Executive Summary

This project focuses on building a modern, dynamic data pipeline using the Spotify dataset with Azure Data Stack, demonstrating practical skills and yielding valuable insights.

Introduction

In this modern data landscape, everything is driven by data. Scalable, automated and analytical End-to-End pipelines are essential for enabling organisations to effectively process raw data into actionable insights that drive business value.

Data Source

- The structured Spotify dataset was sourced from the GitHub repository of renowned data engineer Anshlamba.
- There are two separate files: one containing over 500 rows for the initial load and another file with approximately 50 rows designated for the incremental process.

Ingestion Layer - Azure Data Factory

- The initial step is to ingest the raw data into the SQL database.
- Then, ADF was utilised to both ingest and orchestrate the pipeline execution.

Storage Layer - Azure Data Lake Storage Gen 2

- The ingested data is then staged within Azure Data Lake Storage Gen 2.

Transformation Layer - Databricks

- Data then flows from the Storage layer to Azure Databricks, where it undergoes transformation processing using PySpark.

Serving Layer

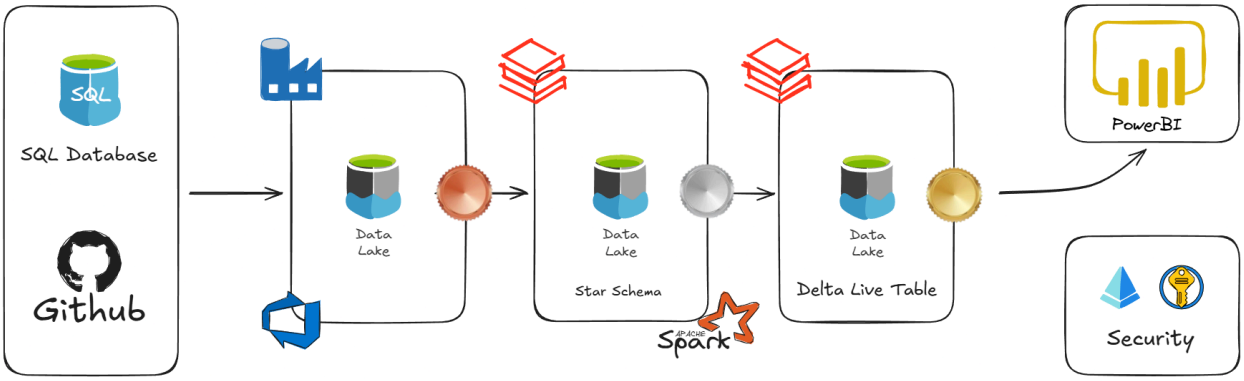
- Following the transformation, the resulting model is served directly.
- Based on the downstream usage, various reporting tools can then be utilised to generate comprehensive reports and dashboards.

Monitoring

- Implemented using Azure Factory Git CI/CD pipeline and an Azure logic app

Overall Flow Summary

GitHub | SQL Database → ADF → Azure Data Lake Storage Gen 2 → Databricks → Power BI.



Architecture Diagram

The Medallion Architecture, a pattern adopted within Data Engineering, was implemented for this project. It comprises three distinct layers:

- Bronze layer: This layer holds the raw, source-system data.
- Silver layer: This layer stores validated, cleaned and harmonised data
- Gold layer: This layer contains refined, aggregated and feature-engineered data.

Architecture Overview

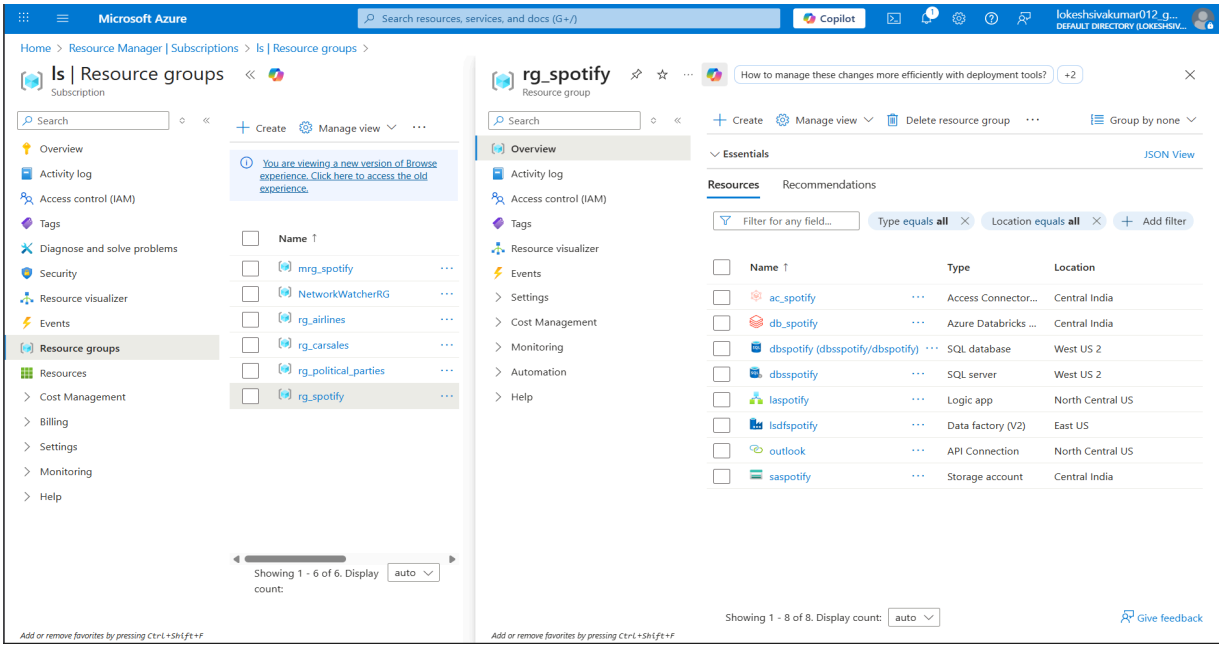
The following components facilitate the pipeline:

- GitHub: Used to store and manage the raw source data.
- SQL Databases: The raw data is initially copied into the local SQL database for staging.
- Azure Data Factory: ADF is used to ingest data into the bronze layer.
- Azure Data Lake Storage Gen 2: This is where all our data gets stored.
- Azure Databricks: Data transformation and refinement occur here, utilising PySpark.
- Azure Key Vault: This service enhances security and secures access credentials.

Implementation Details

Environment

Microsoft offers a free 30-day trial period to access its services. I have created the following resource group named rg_spotify. Within it, I have used the required services, namely Azure Data Factory, Azure Storage Account, Azure Data Lake Storage Gen 2, Azure Databricks, Azure Databricks Access Connector and the Logic app as shown in the image below.



Azure Services

Data Source

We needed a place to hold our raw data before starting the pipeline. I chose a SQL database for this step. From here, our data is ingested into our bronze layer. So I created a username and password for secure access to our SQL database.

Data Ingestion Using Azure Data Factory

Connecting SQL Server and Azure Data Factory

ADF needed a way to talk to our SQL Server database. They didn't have a connection yet. To fix this, I created a Linked Service to act as a secure bridge between ADF and our SQL database, along with the Dataset so that ADF know exactly which table to read.

Data Ingestion

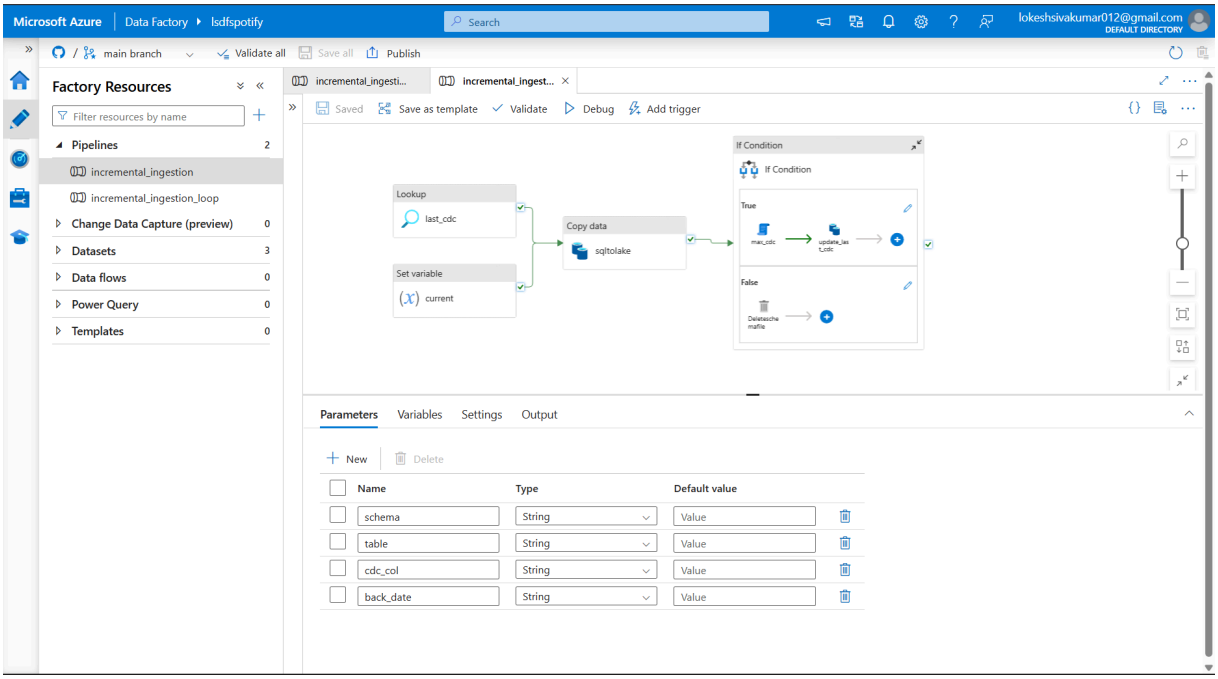
I have used the following activities in the ADF to build a flexible pipeline with a parameterisation approach.

- LookUp: To find out the last time we ran the pipeline.
- Set variable: To get the current date using the utcnow function.
- Copy data: From SQL into the Bronze layer.
- If Condition: To check if new data was read, if then
- Script: To get the max value of the last updated cdc column (date), and
- Copy data: To update the last updated date from ingestion.
- Delete: To remove the temporary files.

Tracking New Data

We need a way to remember the last time we loaded data. I also added a back date function. This allows us to re-run data from a past day if needed. I used two files for this:

1. cdc.json: Stores the date of the last successful data load.
2. empty.json: Helps in interchange of the last ingested date.



Incremental Ingestion Pipeline

To make this incremental pipeline work for many tables, not just one. I put the pipeline inside a ForEach Activity, which helps the process loop and repeat the steps.

