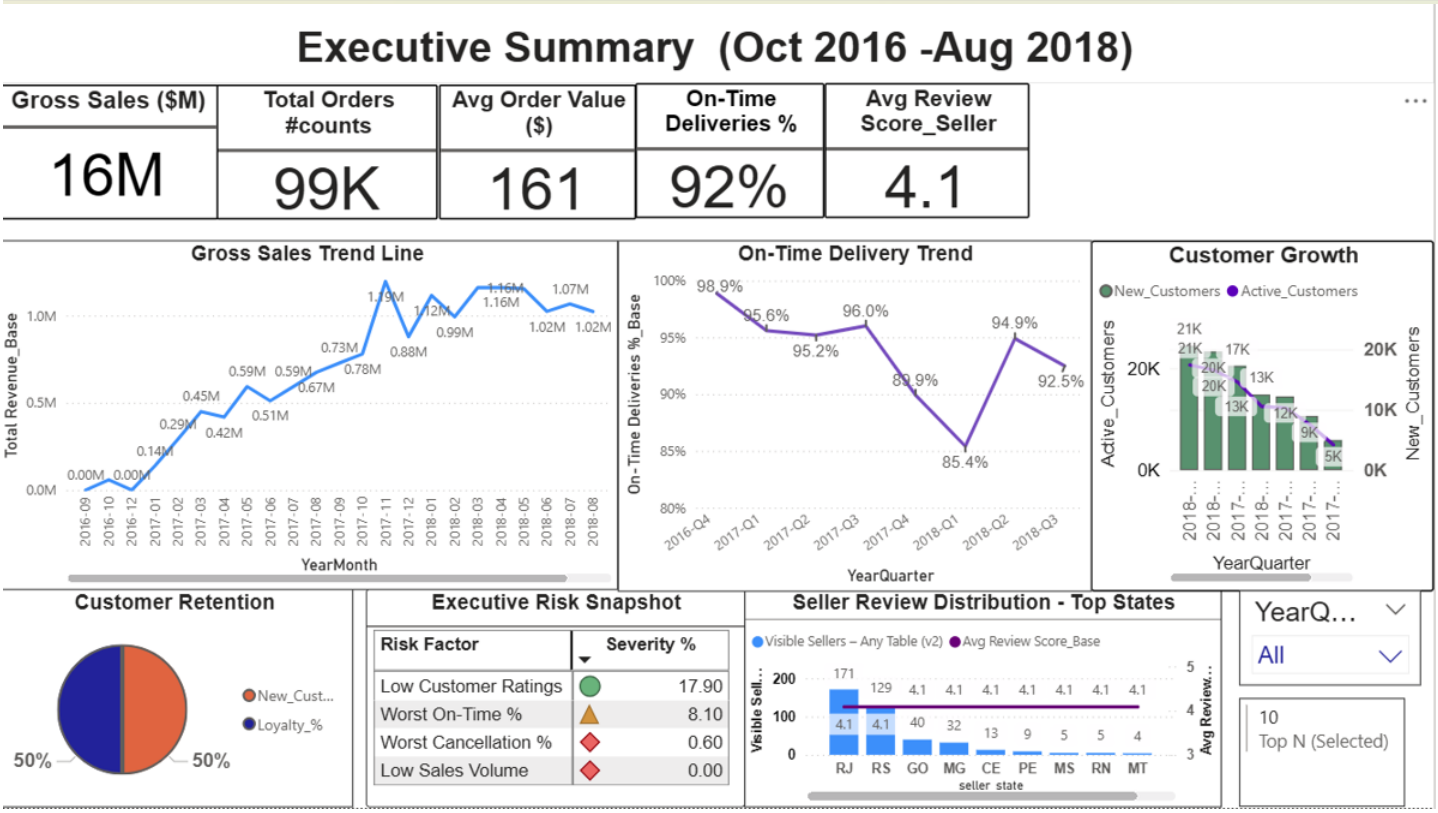
"Delays erode trust fast. I want an early-warning system on delivery performance."

