

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

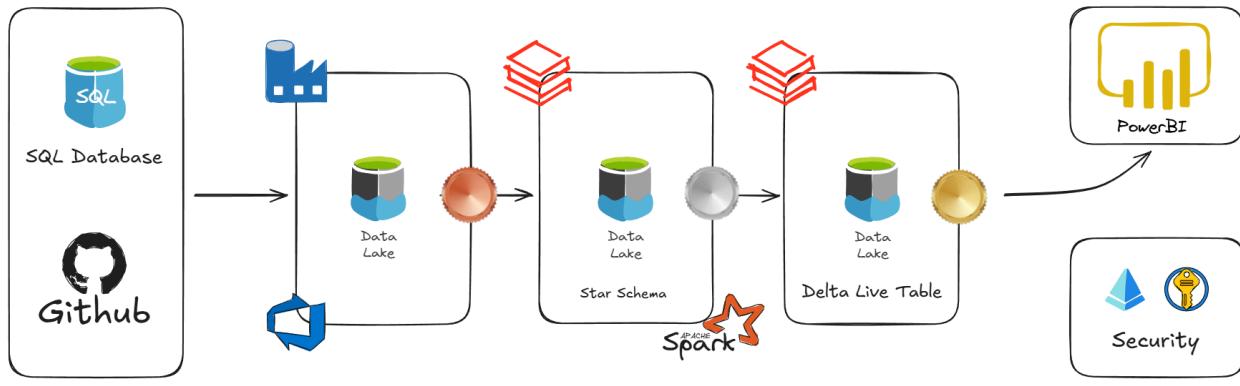
- Following the transformation, the resulting model is served directly.
- Based on the downstream usage, various reporting tools can then be utilised to generate comprehensive reports and dashboards.

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.

The screenshot shows the Microsoft Azure Resource Manager interface. On the left, the 'Is | Resource groups' page is visible, showing a list of resource groups like 'mrg_spotify', 'NetworkWatcherRG', 'rg_airlines', 'rg_carsales', 'rg_political_parties', and 'rg_spotify'. On the right, the 'rg_spotify' resource group details page is shown, featuring an 'Overview' section and a 'Resources' table. The 'Resources' table lists various Azure services with columns for Name, Type, and Location. The services listed include 'ac_spotify' (Access Connector), 'db_spotify' (Azure Databricks), 'dbspotify (dbsspotify/dbspotifly)' (SQL database), 'dbsspotify' (SQL server), 'laspotify' (Logic app), 'lsdfspotify' (Data factory (V2)), 'outlook' (API Connection), and 'saspotify' (Storage account). All resources are located in Central India.

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

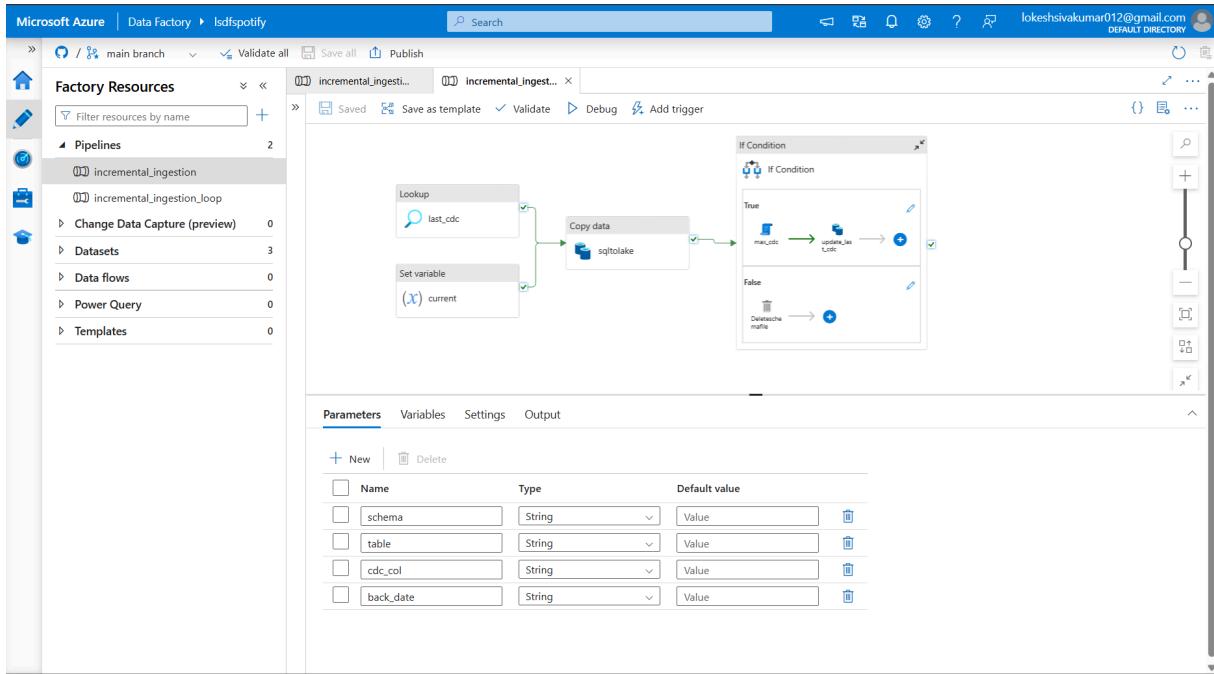
I have used the following activities in the ADF to build a flexible pipeline with a parameterisation approach.