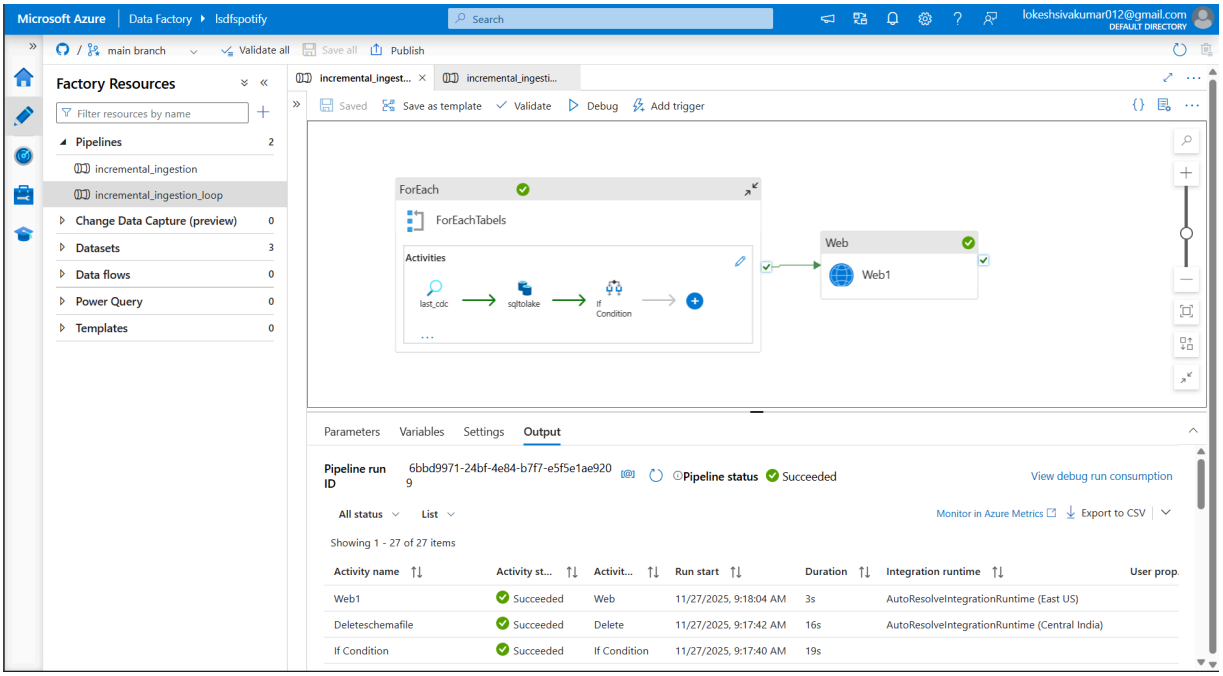
Smart Looping

- schema and Table: Tells the pipeline exactly where to find the data.
- cdc_col: to check for the latest change.
- back_date: allows us to rerun old data if needed.

The above parameters help our loop to work effectively. The details for every table are given in an array.

Scheduling

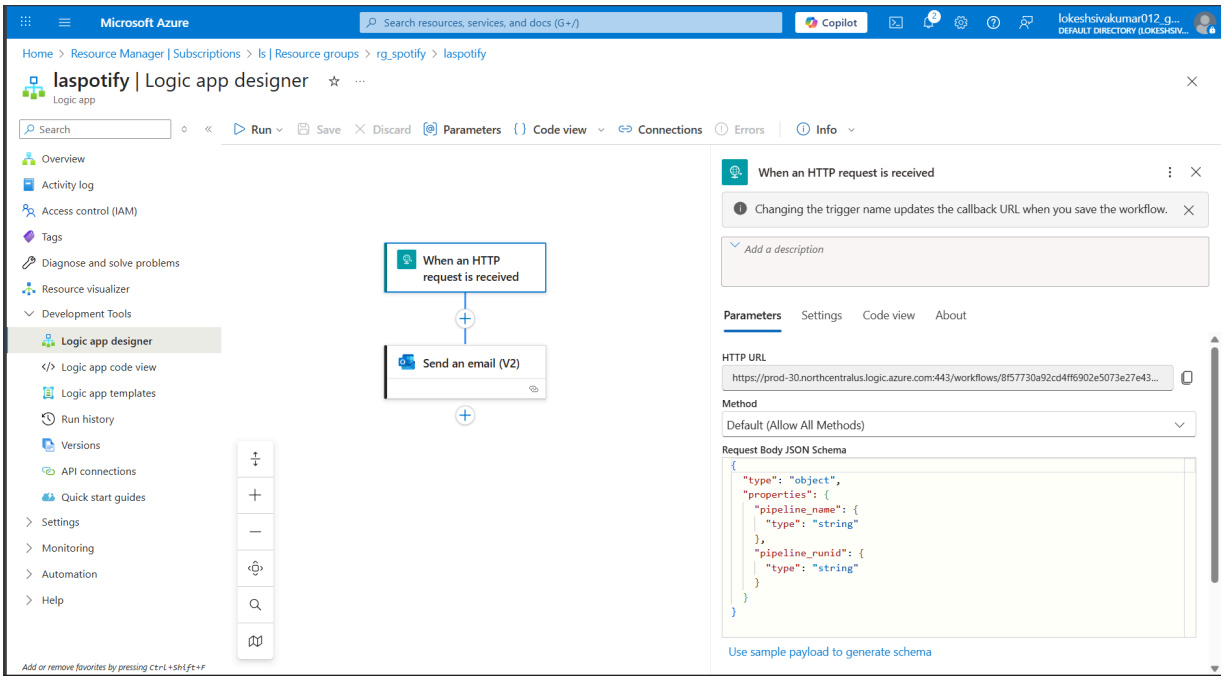
After setting up the pipeline, I used the Debug option to run a test. Next, I tested it with the incremental date using an Add Trigger option. Every time it runs, it fetches data from the SQL database. This keeps our Bronze data fresh.



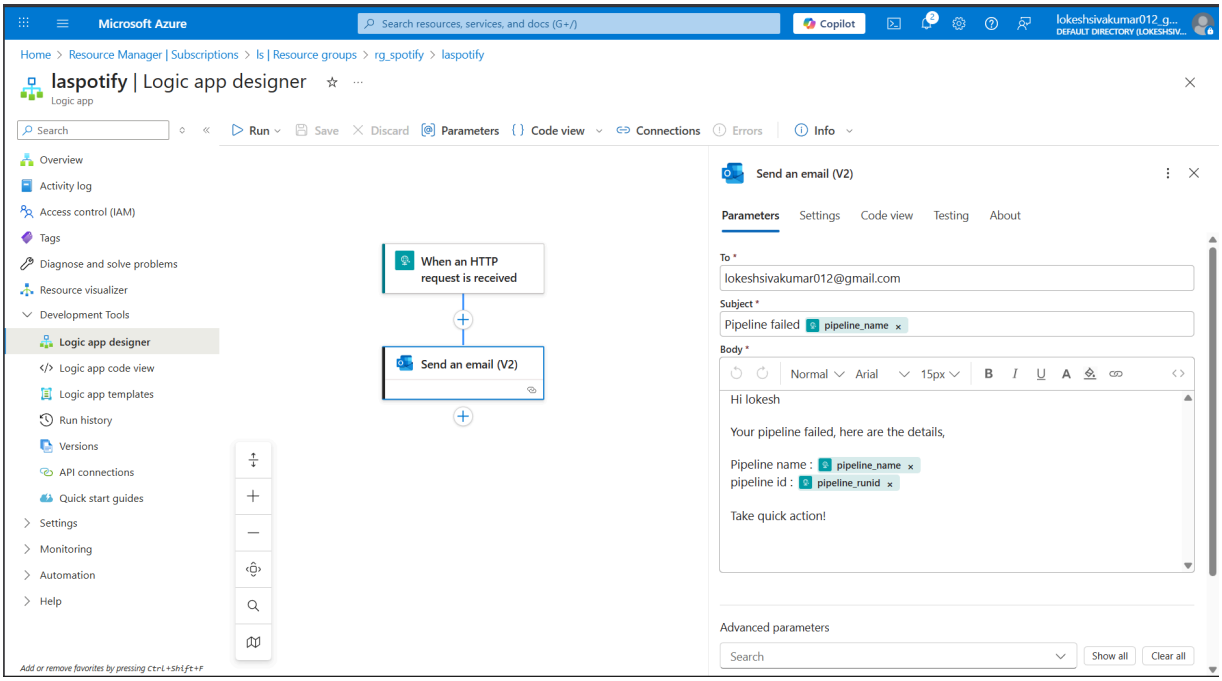
Incremental Ingestion Loop Pipeline

The raw data gets successfully moved from the SQL database into our Bronze layer. The raw data is stored in the following five tables: DimUser, DimArtist, DimAlbum, DimDate, and FactStream.