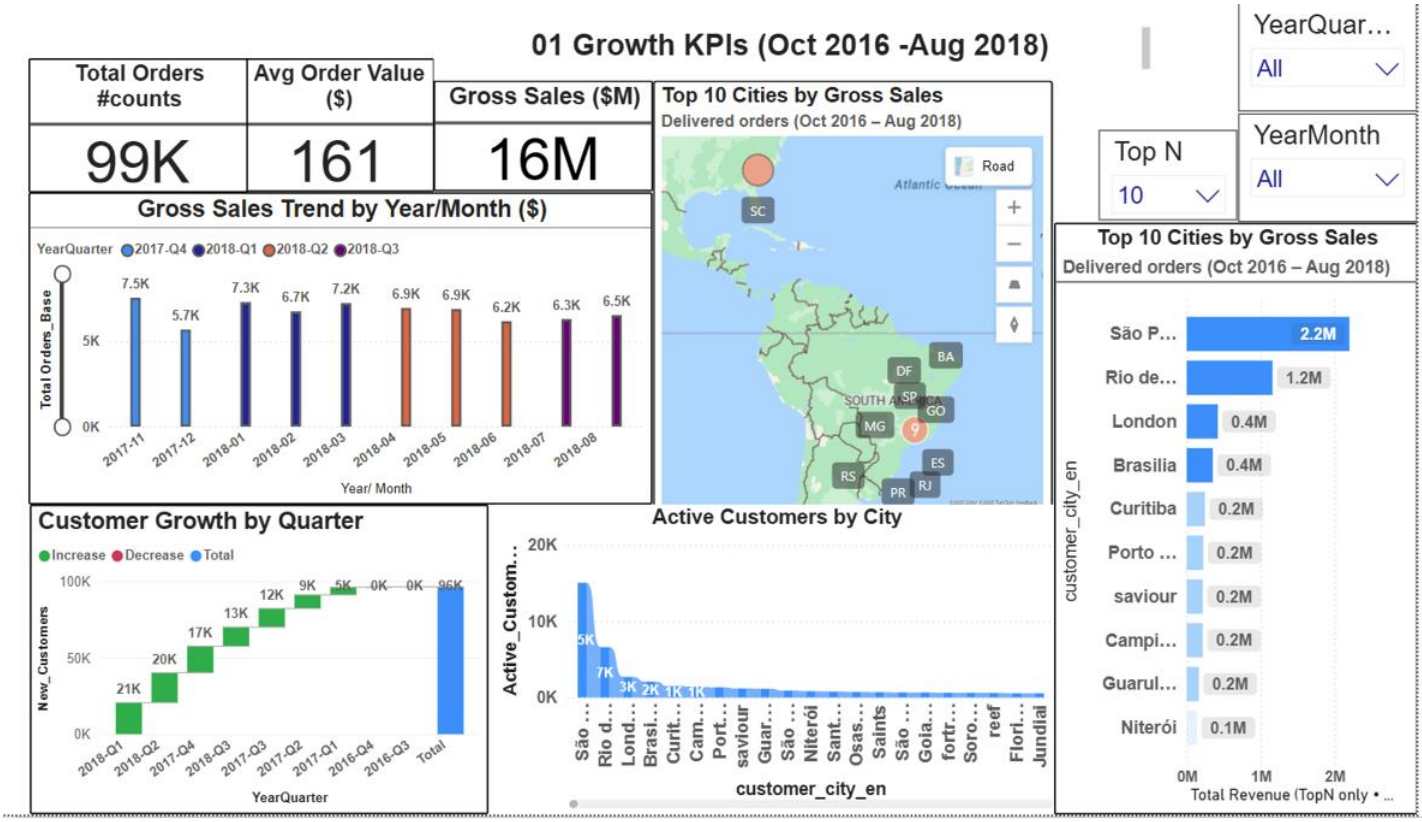
4. **Are sellers strengthening or harming trust?**

