DATA PIPELINE DESIGN IN FABRIC

This document presents the design and implementation of the data pipeline developed in **Microsoft Fabric** to support the visualization and analysis of supplier and order performance. The pipeline was built as an end-to-end solution that automates the entire process of **data ingestion, cleaning, transformation, integration, and loading** into a centralized **semantic model**. This model serves as the foundation for creating interactive **Power BI dashboards and reports** that provide insights into key business metrics such as order completion, supplier activity, and service efficiency.

1. Overview of the Data Pipeline Architecture

In the Microsoft Fabric environment, all data assets are stored in **OneLake**, the unified underlying storage layer for every workspace. OneLake ensures that all data (lakehouses, delta tables, warehouses, and other items) is physically stored in a single, governed data lake in **Delta Parquet format**, eliminating duplication and improving data consistency across the workspace.

Because OneLake enables all participants in the Fabric workspace to access and contribute to the *same* data lake, it supports collaboration without the risk of dataset silos or redundant copies. This **shared data fabric** is one of the primary reasons why we select Fabric as the platform for our pipeline: it allows us to build a full end-to-end solution entirely within one environment.

Our workspace architecture is organized into three key components:

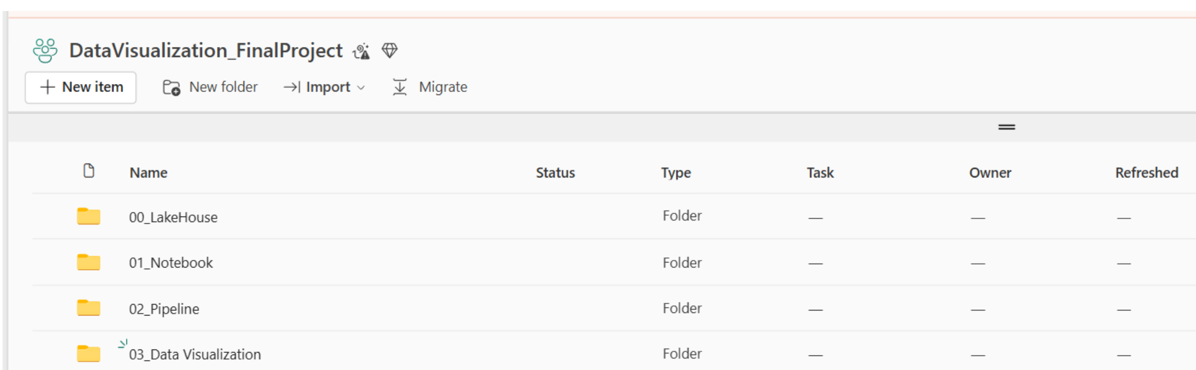


Figure 1: Our workspace architecture

- **Lakehouse**: the storage layer housing all raw files and transformed tables

- **Notebooks**: PySpark (or Spark) code for data transformation, cleaning, and table creation
- **Pipeline/Orchestration**: scheduling and executing notebook tasks and dataflows on a timeline

Within the Lakehouse, we adopt the **medallion (layered) architecture**: **Bronze**, **Silver**, and **Gold**. Each layer serves a distinct purpose in the data processing journey.

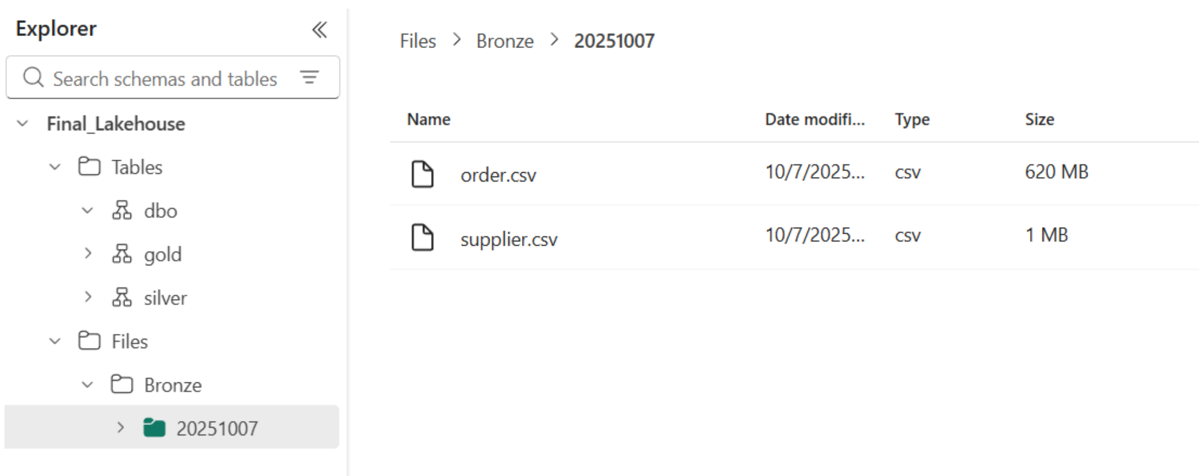


Figure 2: The architecture of lakehouse in Fabric

Bronze Layer

- The Bronze layer acts as the raw ingestion zone.
- We upload each data file (e.g., supplier CSV, order CSV) as-is, preserving its original format and structure.
- These files are stored in subfolders named by the ingestion date (e.g. **Files/Bronze/20251007/**).
- This approach preserves original data fidelity and provides an auditable trail of raw history.

Silver Layer

- In the Silver layer, we convert the Bronze data into structured Delta Parquet tables.
- This stage focuses on cleaning, normalization, type casting, and schema enforcement.

- The Silver tables represent *the latest cleaned snapshot* of each entity - for example, a silver.order table and a silver.supplier table.