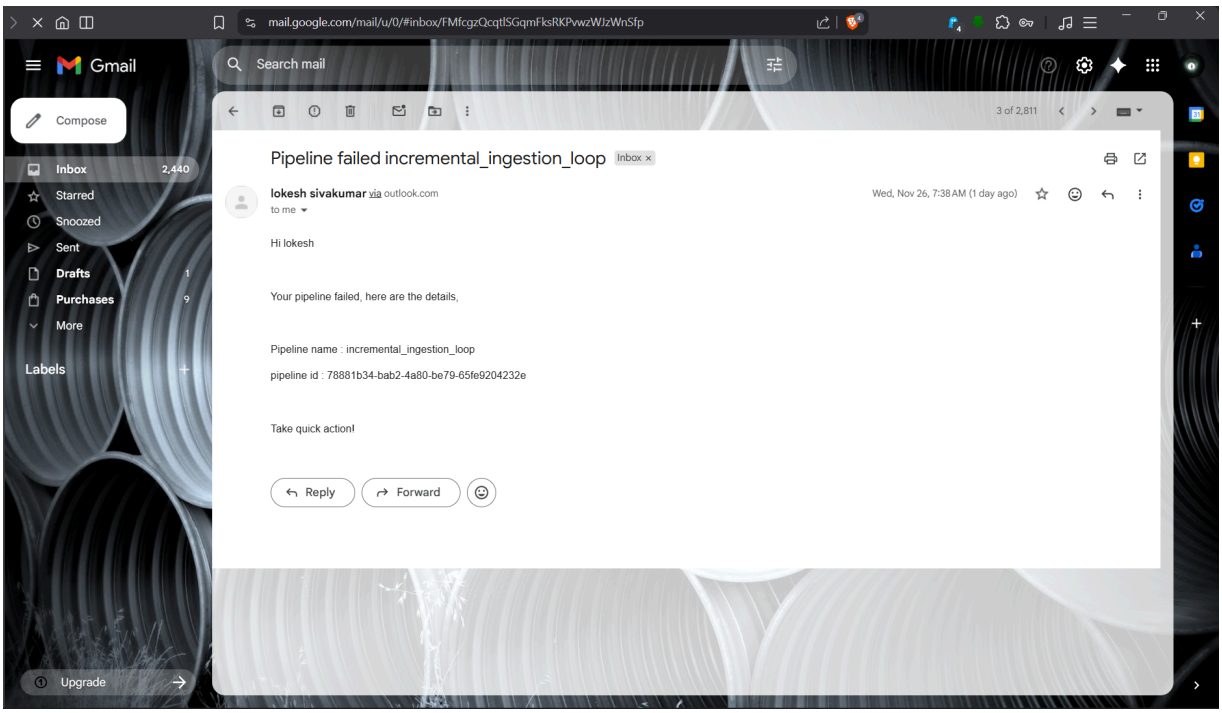
Getting Alerts



Logic App Designer



Logic App Designer



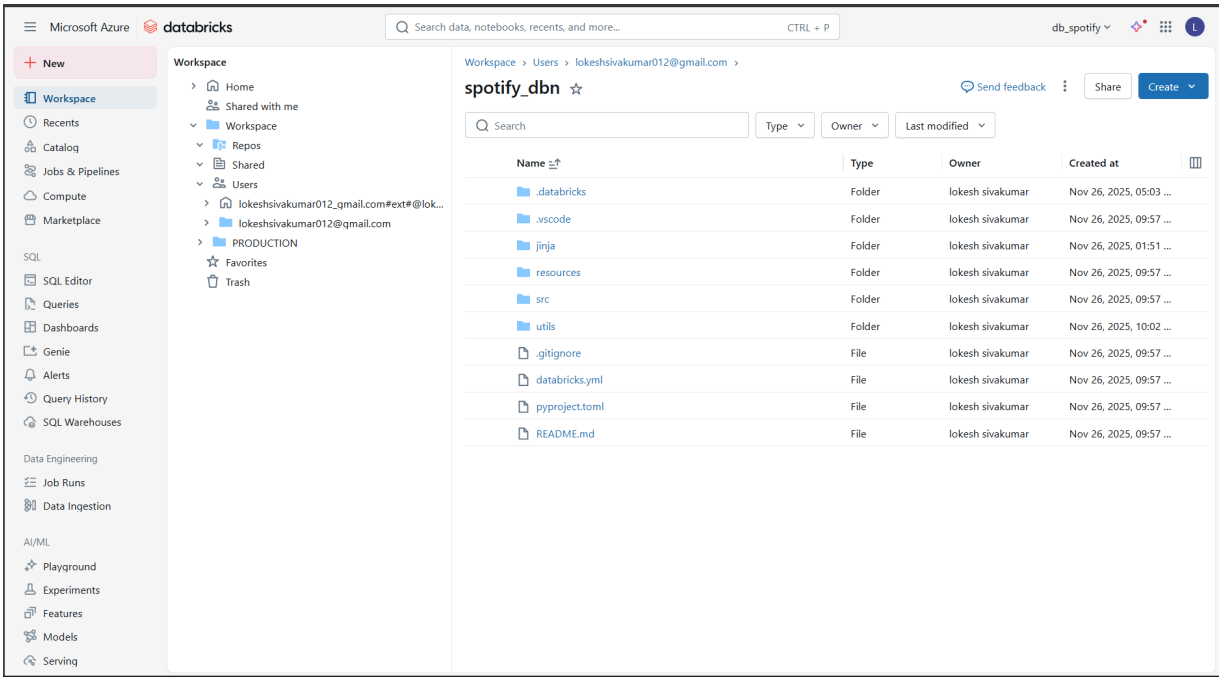
Alert Email

To get alerts, I added a Web Activity, a simple messenger. If our pipeline throws an error, the Web Activity talks to our Logic App, and it will send us an alert immediately. The images above show this workflow.

Data Transformation using Azure Databricks

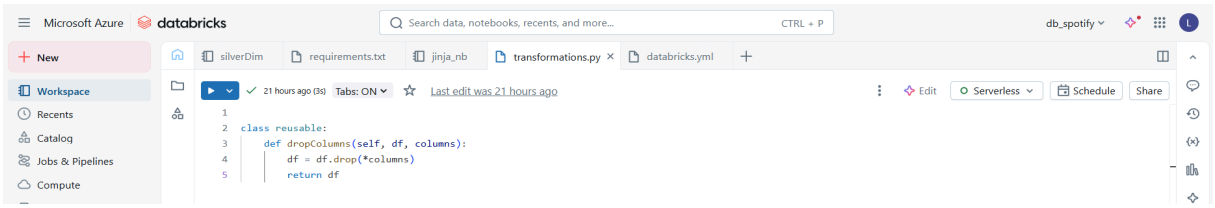
I have leveraged several tools here:

- Databricks Unity Catalog: This helps us manage and secure all our data and tables. It gives us one place to control everything.
- Databricks Access Connector: It uses a Managed Identity, a secure, password-less way for services to connect. It makes our Databricks securely read and write data to ADLS Gen2 without a secret key.
- Spark Streaming with Autoloader: This is the automatic way to handle new data as soon as it arrives. Autoloader watches the files and starts the process automatically.
- Pyspark Transformation: To write all the code for cleaning, validating, and shaping the data.
- Metadata-Driven pipeline: I built a flexible pipeline to create PySpark code using metadata and Jinja2. It helps in handling the dynamic needs of our business requirements without repeating the same process.
- Databricks Asset Bundle: DAB acts like a master blueprint for our Databricks work. It packages all our files and settings together. This helps us quickly deploy our entire project to a new testing or production environment.



Databricks Silver Layer

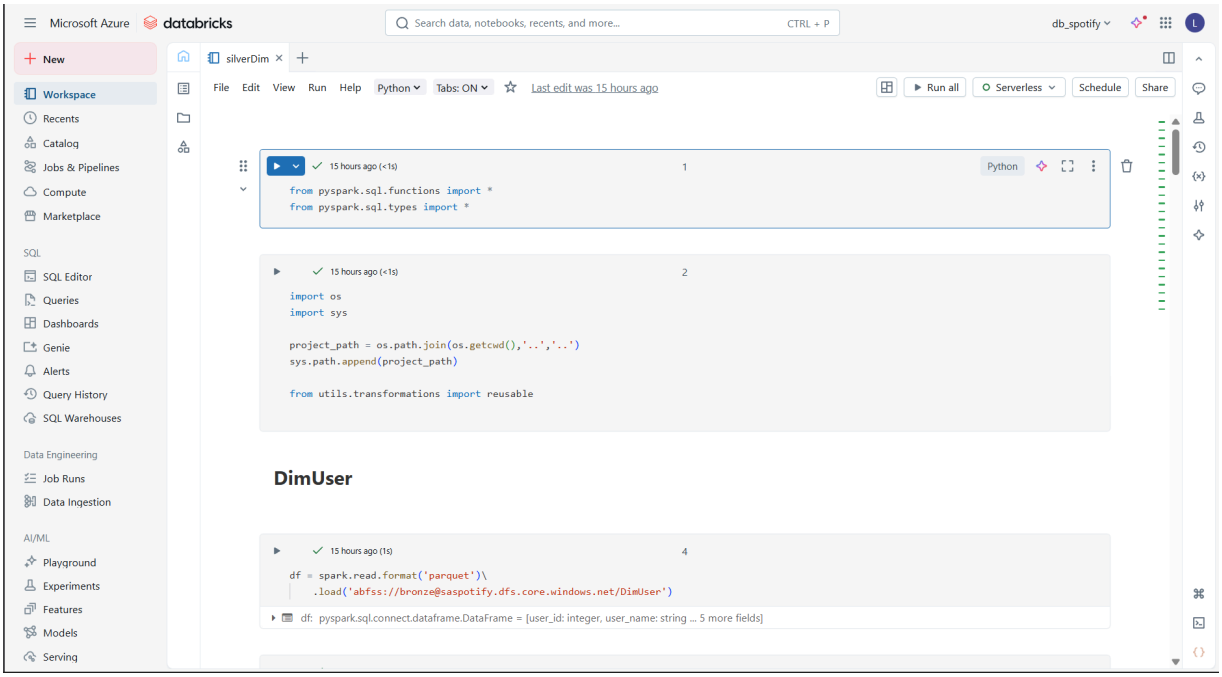
The above image shows our entire project folder using DAB. Inside, I have built a reusable Python class. This class holds our code for cleaning the data frame. By this, we can use this class anytime we need to clean a dataset. This helps us to remove repeated lines of code for different datasets. It makes our code simple, clean and very efficient.