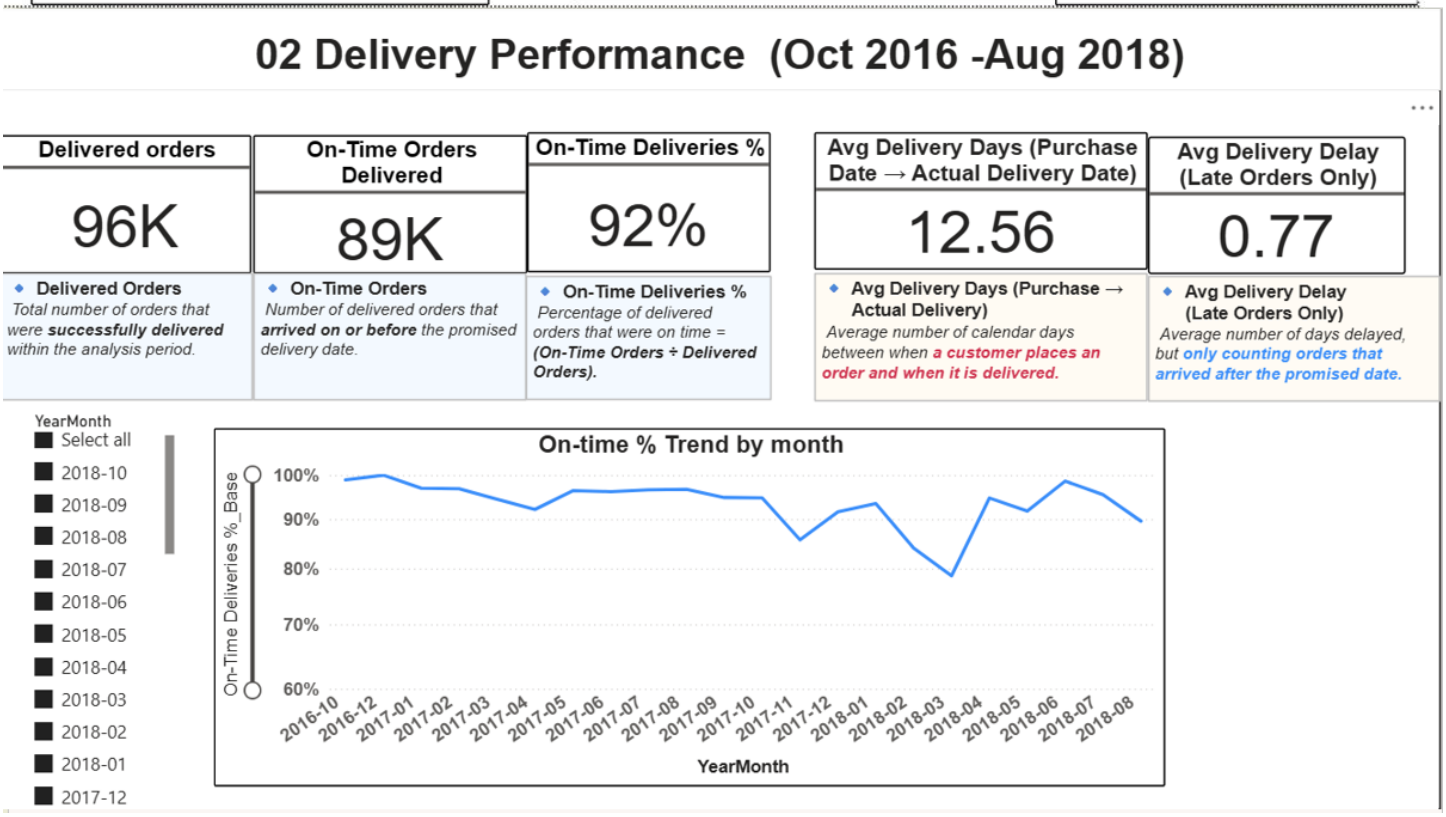
"Our sellers *are* the marketplace. Good ones build trust; weak ones destroy it."

5. **Where are the biggest risks this month?**

"I don't want 20 pages of KPIs — I need a red flag list of risks now."




01 Growth KPIs (Oct 2016 -Aug 2018)
Total Orders #counts
99K
Avg Order Value ($)
161
Gross Sales ($M)
16M
Gross Sales Trend by Year/Month ($)
YearQuarter 2017-Q4 2018-Q1 2018-Q2 2018-Q3
Total Orders_Base
Year/ Month
Top 10 Cities by Gross Sales
Delivered orders (Oct 2016 – Aug 2018)
Road
YearQuar...
All
Top N
10
YearMonth
All
Top 10 Cities by Gross Sales
Delivered orders (Oct 2016 – Aug 2018)
São P... 2.2M
Rio de... 1.2M
London 0.4M
Brasilia 0.4M
Curitiba 0.2M
Porto ... 0.2M
saviour 0.2M
Campi... 0.2M
Guarul... 0.2M
Niterói 0.1M
customer_city_en
Total Revenue (TopN only • ...
Customer Growth by Quarter
Increase Decrease Total
New_Customers
YearQuarter
Active Customers by City
Active_Custom...
customer_city_en