- 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.

The screenshot shows the Microsoft Azure Data Factory pipeline editor. On the left, the 'Factory Resources' sidebar lists Pipelines, Datasets, Data flows, Power Query, and Templates. The main workspace displays a pipeline named 'incremental_ingestion'. This pipeline contains a 'ForEach' activity named 'ForEachTables' which iterates over a dataset. Inside the 'ForEachTables' loop, there is a sequence of activities: 'last_cdc' (a CDC connector), 'sqtoltake' (a copy activity), and 'Condition' (a decision activity). Following the 'Condition' activity is a 'Web' activity named 'Web1'. Below the pipeline editor, the 'Output' tab is selected, showing details about a pipeline run. The run ID is 6bbd9971-24bf-4e84-b7f7-e5f5e1ae920, and it has a status of 'Succeeded'. A table below lists the activities and their execution details.

Activity name	Activity st...	Activit...	Run start	Duration	Integration runtime	User prop.
Web1	Succeeded	Web	11/27/2025, 9:18:04 AM	3s	AutoResolveIntegrationRuntime (East US)	
Deleteschemafile	Succeeded	Delete	11/27/2025, 9:17:42 AM	16s	AutoResolveIntegrationRuntime (Central India)	
If Condition	Succeeded	If Condition	11/27/2025, 9:17:40 AM	19s		

Incremental Ingestion Loop Pipeline

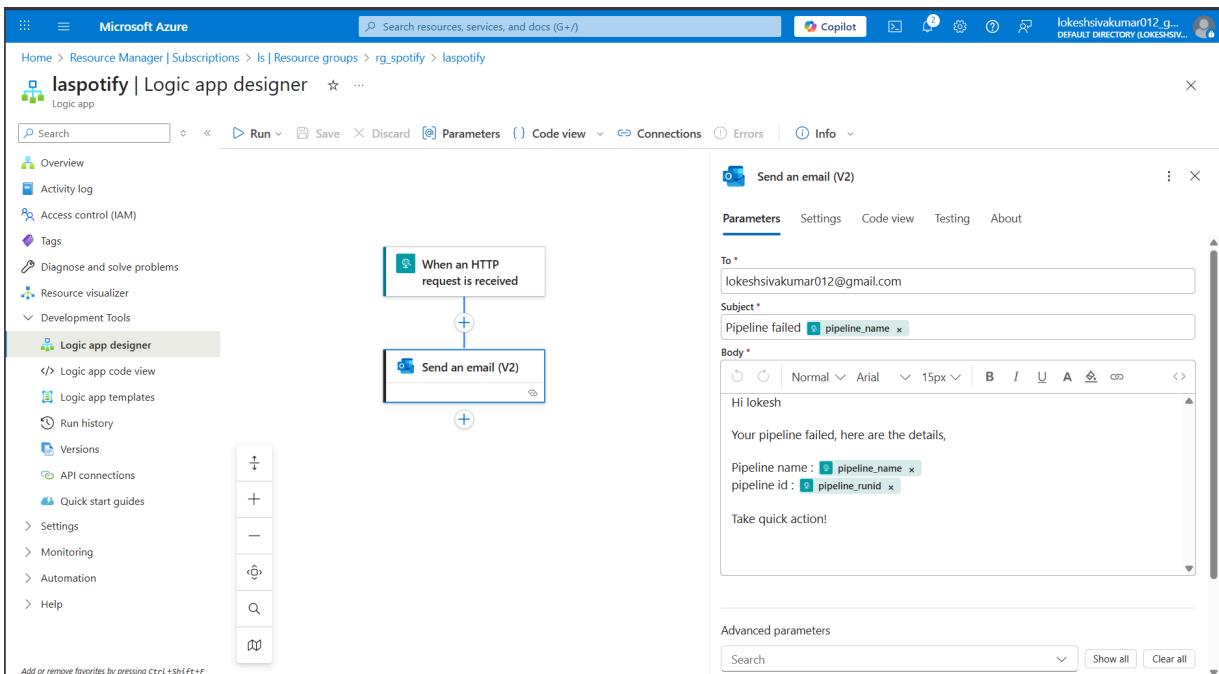
The raw data gets successfully moved from the SQL database into our Bronze layer. The raw data is stored in the following five tables: DimUser, DimArtist, DimAlbum, DimDate, and FactStream.