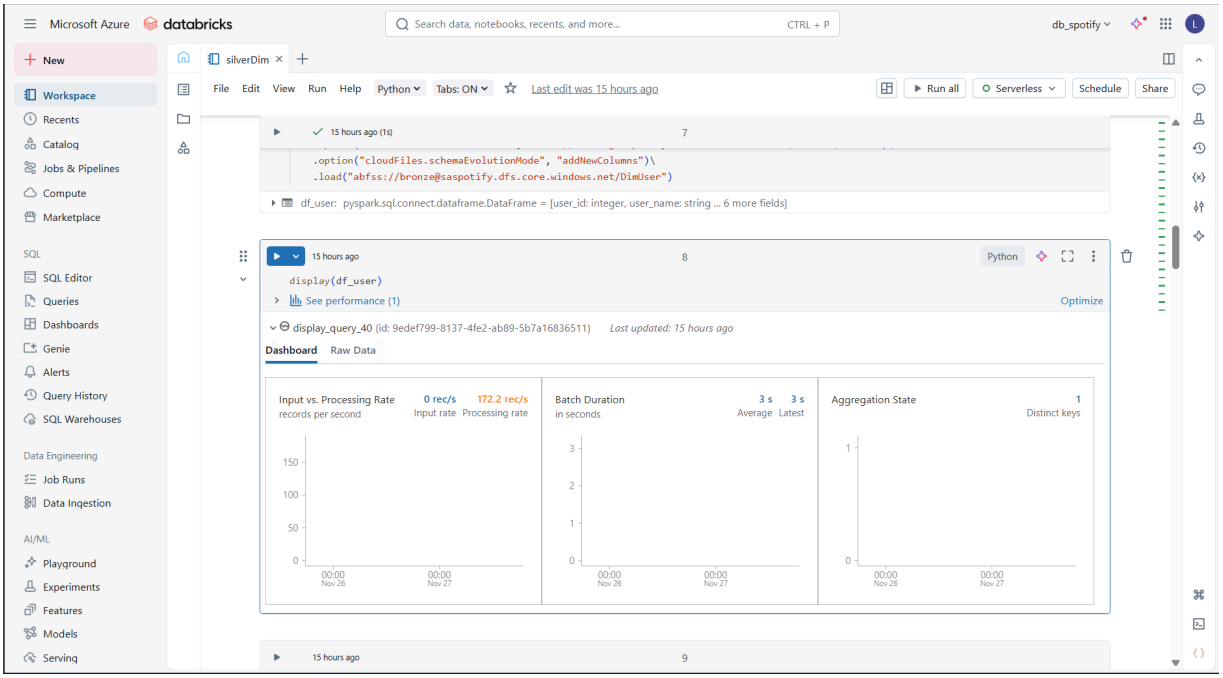
Class reusable

The DimUser table

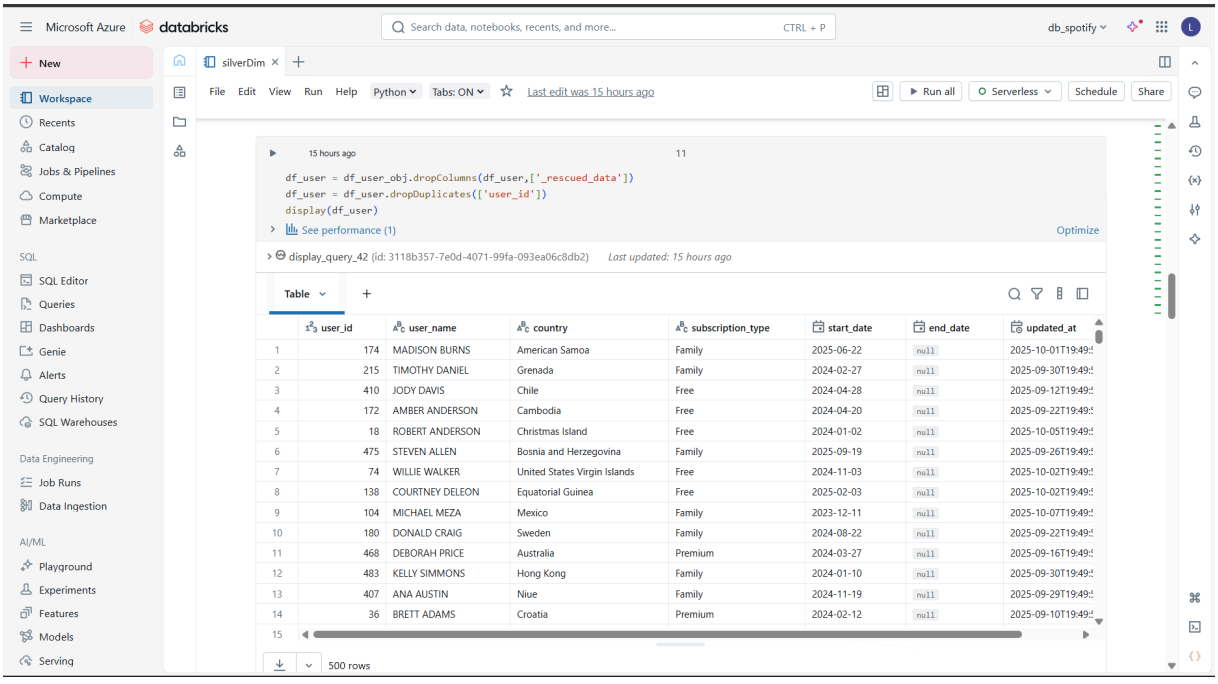
The raw data from the Bronze layer, which we read using the cloudFiles method. This allows us to use streaming for efficiency. Then I performed a simple transformation, converted the user names to UPPER CASE. Dropped the duplicated records in the user_id column. The streaming method automatically added a column called _rescued_data, which was not needed. So I used our reusable Python class to drop that column. Finally, then saved it into our Silver layer. I have used the Delta Table, which is great for reliable, high-quality data.

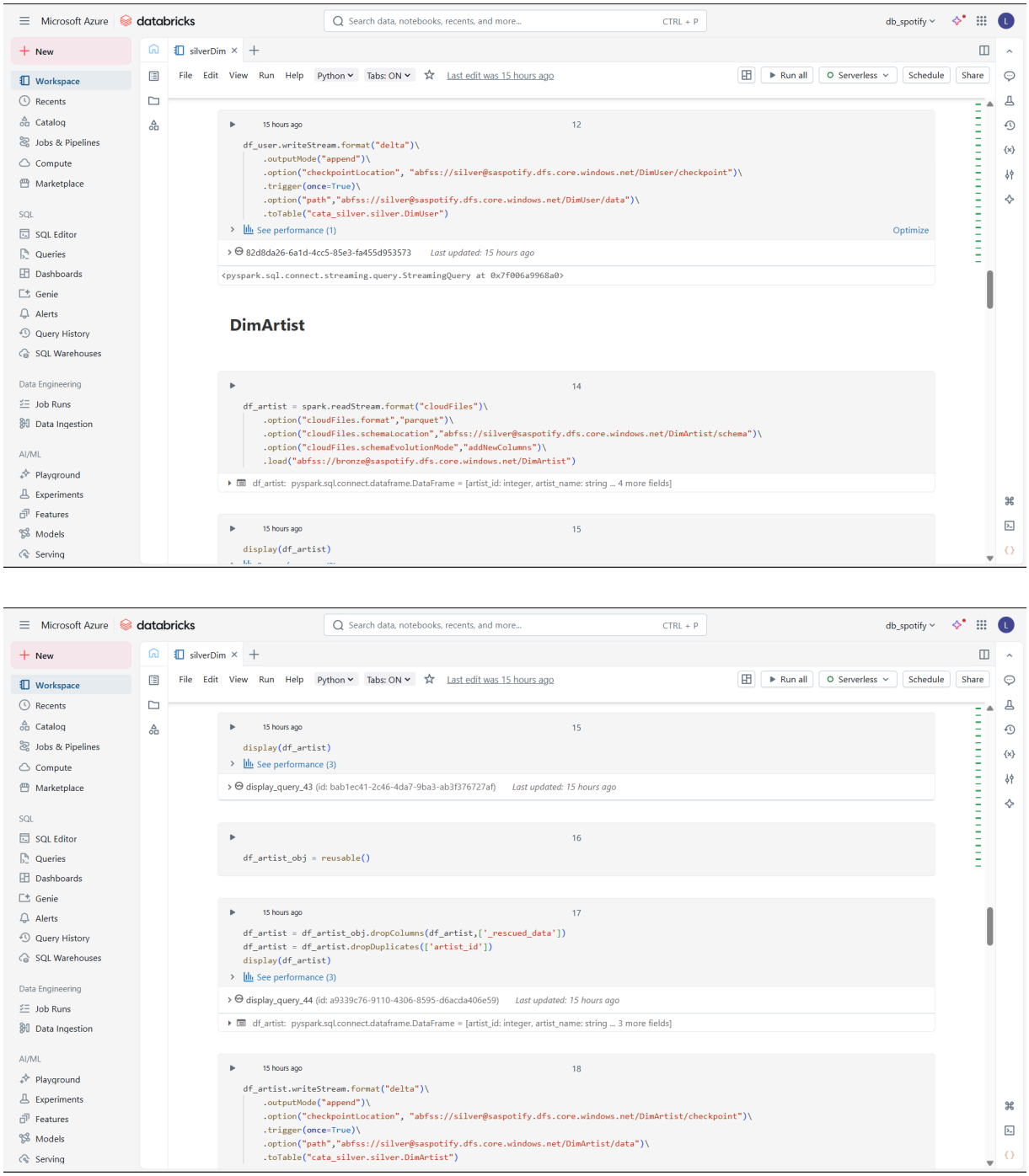


DimUser


Microsoft Azure
databricks
Search data, notebooks, recents, and more...
CTRL + P
db_spotify
New
silverDim
File Edit View Run Help Python Tabs: ON
Last edit was 15 hours ago
Run all
Serverless
Schedule
Share
Workspace
Recents
Catalog
Jobs & Pipelines
Compute
Marketplace
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
AI/ML
Playground
Experiments
Features
Models
Serving
15 hours ago (1s)
7
.option("cloudFiles.schemaEvolutionMode", "addNewColumns")\
.load("abfss://bronze@saspotify.dfs.core.windows.net/DimUser")
df_user: pyspark.sql.connect.dataframe.DataFrame = [user_id: integer, user_name: string ... 6 more fields]
15 hours ago
8
Python
display(df_user)
See performance (1)
Optimize
display_query_40 (id: 9edef799-8137-4fe2-ab89-5b7a16836511)
Last updated: 15 hours ago
Dashboard
Raw Data
Input vs. Processing Rate
records per second
0 rec/s
Input rate
172.2 rec/s
Processing rate
Batch Duration
in seconds
3 s
Average
3 s
Latest
Aggregation State
1
Distinct keys
150
100
50
0
3
2
1
0
00:00
Nov 26
00:00
Nov 27
15 hours ago
9


Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
AI/ML
Playground
Experiments
Features
15 hours ago
9
df_user = df_user.withColumn("user_name",upper(col("user_name")))
display(df_user)
See performance (1)
Optimize
display_query_41 (id: ef8b930c-2767-4c3c-a308-a494feaf69e9)
Last updated: 15 hours ago
df_user: pyspark.sql.connect.dataframe.DataFrame = [user_id: integer, user_name: string ... 6 more fields]
10
df_user_obj = reusable()