02 Delivery Performance (Oct 2016 -Aug 2018)
Delivered orders
96K
On-Time Orders Delivered
89K
On-Time Deliveries %
92%
Avg Delivery Days (Purchase Date → Actual Delivery Date)
12.56
Avg Delivery Delay (Late Orders Only)
0.77
Delivered Orders
Total number of orders that were successfully delivered within the analysis period.
On-Time Orders
Number of delivered orders that arrived on or before the promised delivery date.
On-Time Deliveries %
Percentage of delivered orders that were on time = (On-Time Orders ÷ Delivered Orders).
Avg Delivery Days (Purchase → Actual Delivery)
Average number of calendar days between when a customer places an order and when it is delivered.
Avg Delivery Delay (Late Orders Only)
Average number of days delayed, but only counting orders that arrived after the promised date.
YearMonth
Select all
2018-10
2018-09
2018-08
2018-07
2018-06
2018-05
2018-04
2018-03
2018-02
2018-01
2017-12
On-time % Trend by month
On-Time Deliveries %_Base
YearMonth

03 Customer Retention (Oct 2016 -Aug 2018)

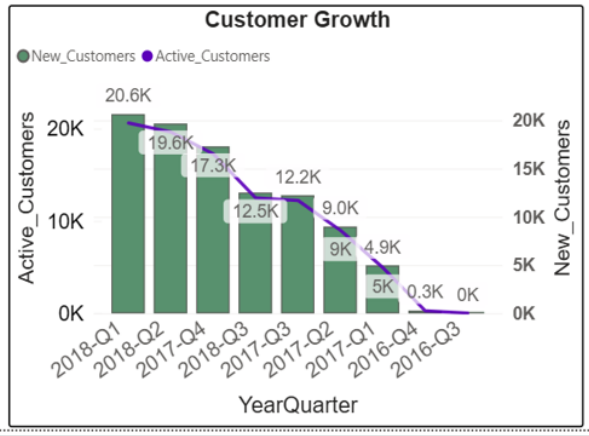
Quarterly new-customer acquisition is slowing.

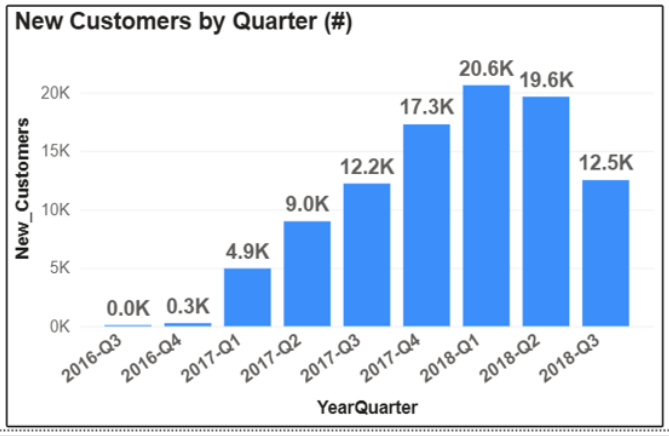
At the same time, the real story in the data is that active customers average 6–7K per month, but are trending downward into 2018.

Repeat purchases are rare, so overall lift depends on **winning back inactive customers and securing second purchases quickly**.

Possible Reasons For Decline in New Customers

- **Market Saturation** – Rapid early growth left fewer untapped customers.
- **Seasonality** – Q3/Q4 shifts in demand (holidays, school, budget cycles).
- **Operational Issues** – Bottlenecks in logistics or onboarding slowed growth.
- **Marketing Shift** – Less ad spend, more focus on retention/profitability.
- **Competition** – New rivals and promotions diverted potential customers.