Getting Alerts

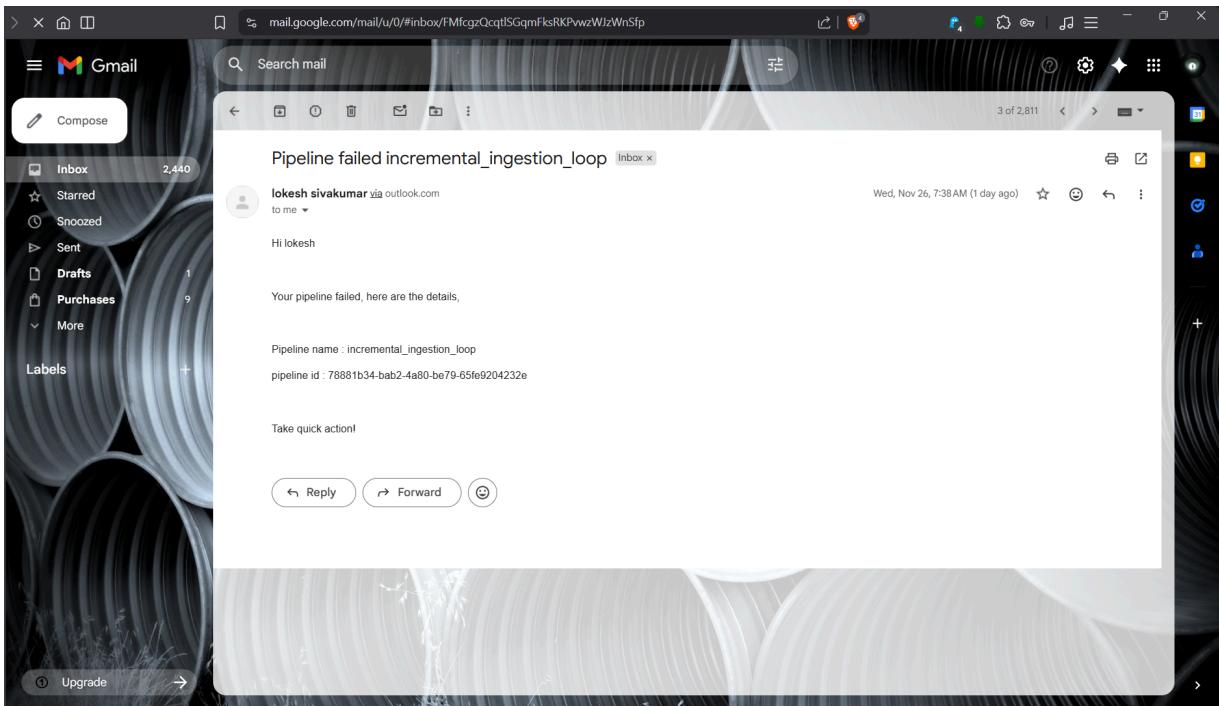
The screenshot shows the Microsoft Azure Logic App designer. On the left, the 'Logic app designer' sidebar lists Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource visualizer, Development Tools (selected), Logic app code view, Logic app templates, Run history, Versions, API connections, Quick start guides, Settings, Monitoring, Automation, and Help. The main workspace shows a logic app named 'laspotify'. It consists of two main steps: 'When an HTTP request is received' and 'Send an email (V2)'. To the right of the logic app, there is a detailed view of the 'When an HTTP request is received' step, including its parameters, settings, and code view. The 'HTTP URL' is set to https://prod-30.northcentralus.logic.azure.com:443/workflows/8f57730a92cd4ff6902e5073e27e43... and the 'Method' is set to Default (Allow All Methods). The 'Request Body JSON Schema' is displayed as a JSON object:

```
{
  "type": "object",
  "properties": {
    "pipeline_name": {
      "type": "string"
    },
    "pipeline_runid": {
      "type": "string"
    }
  }
}
```

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.