Gold Layer

- The Gold layer is the analytic semantic layer built for reporting and BI.
- In this layer, we materialize dimension tables (with SCD Type 2) and fact tables using the cleaned Silver data.
- The Gold tables use surrogate keys, historical versioning, and pre-joined relationships to enable efficient analytics.
- Power BI (via Direct Lake or semantic model) consumes these Gold tables directly, supporting time intelligence, slicers, and consistent metrics.

This layered architecture ensures a clear separation of responsibilities across the data pipeline: raw data ingestion occurs in the Bronze layer, data cleaning and standardization are performed in the Silver layer, and analytical modeling takes place in the Gold layer. This structure enhances data quality by allowing errors to be traced and corrected upstream, improves scalability and performance by optimizing Gold tables for BI workloads, and promotes collaboration as all artifacts are centrally stored and accessible within the shared Fabric Lakehouse environment.

2. Pipeline Components

2.1 Bronze to Silver Notebook: Data Cleaning and Standardization

This notebook is designed to handle data ingestion from the **Bronze layer** incrementally based on the ingestion date. Each time it runs, the notebook reads files stored in subfolders named by their respective ingestion dates, allowing it to capture only newly added or updated data. This incremental approach ensures efficient processing, reduces unnecessary computation, and helps maintain a clear audit trail of data sources over time.

```
# Parameters (pipeline should inject ingest_date; edit for testing)
# ingest_date = "20251007"                    # folder name (YYYYMMDD) — pipeline must set this
bronze_folder = f"Files/Bronze/{ingest_date}/"
silver_order_table = "silver.order"
silver_supplier_table = "silver.supplier"
```

✓ - Command executed in 384 ms by Giang Huong on 9:53:53 PM, 10/11/25

Within the notebook, a series of **data-cleaning and standardization** operations are performed to ensure data consistency and readiness for analysis. These include trimming extra whitespace from text fields, removing rows that contain null or invalid values, and eliminating unmeaningful or redundant columns. Special attention is given to time-related attributes—raw datetime strings are parsed and separated into distinct **date** and **time (HH:MM)** columns. This transformation standardizes the time format across all datasets, improving readability and enabling accurate time-based analysis later in the pipeline.

```
# Trim/normalize column names (remove accsupplier_idental whitespace)
for c in src_raw.columns:
    if c != c.strip():
        src_raw = src_raw.withColumnRenamed(c, c.strip())

# Normalize the supplier supplier_id column ("supplier_id")
#    - remove trailing ".0" for integer-like strings
#    - convert scientific notation to integer string when possible
#    - keep original otherwise
if "supplier_id" in src_raw.columns:
    supplier_id_trim = F.trim(F.col("supplier_id").cast(StringType()))

    supplier_id_norm = (
        F.when(supplier_id_trim.isNull(), None)
         # case: plain integer or integer with .0 suffix (e.g. 12345 or 12345.0)
         .when(supplier_id_trim.rlike(r'^[0-9]+(\.0+)?$'), F.regexp_replace(supplier_id_trim, r'\.0+$', ''))
         # case: scientific notation e.g. 8.3624089559E10 -> cast double then long then string
         .when(supplier_id_trim.rlike(r'^[0-9]+(\.[0-9]+)?[eE][+-]?[0-9]+$'),
               F.col("supplier_id").cast("double").cast("long").cast(StringType()))
         # fallback: remove any accsupplier_idental surrounding quotes and trim
         .otherwise(F.regexp_replace(supplier_id_trim, r'(^"|"$)|(^\'|\'$)', ''))
    )

    src_raw = src_raw.withColumn("supplier_id", supplier_id_norm)
else:
    raise ValueError("Expected column 'supplier_id' in supplier CSV but not found.")
```

```
# List of original timestamp columns present in source
time_columns = [
    "create_time","order_time","accept_time","board_time",
    "pickup_time","complete_time","cancel_time"
]
datetime_fmt = "MMMM d, yyyy, HH:mm"
# Convert each textual time column to a timestamp, then split into date and time parts
for col_name in time_columns:
    ts_expr = F.to_timestamp(F.col(col_name), datetime_fmt)
    date_col = f"{col_name}_date"
    time_col = f"{col_name}_time"
    src_raw = src_raw.withColumn(date_col, F.to_date(ts_expr)) \
                     .withColumn(time_col, F.date_format(ts_expr, "HH:mm"))
```

After completing all transformation steps, the notebook writes the cleaned and standardized data into the **Silver layer** as a **Delta table**. A **merge mechanism** (using business keys) is applied to maintain data integrity. When a record from the Bronze layer matches an existing record in Silver, the system updates the existing entry; otherwise, it inserts the new row. This ensures that the Silver table always reflects the **latest and most reliable version** of the data, serving as a consistent and high-quality source for downstream processing and reporting.

```
# 4) Perform MERGE using DeltaTable API
#    Build join condition for composite keys
join_condition = " AND ".join([f"t.{k} = s.{k}" for k in business_keys])

# Get DeltaTable reference (if table was just created above, this will succeed)
target = DeltaTable.forName(spark, silver_table_fq)

# But whenMatchedUpdateAll/whenNotMatchedInsertAll() is concise and works if column names match.
print("Starting MERGE (upsert) into", silver_table_fq)
(target.alias("t")
       .merge(src_dedup.alias("s"), join_condition)
       .whenMatchedUpdateAll()
       .whenNotMatchedInsertAll()
       .execute())
print("MERGE completed into", silver_table_fq)
```

2.2 Silver to Gold Notebook: Data Modeling and Integration

This notebook focuses on transforming standardized data from the Silver layer into an analytical data model stored in the Gold layer, which is optimized for reporting and visualization in Power BI. The process involves constructing both dimension (Dim) and fact (Fact) tables following a **Snowflake schema** design, where certain dimensions such as City and District are normalized into separate but related tables. This schema enhances data integrity and reduces redundancy while maintaining efficient query performance and clear hierarchical relationships between business entities.