04 Seller Trust & Ranking (Oct 2016 -Aug 2018)

On-Time Deliveries %	Delivered Orders
92%	96K

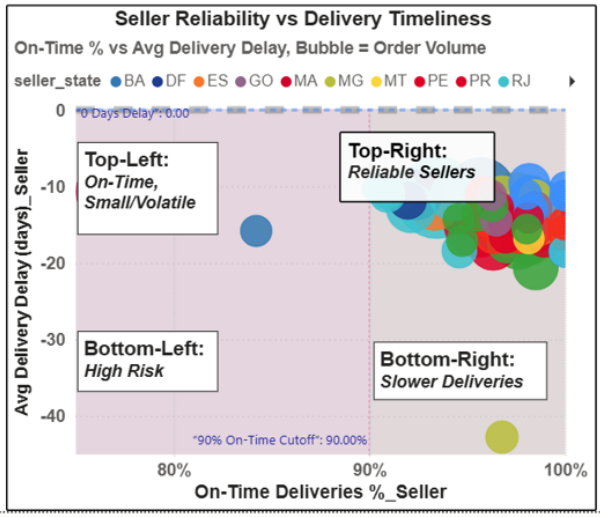
% Orders Cancelled	Delivery Delay (days)
0.5%	-11.20

YearMonth: Multiple s…

Top N: 10

Seller Reliability & Order Performance

seller_id	Delivered Orders	On-Time Deliveries %
6560211a19b47992c3666cc44a7e94c0	1819	0.94
4a3ca9315b744ce9f8e93743614938 84	1772	0.89
cc419e0650a3c5ba77189a1882b7556a	1651	0.94
1f50f920176fa81dab994f9023523100	1399	0.89
da8622b14eb17a	1311	0.92
Total	32788	0.92



seller_id: All

05 TOP 5 Risks

Worst On-Time % — Top 10 Sellers

seller_id	Visible Sellers – Any Table (v2)	On-Time Deliveries %_Seller	seller_state
001cca7ae9ae17fb1caed9dfb1094831	1	0.93	ES
001e6ad469a905060d959994f1b41e4f	1		RJ
02a2272692e13558373c66db98f05e2e	1		RJ
02d35243ea2e497335cd0f076b45675d	1	0.64	RN
Total	**424**	**0.83**	

Worst Avg Delay (days) — Top 10 Sellers

seller_id	Visible Sellers – Any Table (v2)	Avg Delivery Delay (days)_Seller	seller_state
001e6ad469a905060d959994f1b41e4f	1		RJ
02a2272692e13558373c66db98f05e2e	1		RJ
02d35243ea2e497335cd0f076b45675d	1	-5.45	RN
0417b067eeab773d2f7061a726dc477f	1	-3.21	SC
04843805947f0fc584fc1969b6e50fe7	1	0.63	RS
Total	**183**	**-10.47**	

Worst Cancellation Rate % — Top 10 Sellers

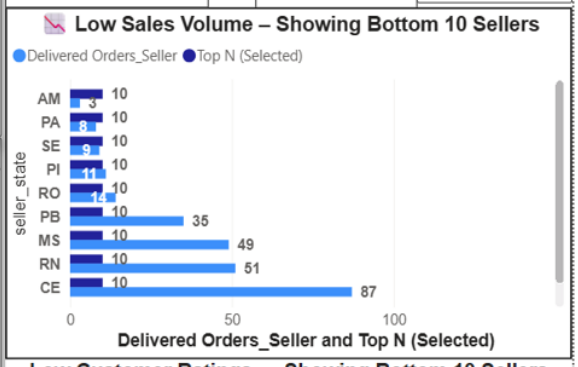
seller_id	Visible Sellers – Any Table (v2)	Cancellation Rate %_Seller	seller_state
001cca7ae9ae17fb1caed9dfb1094831	1		ES
001e6ad469a905060d959994f1b41e4f	1	1.00	RJ
003554e2dce176b5555353e4f3555ac8	1		GO
01bee9d254e0143f0ee0791b060b2e47	1		GO
Total	**581**	**0.02**	

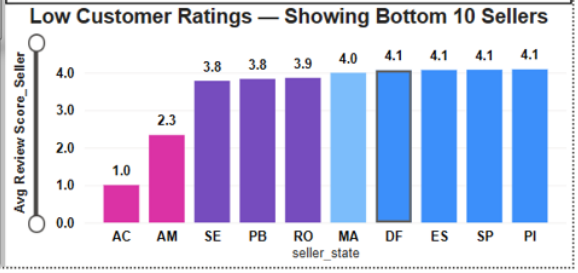
Total No. of Sellers

3095

10
Top N (Selected)

Show Worst N
Top N
10





Risks Identified & Action Steps

The Risk Snapshot shows three hot spots: **low ratings, on-time delivery gaps, and cancellations**.