The screenshot shows the Microsoft Azure Databricks workspace interface. On the left, there is a sidebar with various navigation options: New, Workspace (selected), Recents, Catalog, Jobs & Pipelines, Compute, Marketplace, SQL, SQL Editor, Queries, Dashboards, Genie, Alerts, Query History, and SQL Warehouses. Below these are sections for Data Engineering (Job Runs, Data Ingestion) and AI/ML (Playground, Experiments, Features, Models, Serving). The main area displays a file browser for a project named 'spotify_dbn'. The browser shows a hierarchical structure of folders and files. At the top right of the browser, there are buttons for 'Send feedback', 'Share', and 'Create'. A search bar and filters for Type, Owner, and Last modified are also present. The table below lists the contents of the 'spotify_dbn' folder:

Name	Type	Owner	Created at
.databricks	Folder	lokesh sivakumar	Nov 26, 2025, 05:03 ...
.vscode	Folder	lokesh sivakumar	Nov 26, 2025, 09:57 ...
jinja	Folder	lokesh sivakumar	Nov 26, 2025, 01:51 ...
resources	Folder	lokesh sivakumar	Nov 26, 2025, 09:57 ...
src	Folder	lokesh sivakumar	Nov 26, 2025, 09:57 ...
utils	Folder	lokesh sivakumar	Nov 26, 2025, 10:02 ...
.gitignore	File	lokesh sivakumar	Nov 26, 2025, 09:57 ...
databricks.yml	File	lokesh sivakumar	Nov 26, 2025, 09:57 ...
pyproject.toml	File	lokesh sivakumar	Nov 26, 2025, 09:57 ...
README.md	File	lokesh sivakumar	Nov 26, 2025, 09:57 ...

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.

```

1
2 class reusable:
3     def dropColumns(self, df, columns):
4         df = df.drop(*columns)
5         return df

```

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

```

1
from pyspark.sql.functions import *
from pyspark.sql.types import *

2
import os
import sys

project_path = os.path.join(os.getcwd(), '..', '..')
sys.path.append(project_path)

from utils.transformations import reusable

3
df = spark.read.format('parquet') \
    .load('abfss://bronze@sasspotify.dfs.core.windows.net/DimUser')

4
df = pyspark.sql.connect.DataFrame = [user_id:integer, user_name:string ... 5 more fields]

```

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 (1) .option("cloudfiles.schemaEvolutionMode", "addNewColumns")\ .load("abfss://bronze@spotify.dfs.core.windows.net/DimUser")

df_user: pyspark.sql.connect.DataFrame [user_id: integer, user_name: string ... 6 more fields]

15 hours ago (1) display(df_user)

See performance (1)

display_40 (id: 9edef799-8137-4fe2-ab89-5b7a16836511) Last updated: 15 hours ago

Optimize

Dashboard Raw Data

Input vs. Processing Rate records per second 0 rec/s 172.2 rec/s Input rate Processing rate

Batch Duration in seconds 3 s 3 s Average Latest

Aggregation State 1 Distinct keys

15 hours ago

The screenshot shows a Databricks notebook interface. On the left, there's a sidebar with navigation links: Alerts, Query History, SQL Warehouses, Data Engineering, Job Runs, Data Ingestion, AI/ML, Playground, Experiments, and Features. The main area displays two code cells. The first cell has an execution count of 9 and contains Scala code that uppercases user names. It includes a 'See performance' link and an 'Optimize' button. The second cell has an execution count of 10 and contains Python code that creates a reusable DataFrame object.

```
15 hours ago
df_user = df_user.withColumn("user_name", upper(col("user_name")))
display(df_user)
▶ See performance [1]
@ display_query_41 (id: ef8b930c-2767-4c3c-a308-a494feaf69e9) Last updated: 15 hours ago
▶ df_user: pyspark.sql.connect.DataFrame = [user_id: integer, user_name: string ... 6 more fields]
```

```
df_user_obj = reusable()
```

The screenshot shows the Microsoft Azure Databricks workspace interface. On the left, the sidebar includes sections for Workspace, Recents, Catalog, Jobs & Pipelines, Compute, Marketplace, SQL, SQL Editor, Queries, Dashboards, Genie, Alerts, Query History, and SQL Warehouses. The main area displays a Python notebook titled 'silverDim'. The notebook contains the following code:

```
▶ 15 hours ago
df_user = df_user_obj.dropColumns(['_rescued_data'])
df_user = df_user.dropDuplicates(['user_id'])
display(df_user)
> See performance (1)
```