The notebook begins by creating **time-related dimension tables**, **Dim Date** and **Dim Hour**, which form the temporal backbone of the data model. These two tables are generated programmatically rather than derived from transactional data, as their values are universal and not subject to change over time. Therefore, **Slowly Changing Dimension (SCD) Type 2** logic is **not applied** to these tables.

2.2.1 Conceptual and Logical Model Design

The Gold layer's data model was designed following **dimensional modeling principles** to transform operational data, originally stored in an OLTP schema into, a structure optimized for analytical workloads (OLAP). In its raw form, the transactional schema was normalized for write performance but unsuitable for multidimensional analysis, resulting in inefficient query execution and complex time-based aggregation.

To address these challenges, a **Snowflake schema** was implemented as the core analytical model. This schema enables:

- **Reduced data redundancy**, by separating descriptive data into shared dimension tables.
- **Improved analytical efficiency**, by simplifying aggregations across business entities.
- **Hierarchical representation of business structures**, particularly the spatial hierarchy of City → District.

At the conceptual level, the schema centers around one primary business process

- **Order Fulfillment**, which is represented by the **Fact Order** table. This table records quantitative events such as delivery distance, order completion time, and cancellation status. Surrounding the fact table are eight descriptive dimensions that capture contextual information ("who, what, when, where") for each transaction:

Table 1: Dimension description

Dimension	Description
Dim Supplier	Master and operational attributes of the delivery supplier (driver).
Dim Date	Calendar-based temporal attributes (day, month, quarter, year).
Dim Hour	Intraday time attributes (hour, minute).
Dim Service	Type of delivery service offered.
Dim Status	Operational order status.
Dim Stop Status	Delivery stop outcome or condition.
Dim District	Sub-city geographic entity (district).

Dim City	Higher-level geographic entity (city).

Unlike a pure Star schema, the Snowflake schema introduces a normalized relationship between Dim District and Dim City to accurately represent the hierarchical nature of geographic data. This normalization improves referential integrity, reduces redundancy, and better reflects real-world administrative structures.

The model was developed specifically for the **HN-Express** division, focusing on orders delivered within **Hanoi in 2023**. This spatial and temporal scope ensures analytical precision while keeping the dataset at a manageable scale for efficient processing within Microsoft Fabric.