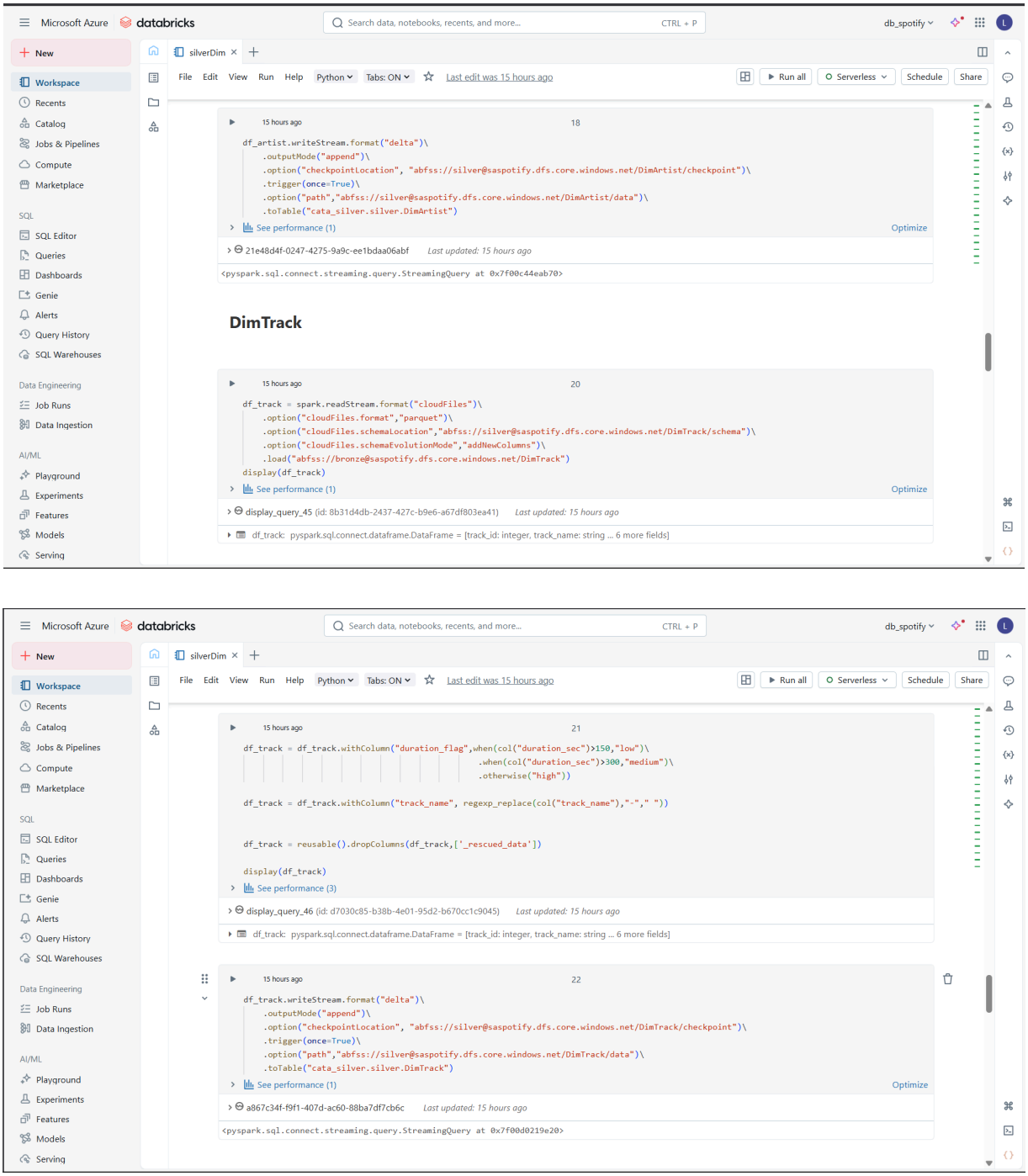

Microsoft Azure
databricks
Search data, notebooks, recents, and more...
CTRL + P
db_spotify
New
silverDim
File Edit View Run Help Python Tabs: ON
Last edit was 15 hours ago
Run all
Serverless
Schedule
Share
Workspace
Recents
Catalog
Jobs & Pipelines
Compute
Marketplace
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
AI/ML
Playground
Experiments
Features
Models
Serving
15 hours ago
11
df_user = df_user_obj.dropColumns(df_user,['_rescued_data'])
df_user = df_user.dropDuplicates(['user_id'])
display(df_user)
See performance (1)
Optimize
display_query_42 (id: 3118b357-7e0d-4071-99fa-093ea06c8db2)
Last updated: 15 hours ago
Table
user_id
user_name
country
subscription_type
start_date
end_date
updated_at
1 174 MADISON BURNS American Samoa Family 2025-06-22 null 2025-10-01T19:49:
2 215 TIMOTHY DANIEL Grenada Family 2024-02-27 null 2025-09-30T19:49:
3 410 JODY DAVIS Chile Free 2024-04-28 null 2025-09-12T19:49:
4 172 AMBER ANDERSON Cambodia Free 2024-04-20 null 2025-09-22T19:49:
5 18 ROBERT ANDERSON Christmas Island Free 2024-01-02 null 2025-10-05T19:49:
6 475 STEVEN ALLEN Bosnia and Herzegovina Family 2025-09-19 null 2025-09-26T19:49:
7 74 WILLIE WALKER United States Virgin Islands Free 2024-11-03 null 2025-10-02T19:49:
8 138 COURTNEY DELEON Equatorial Guinea Free 2025-02-03 null 2025-10-02T19:49:
9 104 MICHAEL MEZA Mexico Family 2023-12-11 null 2025-10-07T19:49:
10 180 DONALD CRAIG Sweden Family 2024-08-22 null 2025-09-22T19:49:
11 468 DEBORAH PRICE Australia Premium 2024-03-27 null 2025-09-16T19:49:
12 483 KELLY SIMMONS Hong Kong Family 2024-01-10 null 2025-09-30T19:49:
13 407 ANA AUSTIN Niue Family 2024-11-19 null 2025-09-29T19:49:
14 36 BRETT ADAMS Croatia Premium 2024-02-12 null 2025-09-10T19:49:
15
500 rows



DimArtist

The DimArtist table

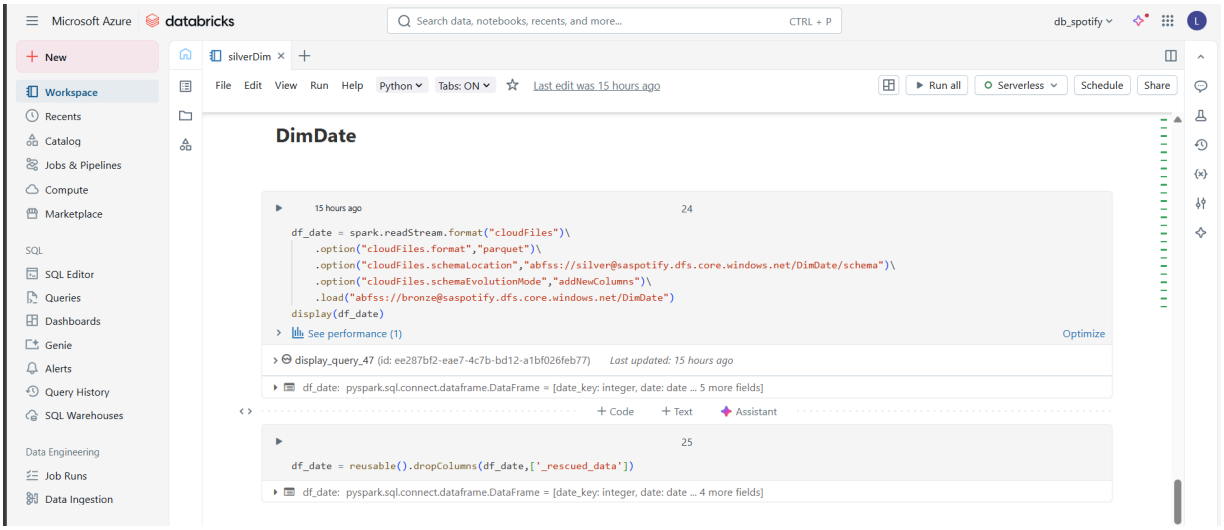
I followed the same steps to read the data from the Bronze layer, dropping duplicate records in the artist_id column and dropping _rescued_data by using our reusable Python class. Finally, then saved it into our Silver layer as a Delta Table.



DimTrack

The DimTrack table

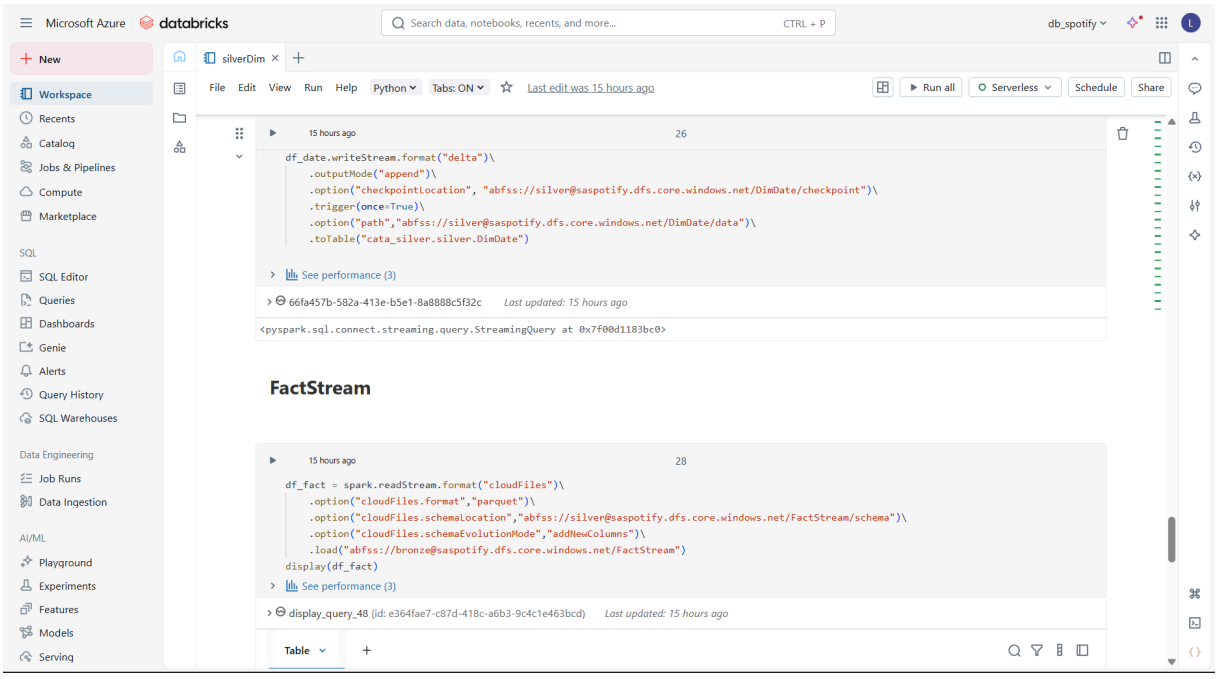
I followed the same steps to read the data from the Bronze layer and drop _rescued_data by using our reusable Python class. Created a duration_flag column using the withColumn function and marked it as low, medium and high. Then, with regexp_replace, I have replaced a hyphen with a space. Finally, then saved it into our Silver layer as a Delta Table.