Action: roll out an **Earned Trust program** — coaching and incentives tied to OTD, cancellation, and review lifts.

Seller reviews are concentrated in top states but flat at ~4.1.

Focus needs to be on lifting underperforming high-volume states.

Net: growth is real, but we protect it by pushing OTD ≥95% and reviews ≥4.3."

Monthly sales confirm seasonality spikes — we should replicate peak-month playbooks two months early. Revenue is highly concentrated in a handful of states.

Action:

Strategy: **land-and-expand** in the top five, and pilot two adjacent states with similar demographics.

Quarterly new customers are slowing, so overall lift hinges on retention. We must win back inactive customers and secure second purchases within 30 days.

By city, we see a long tail of small but meaningful customer bases.

Our next $1M comes from targeting the **next 10 cities** with lightweight demand gen and faster delivery promises.

Decision: Double down on top-5 states and surgically activate the next 10 cities."

Growth is strong, retention is steady, but seller reliability is our weak spot To scale, we must *protect trust* with OTD ≥95%, reviews ≥4.3, and a ruthless focus on reliable sellers."

Risks Identified & Action Steps

"Here's execution reliability.

We delivered **96K orders, 89K on time → 92% OTD**. Average delivery is **12.6 days** from purchase, and late orders are only **0.77 days delayed**.

Good news: delays are shallow; bad news: OTD is inconsistent.

Trend line shows months in the mid-80s OTD, then recovery to ~95%. This volatility is the biggest threat to trust.

Action: enforce stricter SLAs with lagging sellers/carriers, and fund predictive monitoring to flag risks before they damage NPS."

"Here's highlights our seller-level reliability.

Overall: **92% OTD, 96K orders, cancellation only 0.5%**. But within that, large sellers dip to **89% OTD**, below our 90% cutoff.

The scatterplot shows clusters of sellers consistently underperforming while still driving volume — structural risk.

Action: implement a **Seller Reliability Program** with a 95% OTD minimum threshold, tiered performance recognition, and penalties for laggards."

"Now to loyalty.

Headline metrics show **100% retention, new, and repeat purchase** (due to setup quirk). The real story is in the detail: **6–7K active customers per month, trending downward** into 2018.

New vs returning looks balanced, but churn is high — we're replacing lost customers with new ones instead of compounding loyalty.

Action: approve retention programs. Specifically:

- Loyalty reward for **second purchase within 30 days**
- Win-back campaigns for customers inactive beyond 90 days."

"Finally, here's where exposure lives.

- **Worst OTD %:** Some sellers as low as **83%**, concentrated in RJ, ES, RN.
- **Worst Avg Delays**: >10 days in RJ and RN.
- **Worst Cancellation Rates**: up to 100%, abandoned sellers.
- **Low Sales Volume**: 3K+ sellers contribute negligible orders.
- **Low Ratings**: states AC and AM average **1.0–2.3 ratings**, damaging trust.

Action: approve a **'Cut or Coach' program**. We offboard bottom-decile sellers, retrain salvageable ones, and protect expansion states by removing sub-3.5 rating sellers.

Challenges And Future Roadmap

CHALLENGES AND FUTURE ROADMAP

As part of the project initiative, the team identified key technical challenges that could compromise pipeline reliability and reporting accuracy. To address these risks, robust solutions were implemented across the notebook execution and semantic modeling layers, ensuring proactive error handling and seamless dashboard refresh.

Challenge 1: Ensuring Robust Error Handling in Notebooks

Problem:

Notebook failures such as schema mismatches or missing files can silently disrupt pipeline execution, leading to incomplete data processing and delayed reporting.

Solution:

- Implemented try-except blocks with exception chaining to capture and propagate errors.
- Integrated Outlook alerting to notify the data engineering team with actionable error context, including pipeline name, timestamp, and failure details.
- Ensured that all validation notebooks log execution status to a centralized Delta table for traceability and trend analysis.

Challenge 2: Power BI Semantic Model Refresh Failures

Problem:

Power BI dashboards may fail to update due to schema changes or upstream data issues, resulting in outdated or missing insights for stakeholders.