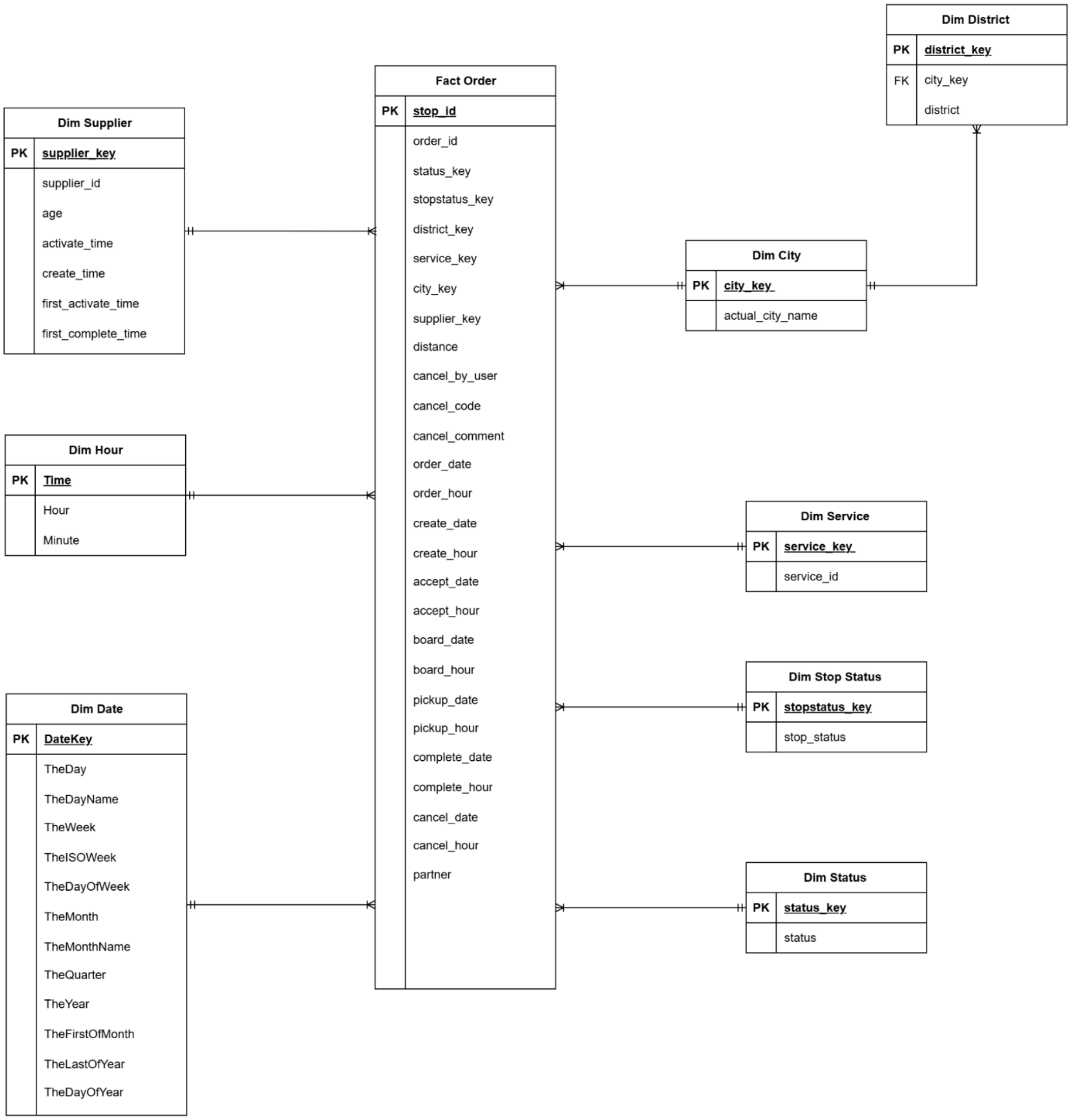


Figure 4: Snowflake Schema Model of Data Mart

2.2.2 Source-to-Destination Mapping

To operationalize the logical schema, data sources from the operational system were mapped to their respective target tables in the Gold Data Mart. This mapping ensures full lineage and traceability from source to warehouse, forming the foundation for ETL implementation.

Table 2: Source-to-Destination Data Flow in the ETL Pipeline.

Data Source Table	Target Tables in Data Mart	Description
Order	Dim Service, Dim Status, Dim Stop Status, Dim District, Dim City	Service attributes, order statuses, and geographic information are extracted from the source Order table to populate these distinct dimension tables. This process ensures normalization and reduces data redundancy.
Supplier	Dim Supplier	The driver master data is extracted directly from the source supplier table. Each record represents a unique driver, utilizing a surrogate key to maintain historical tracking.

2.2.3 Physical Modeling and Dimension Construction

The physical design phase converts the logical model into actual database structures, including the creation of **surrogate keys**, implementation of **Slowly Changing Dimension (SCD)** logic, and initialization of tables. All transformations were executed within **Fabric Notebooks** using **PySpark**.

A critical design decision is the adoption of **surrogate keys** (auto-incrementing integers) as primary keys for all dimension tables, rather than natural business keys. This follows **Kimball's best practices**, providing three main advantages:

- Decoupling the analytical model from source system volatility;
- Optimizing JOIN performance on integer-based keys;
- Enabling historical version tracking via SCD Type 2.

Dim Date

The **Dim Date** table defines the temporal framework for the entire analytical model. Rather than relying on transactional timestamps from the source system, a dedicated time dimension provides a uniform reference for date-based aggregation and comparison. This approach follows the Kimball methodology, which emphasizes decoupling time analysis from event data to ensure consistent reporting granularity. In the Fabric Notebook, the **Dim Date** table was programmatically generated using PySpark date functions (as shown in *Figure below, Code for Dim Date*). The script first created a continuous sequence of calendar dates, ranging from January 1, 2020, to December 31, 2030.

```
# Define start/end date range
start_date = datetime.date(2020, 1, 1)
end_date = datetime.date(2030, 12, 31)

# Create continuous date sequence
date_df = spark.createDataFrame(
    [(start_date + datetime.timedelta(days=i),)
     for i in range((end_date - start_date).days + 1)],
    ["the_day"]
).withColumn("the_day", F.col("the_day").cast(DateType()))

# Derive all DimDate attributes
dim_date = (
    date_df
    .withColumn("date_key", F.date_format("the_day", "yyyyMMdd").cast(IntegerType()))
    .withColumn("the_day_name", F.date_format("the_day", "EEEE"))
    .withColumn("the_week", F.weekofyear("the_day"))
    .withColumn("the_iso_week", F.weekofyear("the_day"))
    # Day of week: Monday=1, Sunday=7
    .withColumn("the_day_of_week", ((F.dayofweek("the_day") + 5) % 7 + 1))
    .withColumn("the_month", F.month("the_day"))
    .withColumn("the_month_name", F.date_format("the_day", "MMMM"))
    .withColumn("the_quarter", F.quarter("the_day"))
    .withColumn("the_year", F.year("the_day"))
    .withColumn("the_first_of_month", F.trunc("the_day", "month"))
```

```python
    .withColumn(
        "the_last_of_year",
        F.to_date(F.concat_ws("-", F.col("the_year"), F.lit("12"), F.lit("31")))
    )
    .withColumn("the_day_of_year", F.dayofyear("the_day"))
    .select(
        "date_key",
        "the_day",
        "the_day_name",
        "the_week",
        "the_iso_week",
        "the_day_of_week",
        "the_month",
        "the_month_name",
        "the_quarter",
        "the_year",
        "the_first_of_month",
        "the_last_of_year",
        "the_day_of_year"
    )
)

# Write table
dim_date.write.mode("overwrite").saveAsTable(dim_date_table)
```

From this base date sequence, a rich set of descriptive attributes was derived and pre-calculated. These attributes include **date_key** (the primary key, formatted as 'yyyyMMdd'), **the_day_name** (e.g., 'Monday'), **the_week** (week of year), **the_month_name**, **the_quarter**, **the_year**, **the_first_of_month**, **the_last_of_year**, and **the_day_of_year**. This precomputation of all time attributes ensures flexibility and high performance for hierarchical analyses in Power BI (e.g., *Year → Quarter → Month → Day*). Because time is a universal and non-changing concept, the **Dim Date** table is classified as a non–Slowly Changing Dimension (non-SCD). It is a static, versionless table that is loaded once into the Gold layer using the write.mode("overwrite") operation (as shown in the code) during the Data Mart initialization.