DimDate

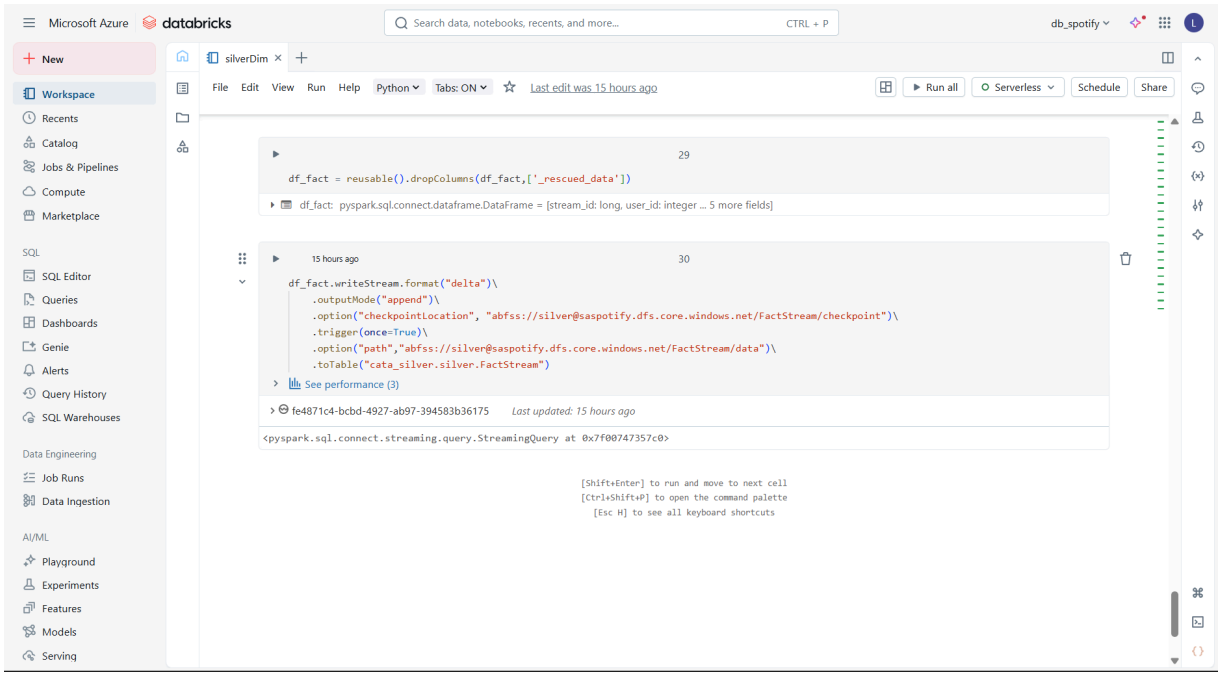
The DimDate table

I followed the same steps to read the data from the Bronze layer and drop _rescued_data by using our reusable Python class. Finally, then saved it into our Silver layer as a Delta Table.



The FactStream table

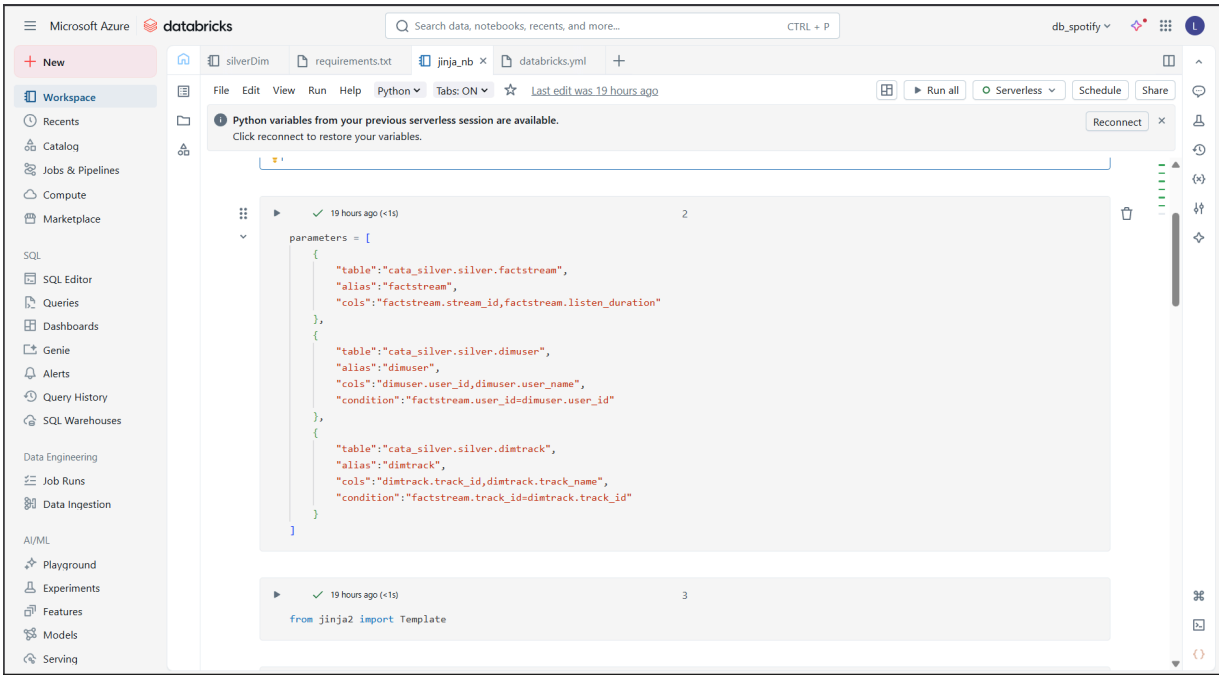
I followed the same steps to read the data from the Bronze layer and drop _rescued_data by using our reusable Python class. Finally, then saved it into our Silver layer as a Delta Table.



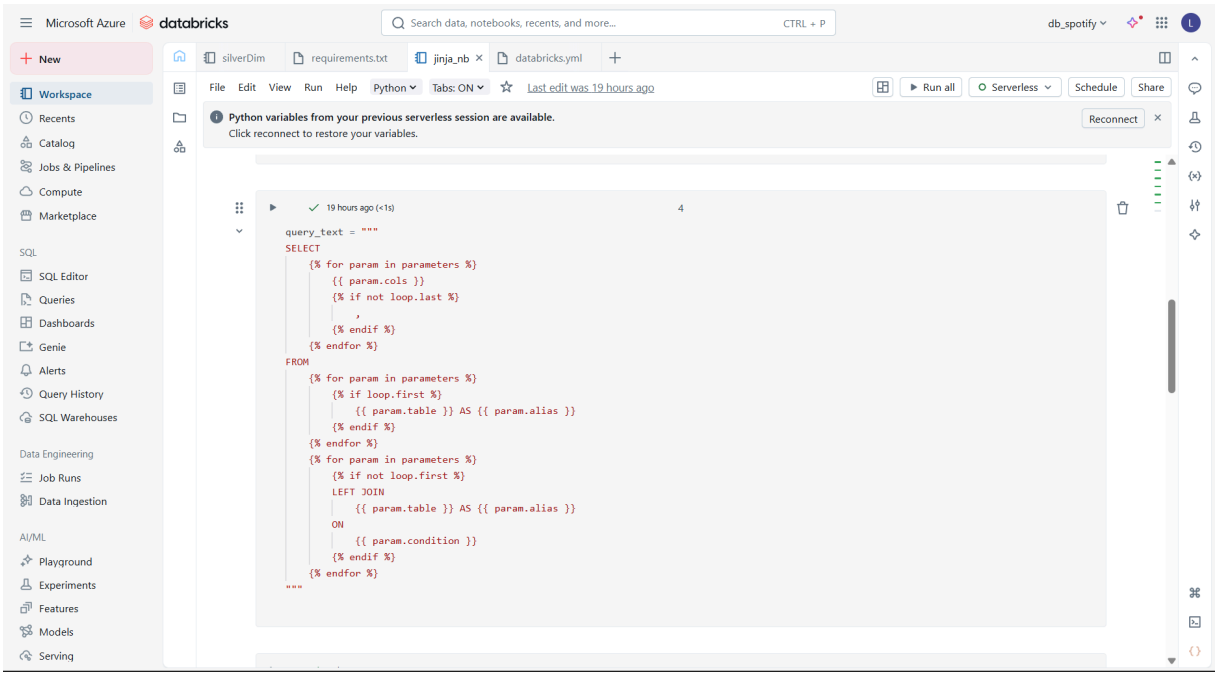
FsctStream

Building a dynamic pipeline with Jinja2

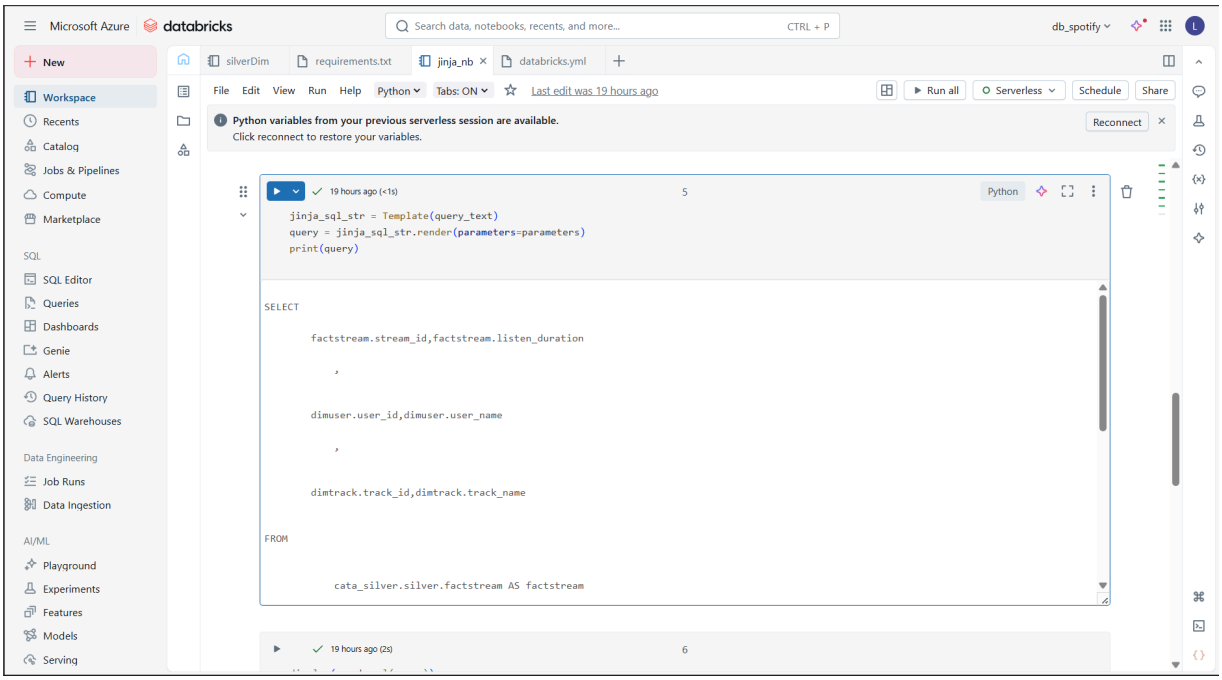
This is to build a dynamic pipeline to serve the dynamic requirements of our stakeholders in an expedited manner.