Solution:

- Automated semantic model refresh using Fabric pipeline triggers.
- Logged refresh status and validation results to the monitoring dashboard.
- Configured Outlook notifications to alert stakeholders upon successful or failed refresh, enabling rapid response and resolution

Challenge 3: Inconsistent data validation across pipeline stages

Problem:

Data anomalies (e.g., nulls, outliers, schema drift) may pass undetected from Bronze to Gold layers, leading to misleading insights or report failures.

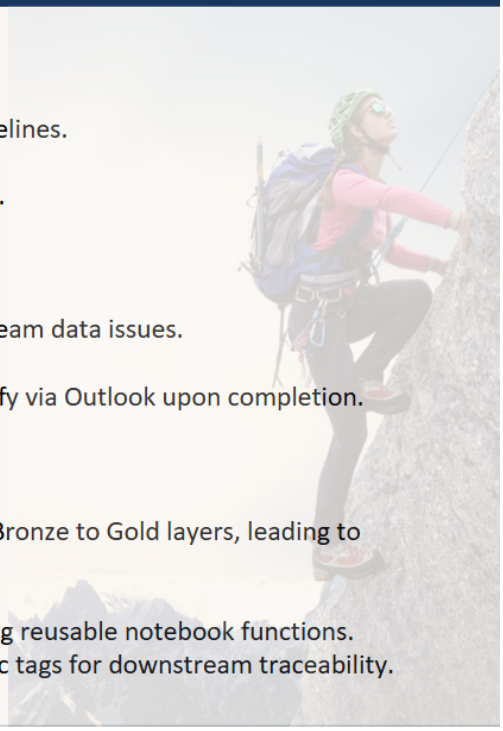
Solution:

Design modular validation checkpoints at each layer (Bronze, Silver, Gold) using reusable notebook functions. Trigger alerts via Outlook when anomalies exceed thresholds, and log semantic tags for downstream traceability.

Architectural Impact

- Strengthened pipeline resilience through modular error handling and alerting.
- Improved observability with centralized logging and dashboard integration.
- Enhanced stakeholder confidence by ensuring timely, accurate, and transparent reporting

Challenges and Solutions



1. Ensuring robust errors handling in notebooks

Notebook failures (e.g., schema mismatch, missing files) can silently break pipelines.

Solution:

Implement try-except blocks and trigger Outlook alert with actionable context.

2. Power BI report fail to update with new data

Power BI semantic models may fail to refresh due to schema changes or upstream data issues.

Solution:

Automate refresh using Fabric pipeline triggers semantic model fresh and notify via Outlook upon completion.

3. Inconsistent data validation across pipeline stages

Data anomalies (e.g., nulls, outliers, schema drift) may pass undetected from Bronze to Gold layers, leading to misleading insights or report failures.

Solution:

Design modular validation checkpoints at each layer (Bronze, Silver, Gold) using reusable notebook functions.
Trigger alerts via Outlook when anomalies exceed thresholds, and log semantic tags for downstream traceability.

IMPACT AND FUTURE ROADMAP

The project has delivered measurable improvements in operational efficiency and data reliability through modular design, automated workflows, and multi-layer validation. Looking ahead, the team is focused on expanding observability and alerting capabilities to further enhance responsiveness and stakeholder confidence.

Impact

1. Operational Efficiency

- Modular Notebooks: Reusable validation and transformation logic reduces manual intervention and accelerates development cycles.
- Automated Execution: Ingestion, transformation, and alerting workflows are fully automated, ensuring consistent and timely data delivery.

2. Data Trust & Quality

- Multi-Step Validation: Embedded schema checks, row count reconciliations, and semantic consistency validations ensure high data integrity across Bronze, Silver, and Gold layers.
- Semantic Alignment: Cleaned and enriched data supports reliable reporting and machine learning workflows.

Future Roadmap

1. Alerting & Observability

- Power Automate Integration: Extend alerting channels beyond Outlook to include Slack, Microsoft Teams, and SMS.
- Real-Time Monitoring: Enhance dashboards with live pipeline status, error trends, and semantic model refresh metrics.
- Proactive Diagnostics: Introduce anomaly detection and predictive alerts to identify issues before they impact reporting.