The notebook also shows a table named 'display_query_42' with the following schema and data:

	user_id	user_name	country	subscription_type	start_date	end_date	updated_at
1	174	MADISON BURNS	American Samoa	Family	2025-06-22	null	2025-10-01T19:49:45.000Z
2	215	TIMOTHY DANIEL	Grenada	Family	2024-02-27	null	2025-09-30T19:49:45.000Z
3	410	JODY DAVIS	Chile	Free	2024-04-28	null	2025-09-12T19:49:45.000Z
4	172	AMBER ANDERSON	Cambodia	Free	2024-04-20	null	2025-09-22T19:49:45.000Z
5	18	ROBERT ANDERSON	Christmas Island	Free	2024-01-02	null	2025-10-05T19:49:45.000Z
6	475	STEVEN ALLEN	Bosnia and Herzegovina	Family	2025-09-19	null	2025-09-26T19:49:45.000Z
7	74	WILLIE WALKER	United States Virgin Islands	Free	2024-11-03	null	2025-10-02T19:49:45.000Z
8	138	COURTNEY DELEON	Equatorial Guinea	Free	2025-02-03	null	2025-10-02T19:49:45.000Z
9	104	MICHAEL MEZA	Mexico	Family	2023-12-11	null	2025-10-07T19:49:45.000Z
10	180	DONALD CRAIG	Sweden	Family	2024-08-22	null	2025-09-22T19:49:45.000Z
11	468	DEBORAH PRICE	Australia	Premium	2024-03-27	null	2025-09-16T19:49:45.000Z
12	483	KELLY SIMMONS	Hong Kong	Family	2024-01-10	null	2025-09-30T19:49:45.000Z
13	407	ANA AUSTIN	Niue	Family	2024-11-19	null	2025-09-29T19:49:45.000Z
14	36	BRETT ADAMS	Croatia	Premium	2024-02-12	null	2025-09-10T19:49:45.000Z
15							

At the bottom, it says '500 rows'.

Microsoft Azure | databricks

silverDim

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago

12

```
df_user.writeStream.format("delta")\
    .outputMode("append")\
    .option("checkpointLocation", "abfss://silver@saspotify.dfs.core.windows.net/DimUser/checkpoint")\
    .trigger(once=True)\
    .option("path", "abfss://silver@saspotify.dfs.core.windows.net/DimUser/data")\
    .toTable("cata_silver.silver.DimUser")
```

See performance (1)

82d8da26-6a1d-4cc5-85e3-fa455d953573 Last updated: 15 hours ago

<pyspark.sql.connect.streaming.query.StreamingQuery at 0x7f006a9968a0>

DimArtist

14

```
df_artist = spark.readStream.format("cloudFiles")\
    .option("cloudFiles.format", "parquet")\
    .option("cloudFiles.schemaLocation", "abfss://silver@saspotify.dfs.core.windows.net/DimArtist/schema")\
    .option("cloudFiles.schemaEvolutionMode", "addNewColumns")\
    .load("abfss://bronze@saspotify.dfs.core.windows.net/DimArtist")
```

df_artist: pyspark.sql.connect.dataframe.DataFrame = [artist_id: integer, artist_name: string ... 4 more fields]

15

```
display(df_artist)
```

See performance (3)

display_query_43 (id: bab1ec41-2c46-4da7-9ba3-ab3f376727af) Last updated: 15 hours ago

Microsoft Azure | databricks

silverDim

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago

15

```
display(df_artist)
```

See performance (3)

display_query_43 (id: bab1ec41-2c46-4da7-9ba3-ab3f376727af) Last updated: 15 hours ago

16

```
df_artist_obj = reusable()
```

17

```
df_artist = df_artist_obj.dropColumns(df_artist, ['_rescued_data'])\
    .dropDuplicates(['artist_id'])\
    display(df_artist)
```

See performance (3)

display_query_44 (id: a9339c76-9110-4306-8595-d6acd406e59) Last updated: 15 hours ago

df_artist: pyspark.sql.connect.dataframe.DataFrame = [artist_id: integer, artist_name: string ... 3 more fields]

18

```
df_artist.writeStream.format("delta")\
    .outputMode("append")\
    .option("checkpointLocation", "abfss://silver@saspotify.dfs.core.windows.net/DimArtist/checkpoint")\
    .trigger(once=True)\
    .option("path", "abfss://silver@saspotify.dfs.core.windows.net/DimArtist/data")\
    .toTable("cata_silver.silver.DimArtist")
```

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.

```

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago
+ silverDim + Search data, notebooks, recents, and more... CTRL + P
db_spotify Share
15 hours ago 18
df_artist.writeStream.format("delta")\
    .outputMode("append")\
    .option("checkpointLocation", "abfss://silver@saspotify.dfs.core.windows.net/DimArtist/checkpoint")\
    .trigger(once=True)\n    .option("path","abfss://silver@saspotify.dfs.core.windows.net/DimArtist/data")\
    .toTable("cata_silver.silver.DimArtist")
> See performance (1)
> 21e48dd4f-0247-4275-9a9c-ee1bdaa06abf Last updated: 15 hours ago
<pyspark.sql.connect.streaming.query.StreamingQuery at 0x7f00c44eb70>