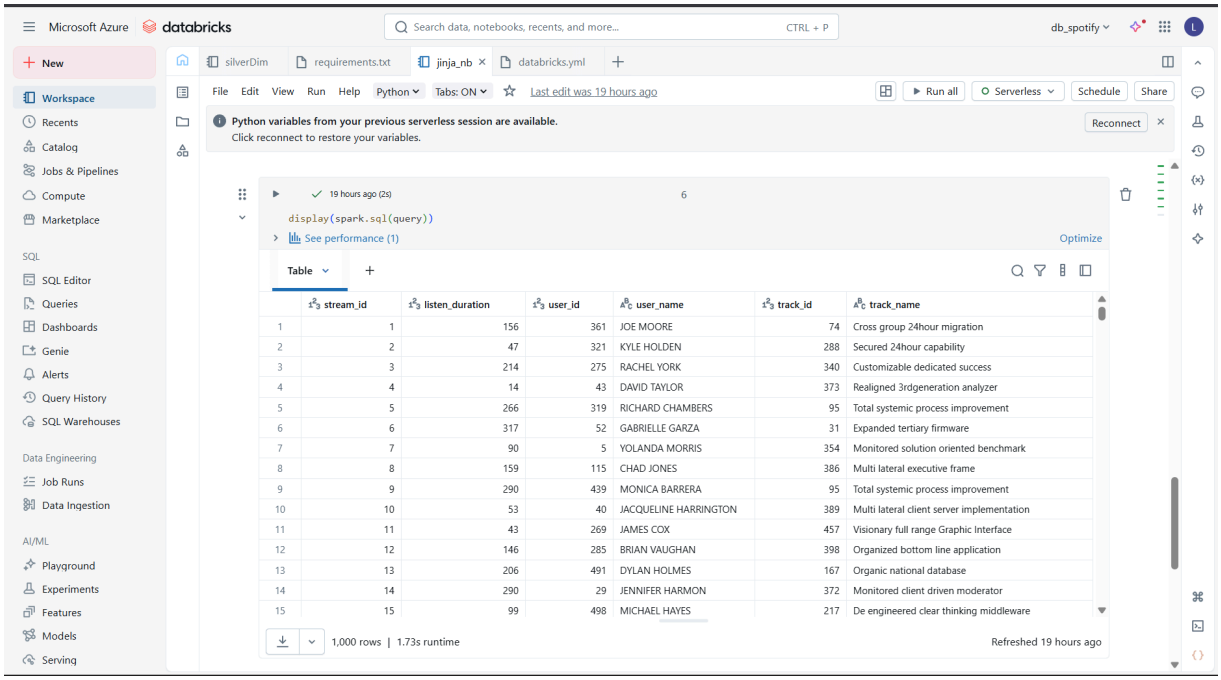
Jinja


Microsoft Azure
databricks
Search data, notebooks, recents, and more...
CTRL + P
db_spotify
New
Workspace
Recents
Catalog
Jobs & Pipelines
Compute
Marketplace
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
AI/ML
Playground
Experiments
Features
Models
Serving
silverDim
requirements.txt
jinja_nb
databricks.yml
File Edit View Run Help
Python
Tabs: ON
Last edit was 19 hours ago
Run all
Serverless
Schedule
Share
Python variables from your previous serverless session are available.
Click reconnect to restore your variables.
Reconnect
19 hours ago (<1s)
4
query_text = """
SELECT
{% for param in parameters %}
{{ param.cols }}
{% if not loop.last %}
,
{% endif %}
{% endfor %}
FROM
{% for param in parameters %}
{% if loop.first %}
{{ param.table }} AS {{ param.alias }}
{% endif %}
{% endfor %}
{% for param in parameters %}
{% if not loop.first %}
LEFT JOIN
{{ param.table }} AS {{ param.alias }}
ON
{{ param.condition }}
{% endif %}
{% endfor %}
"""

Jinja


Microsoft Azure
databricks
Search data, notebooks, recents, and more...
CTRL + P
db_spotify
New
Workspace
Recents
Catalog
Jobs & Pipelines
Compute
Marketplace
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
AI/ML
Playground
Experiments
Features
Models
Serving
silverDim
requirements.txt
jinja_nb
databricks.yml
File Edit View Run Help
Python
Tabs: ON
Last edit was 19 hours ago
Run all
Serverless
Schedule
Share
Python variables from your previous serverless session are available.
Click reconnect to restore your variables.
Reconnect
19 hours ago (<1s)
5
Python
jinja_sql_str = Template(query_text)
query = jinja_sql_str.render(parameters=parameters)
print(query)
SELECT
factstream.stream_id,factstream.listen_duration
,
dimuser.user_id,dimuser.user_name
,
dimtrack.track_id,dimtrack.track_name
FROM
cata_silver.silver.factstream AS factstream
19 hours ago (2s)
6

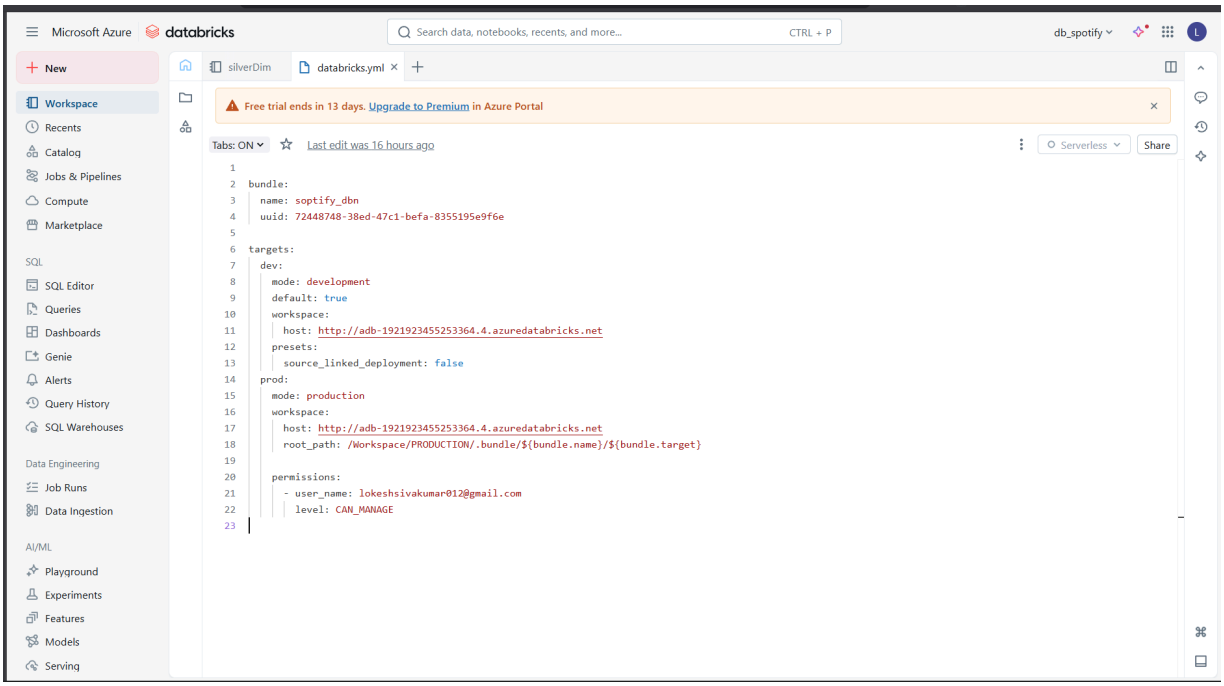
Jinja



Jinja

Databricks Asset Bundle

It is used for a rapid deployment of the entire project across diverse target environments, such as testing and production. The image below depicts its simple implementation.



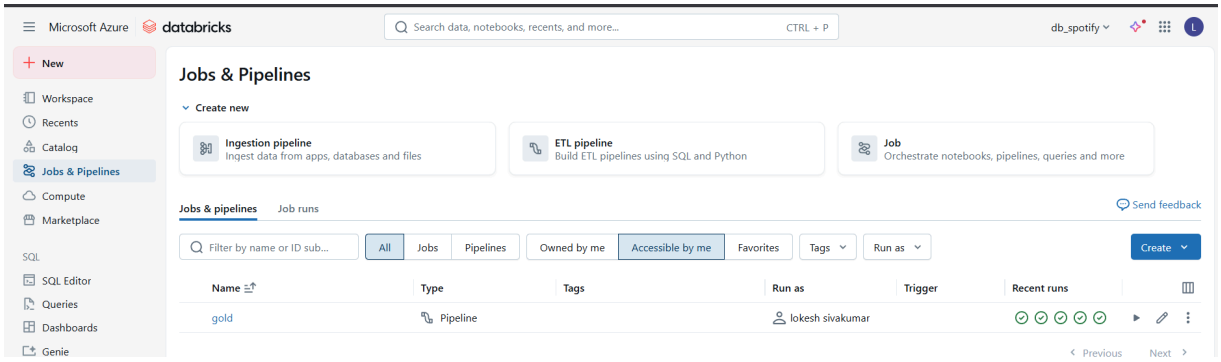
Yml

Gold layer ETL

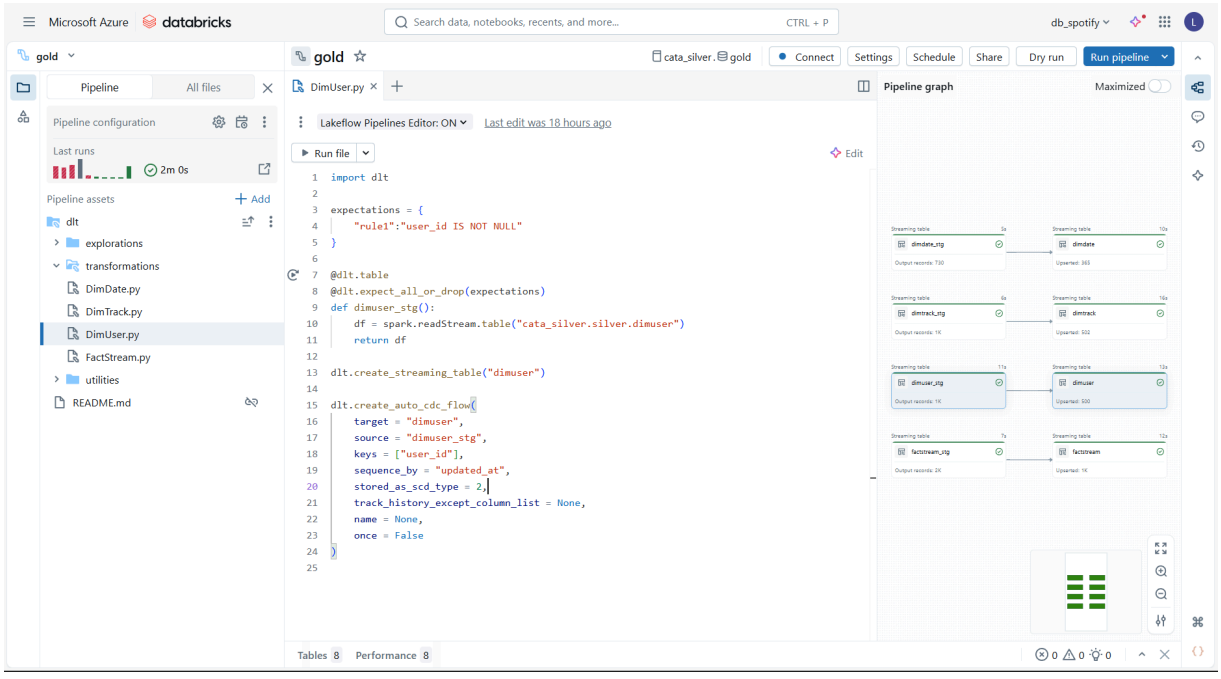
This layer contains the final optimised data for our downstream purpose.