Data preview - dim_date

Showing 1000 rows

	date_key	the_day	the_day_n...	the_week	the_iso_w...	the_day_o...	the_day_o...	the_month	the_mont...	the_quarter	the_year	the_first_o...	the_last_of...
1	20221001	2022-10-01	Saturday	39	39	6	274	10	October	4	2022	2022-10-01	2022-12-31
2	20221002	2022-10-02	Sunday	39	39	7	275	10	October	4	2022	2022-10-01	2022-12-31
3	20221003	2022-10-03	Monday	40	40	1	276	10	October	4	2022	2022-10-01	2022-12-31
4	20221004	2022-10-04	Tuesday	40	40	2	277	10	October	4	2022	2022-10-01	2022-12-31
5	20221005	2022-10-05	Wednesday	40	40	3	278	10	October	4	2022	2022-10-01	2022-12-31
6	20221006	2022-10-06	Thursday	40	40	4	279	10	October	4	2022	2022-10-01	2022-12-31
7	20221007	2022-10-07	Friday	40	40	5	280	10	October	4	2022	2022-10-01	2022-12-31
8	20221008	2022-10-08	Saturday	40	40	6	281	10	October	4	2022	2022-10-01	2022-12-31
9	20221009	2022-10-09	Sunday	40	40	7	282	10	October	4	2022	2022-10-01	2022-12-31
10	20221010	2022-10-10	Monday	41	41	1	283	10	October	4	2022	2022-10-01	2022-12-31

Figure 5: Data of DimDate

Dim Hour

To provide granular intraday (within-a-day) analysis, the **Dim Hour** table was implemented to complement **Dim Date**. This table provides the finest level of temporal granularity required for the analytical model. As shown in the Fabric Notebook script - *Code for Dim Hour*, this dimension was programmatically generated. The script first defined ranges for all 24 hours and 60 minutes, then executed a Cartesian product to create a comprehensive sequence of all 1,440 minutes in a day.

```
# Create full range of hours and minutes
hours = list(range(0, 24))
minutes = list(range(0, 60))
dim_hour_table = "gold.dim_hour"
```

Command executed in 390 ms by Giang Huong on 10:00:53 PM, 10/08/25

```
# Create combinations (Cartesian product)
time_data = [(h, m) for h in hours for m in minutes]

dim_hour_df = spark.createDataFrame(time_data, ["hour", "minute"]) \
    .withColumn("hour", F.format_string("%02d", F.col("hour")).cast(StringType())) \
    .withColumn("minute", F.format_string("%02d", F.col("minute")).cast(StringType())) \
    .withColumn("time", F.concat_ws(":", F.col("hour"), F.col("minute")))

# Reorder columns to match schema
dim_hour_df = dim_hour_df.select("time", "hour", "minute")

# Save to gold table
dim_hour_df.write.mode("overwrite").saveAsTable(dim_hour_table)
```

The resulting dimension table consists of three attributes: **time** (a formatted 'HH:mm' string, e.g., "09:30"), **hour** (a two-digit string), and **minute** (a two-digit string). This minute-level granularity is essential for accurately analyzing supplier activity patterns and identifying precise order peaks throughout the day.

Like **Dim Date**, the **Dim Hour** table serves as a static temporal reference dimension and is not subject to Slowly Changing Dimension (SCD) logic. In contrast, all other dimension tables in this model such as **Dim Supplier**, **Dim City**, **Dim District**, **Dim Service**, **Dim Status**, and **Dim Stop Status** implement the **SCD Type 2** mechanism. This approach preserves historical records whenever attribute changes occur, maintaining the time-consistent integrity of all trend analyses.

Data preview - dim_hour | Showing 1000 rows | Search

	time	hour	minute
1	03:00	03	00
2	03:01	03	01
3	03:02	03	02
4	03:03	03	03
5	03:04	03	04
6	03:05	03	05
7	03:06	03	06
8	03:07	03	07
9	03:08	03	08
10	03:09	03	09

Figure 6: Data of DimHour

Dim Supplier

The Dim Supplier table represents the master data file for the suppliers (drivers), enriched with descriptive attributes defined in the schema (such as **age**, **activate_time**, **create_time**, and **last_activity**). This table is loaded from the Silver supplier data and implements **SCD Type 2** logic to maintain a full historical record of these attributes. For example, when a supplier's information (such as their **age** or **last_activity** timestamp) changes in the source data, the system marks the previous record as inactive (is_current = False) and inserts a new record with the updated attributes. This ensures accurate historical reporting on supplier performance and status over time.

Data preview - dim_supplier | Showing 1000 rows | Search

	supplier_key	supplier_id	age	is_current	effective_start...	effective_end_ts	activate_time	create_time	first_activate_t...	first_complete...	age_group
1	1	84315126058	25	1	2025-10-15 10:59:...	NULL	2023-02-07 21:48:...	2023-02-07 21:41:...	2023-02-09 11:47:...	2023-02-09 14:23:...	23-35
2	2	84315148516	21	1	2025-10-15 10:59:...	NULL	NULL	2022-03-09 09:51:...	2022-03-11 17:36:...	2022-03-12 07:47:...	<= 22
3	3	84315153284	23	1	2025-10-15 10:59:...	NULL	2023-07-10 12:30:...	2023-07-10 12:29:...	2023-07-11 18:06:...	2023-07-12 09:20:...	23-35
4	4	84315173039	32	1	2025-10-15 10:59:...	NULL	NULL	2021-05-29 09:41:...	2021-05-31 18:34:...	2021-06-01 11:28:...	23-35
5	5	84315173818	23	1	2025-10-15 10:59:...	NULL	NULL	2022-09-10 13:33:...	2022-09-12 17:36:...	2022-09-13 10:34:...	23-35
6	6	84315216327	32	1	2025-10-15 10:59:...	NULL	NULL	2022-07-30 14:32:...	2022-07-31 10:34:...	2022-08-01 09:52:...	23-35
7	7	84315218420	37	1	2025-10-15 10:59:...	NULL	NULL	2023-10-25 16:08:...	2023-10-28 16:03:...	2023-10-29 07:54:...	36-50
8	8	84315234125	24	1	2025-10-15 10:59:...	NULL	NULL	2023-10-09 20:09:...	2023-10-12 15:39:...	2023-10-12 21:29:...	23-35
9	9	84315307447	34	1	2025-10-15 10:59:...	NULL	NULL	2023-05-29 15:48:...	2023-06-07 17:04:...	2023-06-08 16:47:...	23-35
10	10	84315333999	26	1	2025-10-15 10:59:...	NULL	2023-08-06 13:47:...	2023-08-06 13:42:...	2023-08-08 17:35:...	2023-08-09 15:00:...	23-35