```

DimTrack

```

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago
+ DimTrack + Search data, notebooks, recents, and more... CTRL + P
db_spotify Share
15 hours ago 20
df_track = spark.readStream.format("cloudFiles")\
    .option("cloudFiles.format","parquet")\
    .option("cloudFiles.schemaLocation","abfss://silver@saspotify.dfs.core.windows.net/DimTrack/schema")\
    .option("cloudFiles.schemaEvolutionMode","addNewColumns")\
    .load("abfss://bronze@saspotify.dfs.core.windows.net/DimTrack")
display(df_track)
> See performance (1)
> display_query_45 (id: 8b31d4db-2437-427c-b9e6-a67df803ea41) Last updated: 15 hours ago
> df_track: pyspark.sql.connect.dataframe.DataFrame = [track_id: integer, track_name: string ... 6 more fields]

```

```

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago
+ silverDim + Search data, notebooks, recents, and more... CTRL + P
db_spotify Share
15 hours ago 21
df_track = df_track.withColumn("duration_flag",when(col("duration_sec")>150,"low")\
    .when(col("duration_sec")>300,"medium")\
    .otherwise("high"))

df_track = df_track.withColumn("track_name", regexp_replace(col("track_name"),"-"," "))
df_track = reusable().dropColumns(df_track,['_rescued_data'])

display(df_track)
> See performance (3)
> display_query_46 (id: d7030cb5-b38b-4e01-95d2-b670cc1c9045) Last updated: 15 hours ago
> df_track: pyspark.sql.connect.dataframe.DataFrame = [track_id: integer, track_name: string ... 6 more fields]

15 hours ago 22
df_track.writeStream.format("delta")\
    .outputMode("append")\
    .option("checkpointLocation", "abfss://silver@saspotify.dfs.core.windows.net/DimTrack/checkpoint")\
    .trigger(once=True)\n    .option("path","abfss://silver@saspotify.dfs.core.windows.net/DimTrack/data")\
    .toTable("cata_silver.silver.DimTrack")
> See performance (1)
> a867c34f-f9f1-407d-ac60-88ba7df7cb6c Last updated: 15 hours ago
<pyspark.sql.connect.streaming.query.StreamingQuery at 0x7f00d0219e20>

```

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.

```

    df_date = spark.readStream.format("cloudFiles")\
        .option("cloudFiles.format","parquet")\
        .option("cloudFiles.schemaLocation","abfss://silver@sasspotify.dfs.core.windows.net/DimDate/schema")\
        .option("cloudFiles.schemaEvolutionMode","addNewColumns")\
        .load("abfss://bronze@sasspotify.dfs.core.windows.net/DimDate")
    display(df_date)

```

The screenshot shows a Databricks notebook titled "DimDate". The sidebar on the left lists various workspace sections like Recents, Catalog, Jobs & Pipelines, Compute, Marketplace, SQL, and Data Engineering. The main area contains a single cell of code. The code reads data from a Bronze layer using a streaming cloud file source and saves it to a Silver layer using a Delta table sink.

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.

```

    df_date.writeStream.format("delta")\
        .outputMode("append")\
        .option("checkpointLocation", "abfss://silver@sasspotify.dfs.core.windows.net/DimDate/checkpoint")\
        .trigger(once=True)\n        .option("path","abfss://silver@sasspotify.dfs.core.windows.net/DimDate/data")\
        .toTable("cata_silver.silver.DimDate")

```

The screenshot shows a Databricks notebook titled "FactStream". The sidebar on the left lists various workspace sections like Recents, Catalog, Jobs & Pipelines, Compute, Marketplace, SQL, and Data Engineering. The main area contains a single cell of code. The code reads data from a Bronze layer using a streaming cloud file source and saves it to a Silver layer using a Delta table sink.