Delta Live Tables

I utilised DLT to construct the Gold Layer pipeline. It automates infrastructure management, dependencies and execution, simplifying the ETL process. It also allowed us to define exceptions and constraints on the data, ensuring high data quality before the data is served.



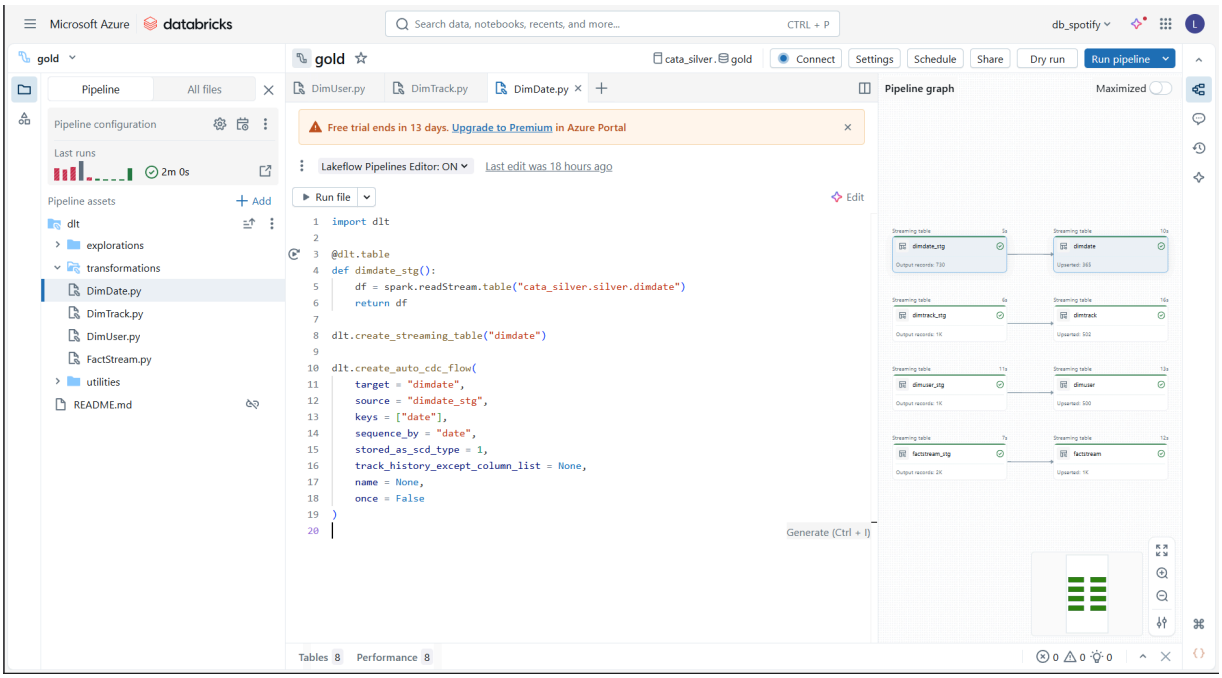
Gold layer



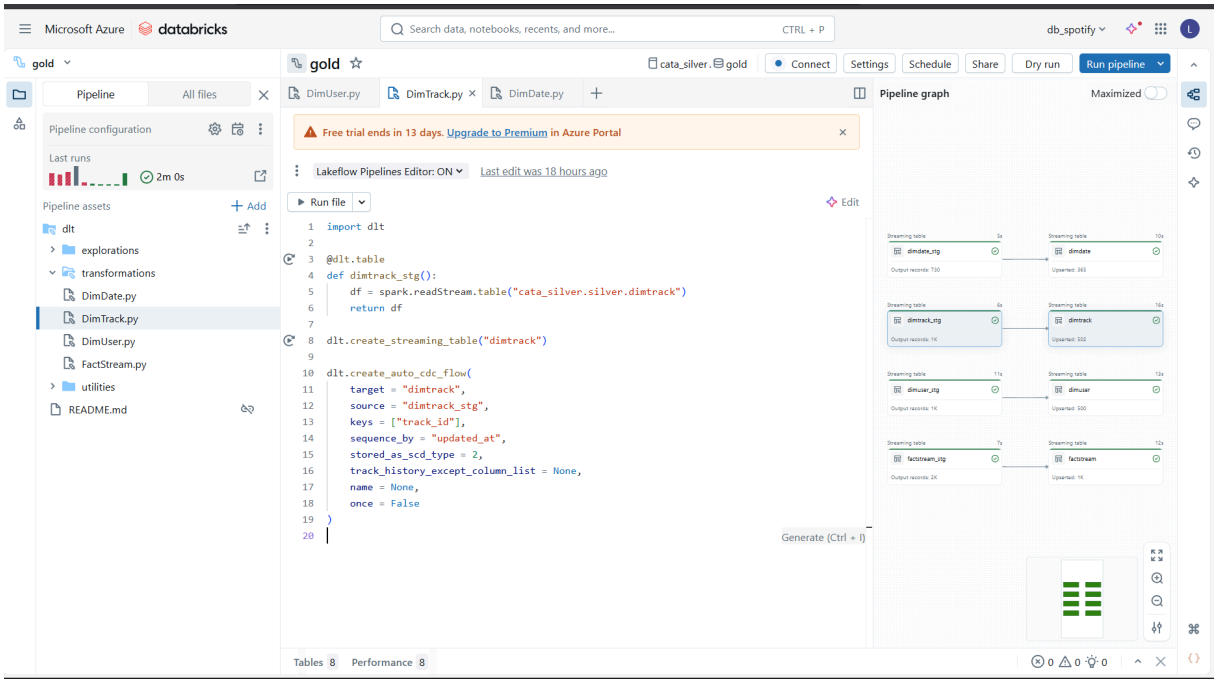
DimUser

Slowly Changing Dimensions

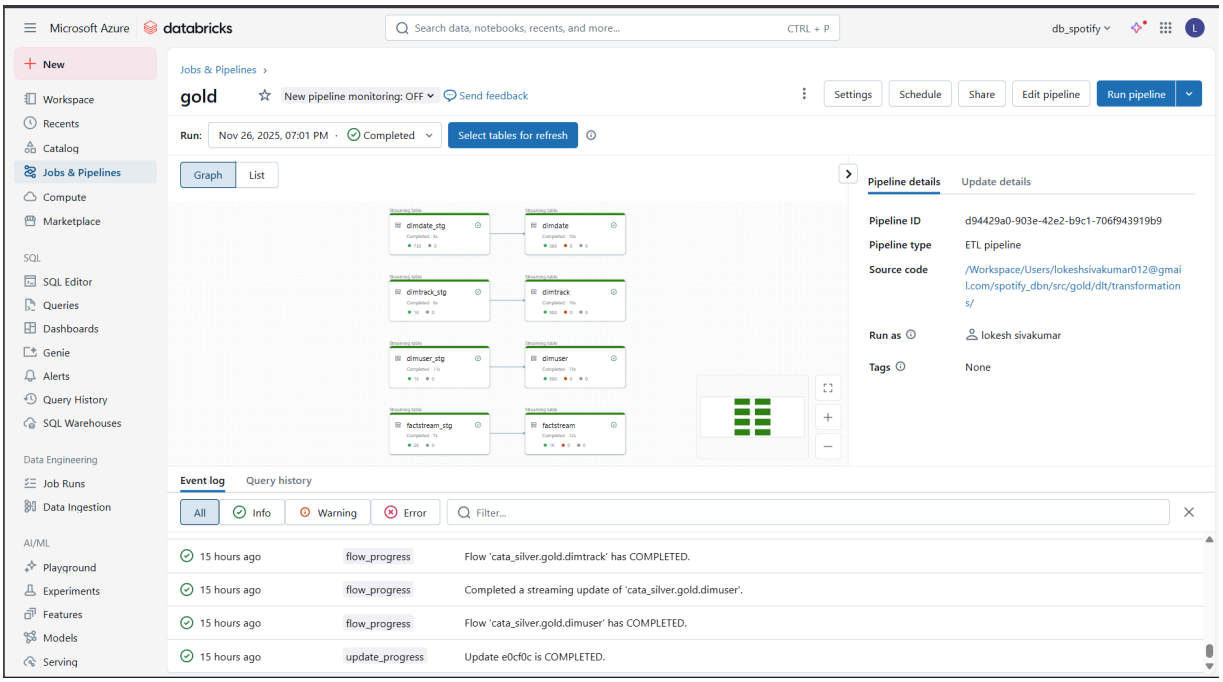
It is essential for tracking changes to dimensional attributes over time. This ensures that historical reports reflect the data as it was at the time of the event, providing accurate historical analysis for our downstream team.



DimDate



DimTrack



Gold Pipeline

Pipeline Testing

I have tested this pipeline with the incremental data set, and it successfully proved its effectiveness by incorporating the new data without any errors.

Serving the Data

Once the data reaches the Gold layer, it is ready to be served to the downstream process team, which includes the Data Analyst. With this, I have ensured that our pipeline is industry-level and production-ready.

Challenges and Solutions

- Handling Large Dataset → Processed using Pyspark for scalability.
- Missing or Duplicate Records → Handled using deduplication techniques.
- Schema Evolution → Implemented Delta Lake for versioning.

Conclusion

I built this scalable, cloud-based, enterprise-ready pipeline from scratch, directly facing and solving real-world challenges, and now I am ready to deliver a production-level pipeline with real business insights.