Figure 7: Data of Dim Supplier

Dim City and Dim District

The **Dim City** and **Dim District** tables capture the geographical hierarchy of business operations. Dim City contains city-level attributes like city_key and actual_city_name, while Dim District refines the hierarchy with district_key, district, and a foreign key linking to Dim City. Both dimensions also apply **SCD Type 2** updates to track location changes or renamings that may occur over time.

Data preview - dim_district Showing 1000 rows

	district_key	is_current	effective_start_ts	effective_end_ts	city_key	district	latitude	longitude
1	20	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Đông Anh	21.13	105.82
2	16	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Gia Lâm	21.02	105.95
3	14	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Hoài Đức	21.02	105.68
4	33	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Ba Vì	21.15	105.4
5	17	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Chương Mỹ	20.85	105.68
6	18	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Đan Phượng	21.09	105.68
7	31	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Thường Tín	20.83	105.88
8	25	1	2025-10-26 08:25:59.070660	*NULL*	1	Huyện Ứng Hòa	20.73	105.8
9	9	1	2025-10-26 08:25:59.070660	*NULL*	1	Quận Ba Đình	21.0366	105.8347
10	8	1	2025-10-26 08:25:59.070660	*NULL*	1	Quận Hoàng Mai	20.97	105.86

Dim Service

The **Dim Service** table represents different service categories or delivery types offered. It is populated from Silver service data, with fields like service_key, service_id, and service_name. SCD Type 2 is applied to preserve historical service configuration changes.

Data preview - dim_service Showing 1000 rows

	service_key	service_id	is_current	effective_start_ts	effective_end_ts
1	1	HAN-EXPRESS	1	2025-10-15 10:10:00.717727	*NULL*

Figure 8: Data of Dim Service

Dim Status and Dim Stop Status

These dimensions store categorical status values that describe the lifecycle of orders, such as delivery status or stop reason. They provide meaningful context for the **Fact Order** table. Both dimensions implement **SCD Type 2** to capture updates to status definitions without losing historical integrity.

Data preview - dim_status Showing 1000 rows

	status_key	status	is_current	effective_start_ts	effective_end_ts
1	1	CANCELLED	1	2025-10-09 08:53:45.733342	*NULL*
2	2	COMPLETED	1	2025-10-09 08:53:45.733342	*NULL*

Data preview - dim_stop_status Showing 1000 rows

	stopstatus_key	stop_status	is_current	effective_start_ts	effective_end_ts
1	1	COMPLETED	1	2025-10-09 08:54:05.452586	*NULL*
2	2	FAILED	1	2025-10-09 08:54:05.452586	*NULL*

Figure 9: Preview Data of Status and Dim Stop Status

Fact Order Table Construction

The **Fact Order** table represents the central component of the Snowflake schema, serving as the “core” of the analytical model. It defines the transactional grain of the system, where each record corresponds to a single order event, either successfully completed or attempted.

The construction of this table is the final and most critical step in the Silver → Gold transformation process within the Fabric Notebook. As illustrated in *figure below - Fact Table Joins*, this process begins by retrieving the **order** table (cleaned in the Silver layer) and performing a sequence of **LEFT JOIN** operations.

```
fact = (order
    # join status -> status_key
    .join(dim_status, on=[order.status == dim_status.status], how="left")
    # join stop_status -> stopstatus_key
    .join(dim_stop_status, on=[order.stop_status == dim_stop_status.stop_status], how="left")
    # join service -> service_key
    .join(dim_service, on=[order.service_id == dim_service.service_id], how="left")
    # join district -> district_key using both city_key and district
     .join(dim_district,on=[(order.district == dim_district.district)], how="left")
    # join supplier -> supplier_key (order.supplier_id => dim_supplier.supplier_id)
    .join(dim_supplier, on=[order.supplier_id == dim_supplier.supplier_id], how="left")
    # .join(silver_supplier, on=[order.supplier_id == silver_supplier.id], how="left")
)
```

These joins link the transactional data with the prebuilt dimension tables (**Dim Status**, **Dim Service**, **Dim District**, **Dim Supplier**, etc.). It is important to note that these join conditions rely on **natural business keys** (e.g., order.status == dim_status.status or order.service_id == dim_service.service_id). The purpose of these joins is to **look up and attach surrogate keys** (e.g., status_key, service_key) corresponding to each transaction.