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.

```

File Edit View Run Help Python Tabs: ON Last edit was 15 hours ago
+ silverDim + 29
df_fact = reusable().dropColumns(df_fact,['_rescued_data'])

+ df_fact: pyspark.sql.connect.DataFrame [stream:id: long, user_id: integer ... 5 more fields]

+ 15 hours ago 30
df_fact.writeStream.format("delta")\
    .outputMode("append")\
    .option("checkpointLocation", "abfss://silver@saspotify.dfs.core.windows.net/FactStream/checkpoint")\
    .trigger(once=True)\n    .option("path","abfss://silver@saspotify.dfs.core.windows.net/FactStream/data")\
    .toTable("cata_silver.silver.FactStream")
> See performance (3)
> fe4871c4-bcbd-4927-ab97-394583b36175 Last updated: 15 hours ago
<pyspark.sql.connect.streaming.query.StreamingQuery at 0x7f00747357c0>

```

[Shift+Enter] to run and move to next cell
[Ctrl+Shift+P] to open the command palette
[Esc H] to see all keyboard shortcuts

FscfStream

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.

```

parameters = [
    {
        "table": "cata_silver.silver.factstream",
        "alias": "factstream",
        "cols": "factstream.stream_id,factstream.listen_duration"
    },
    {
        "table": "cata_silver.silver.dimuser",
        "alias": "dimuser",
        "cols": "dimuser.user_id,dimuser.user_name",
        "condition": "factstream.user_id=dimuser.user_id"
    },
    {
        "table": "cata_silver.silver.dimtrack",
        "alias": "dimtrack",
        "cols": "dimtrack.track_id,dimtrack.track_name",
        "condition": "factstream.track_id=dimtrack.track_id"
    }
]

from jinja2 import Template

```

Jinja

This screenshot shows a Databricks notebook titled "jinja_nb" in Python mode. The notebook contains the following Jinja templated SQL code:

```
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 %}  
    {% endif %}  
    {% for param in parameters %}  
        {% if not loop.first %}LEFT JOIN  
            {{ param.table }} AS {{ param.alias }}  
        ON  
            {{ param.condition }}  
        {% endif %}  
    {% endfor %}  
...  
"""
```

Jinja

This screenshot shows a Databricks notebook titled "jinja_nb" in Python mode. The notebook contains the following Jinja templated Python code:

```
jinja_sql_str = Template(query_text)  
query = jinja_sql_str.render(parameters=parameters)  
print(query)
```

Below this, the generated SQL query is shown:

```
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
```

Jinja

The screenshot shows the Databricks workspace interface. On the left, the sidebar includes sections for Workspace, Recents, Catalog, Jobs & Pipelines, Compute, Marketplace, SQL (SQL Editor, Queries, Dashboards), Genie, Alerts, Query History, and SQL Warehouses. The main area shows a notebook titled 'jinja.nb' with tabs for 'silverDim', 'requirements.txt', 'databricks.yml', and '+'. A message at the top says 'Python variables from your previous serverless session are available. Click reconnect to restore your variables.' Below this, a code cell contains the command `display(spark.sql(query))` and a link to 'See performance (1)'. The result is a table with columns: stream_id, listen_duration, user_id, user_name, track_id, and track_name. The table contains 15 rows of data, with the last row being Michael Hayes. At the bottom, it says '1,000 rows | 1.73s runtime' and 'Refreshed 19 hours ago'.

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.

The screenshot shows the Databricks workspace interface. The sidebar is identical to the previous one. The main area shows a YML file named 'databricks.yml' with the following content:

```

1  bundle:
2    name: spotify_dbn
3    uid: 72448748-38ed-47c1-befa-8355195e9f6e
4
5    targets:
6      dev:
7        mode: development
8        default: true
9        workspace:
10       host: http://adb-1921923455253364.4.azuredatabricks.net
11
12      presets:
13        source_linked_deployment: false
14
15      prod:
16        mode: production
17        workspace:
18          host: http://adb-1921923455253364.4.azuredatabricks.net
19          root_path: /Workspace/PRODUCTION/.bundle/${bundle.name}/${bundle.target}
20
21      permissions:
22        - user_name: lokeshsivakumar012@gmail.com
23        level: CAN_MANAGE

```

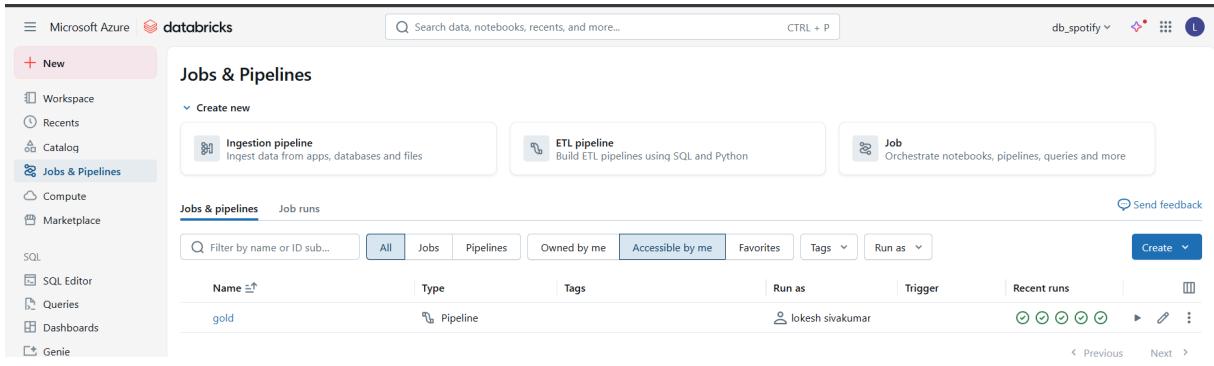
Yml

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.