After all joins are completed, a final SELECT statement is executed (see in code *Fact Table Projection*) to project and shape the final Fact table structure for the Gold layer. This structure, optimized for query performance, consists of three distinct groups of columns:

- **Dimension Surrogate Keys:** These are foreign key fields such as status_key, stopstatus_key, service_key, district_key, and supplier_key.
- **Measures and Degenerate Dimensions:** This group includes key business metrics such as distance, and contextual attributes like cancel_code and cancel_by_user.
- **Temporal Join Keys:** These columns—such as create_date, create_hour, complete_date, and complete_hour—link the fact table with **Dim Date** and **Dim Hour**.

```python
fact_selected = fact.select(
    # Core business fields
    F.col("order_id"),
    F.col("stop_id"),
    F.col("distance"),
    F.col("cancel_by_user"),
    F.col("cancel_code"),
    F.col("cancel_comment"),
    F.col("partner"),

    # Dimension surrogate keys
    F.col("status_key"),
    F.col("stopstatus_key"),
    F.col("district_key"),
    F.col("service_key"),
    F.col("supplier_key"),

    # Date & Hour (as HH:mm text)
    F.col("create_time_date").alias("create_date"),
    F.col("create_time_time").alias("create_hour"),

    F.col("order_time_date").alias("order_date"),
    F.col("order_time_time").alias("order_hour"),

    F.col("accept_time_date").alias("accept_date"),
    F.col("accept_time_time").alias("accept_hour"),

    F.col("board_time_date").alias("board_date"),
    F.col("board_time_time").alias("board_hour"),

    F.col("pickup_time_date").alias("pickup_date"),
    F.col("pickup_time_time").alias("pickup_hour"),

    F.col("complete_time_date").alias("complete_date"),
    F.col("complete_time_time").alias("complete_hour"),

    F.col("cancel_time_date").alias("cancel_date"),
    F.col("cancel_time_time").alias("cancel_hour")

)
```

The resulting **Fact Order** table serves as the analytical core for *Power BI*. By integrating spatial (District), temporal (Date, Hour), and operational (Driver, Service) dimensions, analysts can build high-granularity dashboards to explore performance trends, detect operational bottlenecks at the district level, and monitor delivery efficiency in near real time. The table is loaded using an **incremental ETL** approach, ensuring continuous data freshness without the need for full reloads—fully aligned with the principles of the *Medallion Architecture*.

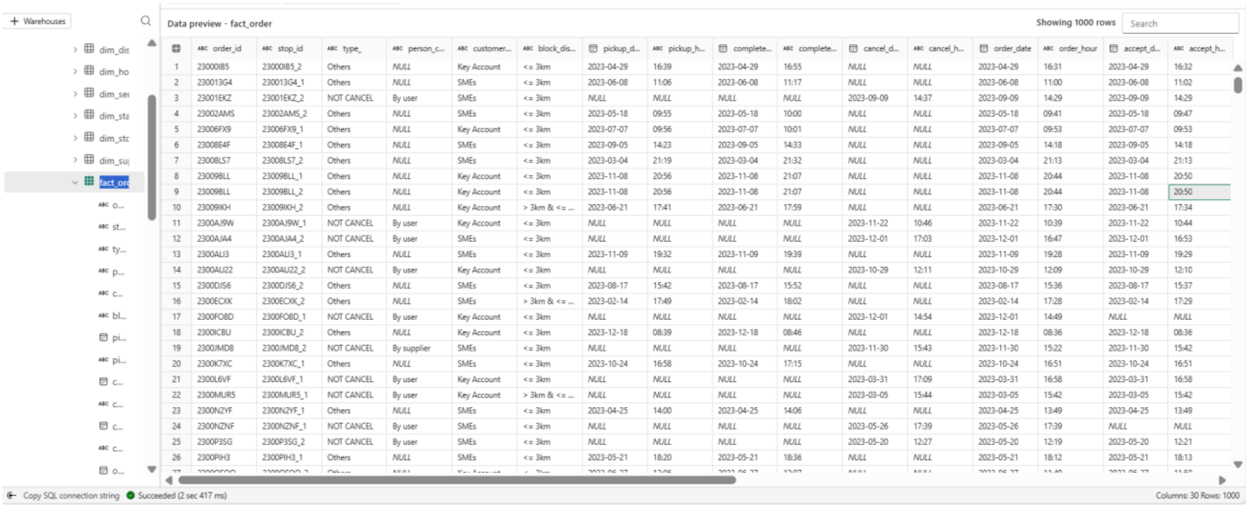
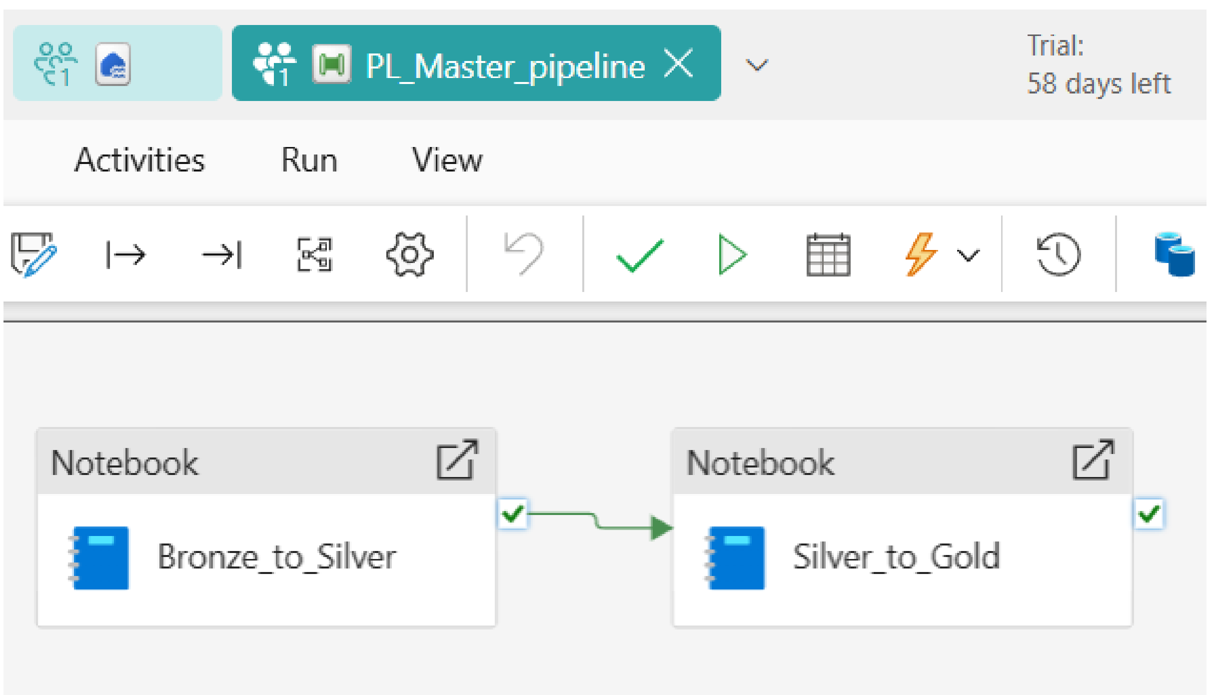


Figure 10: Preview Data of Fact Order

3. Orchestration and Automation in Fabric



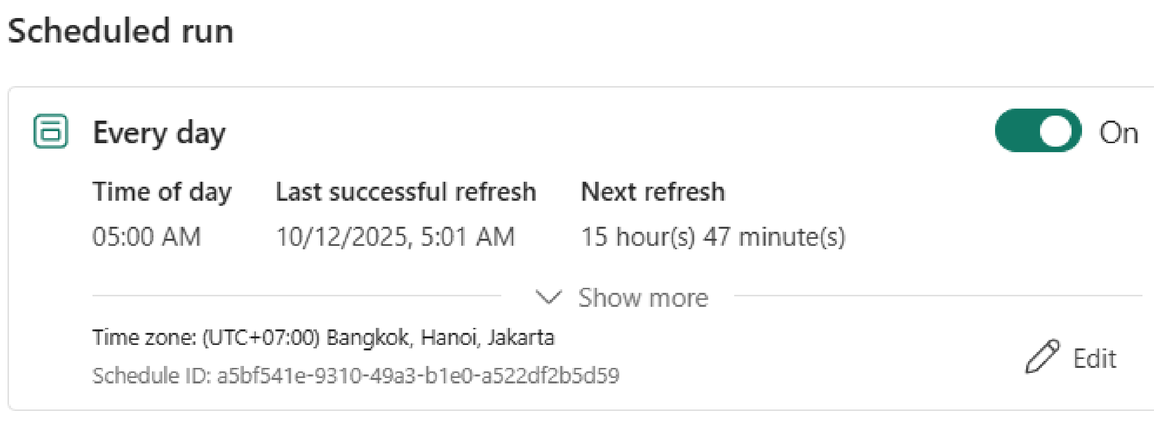


Figure 10: ETL pipeline and schedule running time

To ensure consistent and efficient data processing, the entire workflow is orchestrated using a **Fabric Data Pipeline**. This pipeline automates the execution of two key notebooks: **Bronze to Silver** (responsible for data cleaning and standardization) and **Silver to Gold** (responsible for data modeling and integration). The pipeline is scheduled to run automatically at **5:00 AM every day**, ensuring that all datasets are refreshed with the latest available information before business hours. This automation not only reduces manual intervention but also enhances data reliability, supports timely reporting, and maintains synchronization across all layers of the Lakehouse environment.

4.4. Integration with Power BI Semantic Model

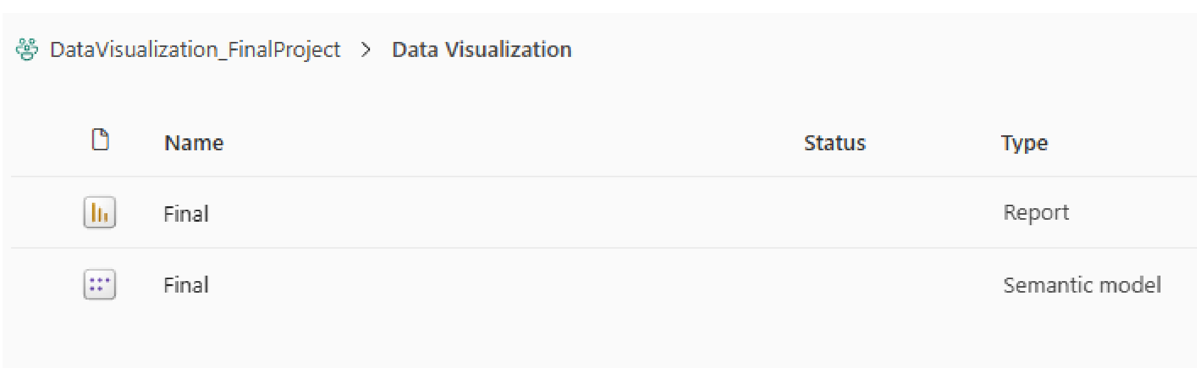


Figure 11: Report and its connected semantic model

In this project, the **Power BI Semantic Model** is connected to the **Gold layer** using **Direct Lake mode**, which allows Power BI to read data directly from the Fabric Lakehouse without data duplication or import. This provides near real-time performance and ensures consistency between the analytical model and the reports. Before connecting to Power BI, the data in the Gold layer was further refined through a transformation notebook, where new columns and derived tables were created by applying specific business rules. Once the semantic model was established, additional **measures and calculated fields** were defined directly in Power BI to support analytical dashboards. This combination of automated data transformation in Fabric and interactive modeling in Power BI delivers a high-performance, scalable, and business-ready reporting solution.