Jobs & Pipelines

Ingestion pipeline

ETL pipeline

Job

Send feedback

Create

gold

Type: Pipeline

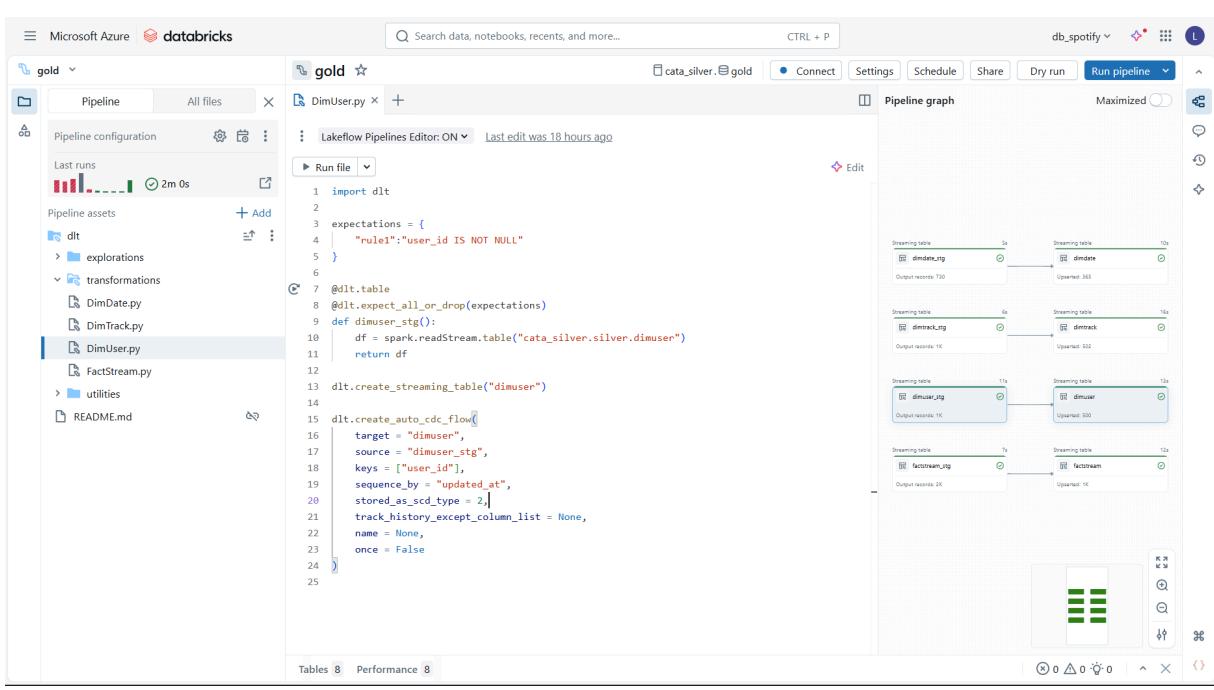
Run as: lokesh sivakumar

Trigger: Recent runs

Recent runs: 5

Previous Next

Gold layer



Pipeline configuration

Last runs: 2m 0s

Pipeline assets: dlt, explorations, transformations, DimDate.py, DimTrack.py, DimUser.py, FactStream.py, utilities, README.md

DimUser.py

```
1 import dlt
2
3 expectations = [
4     "rule1":"user_id IS NOT NULL"
5 ]
6
7 @dlt.table
8 @dlt.expect_all_or_drop(expectations)
9 def dimuser_stg():
10     df = spark.readStream.table("cata_silver.silver.dimuser")
11     return df
12
13 dlt.create_streaming_table("dimuser")
14
15 dlt.create_auto_cdc_flow(
16     target = "dimuser",
17     source = "dimuser_stg",
18     keys = ["user_id"],
19     sequence_by = "updated_at",
20     stored_as_scd_type = 2,
21     track_history_except_column_list = None,
22     name = None,
23     once = False
24 )
```

Tables 8 Performance 8

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.

The screenshot shows the Microsoft Azure Databricks Pipeline Editor interface. The left sidebar displays the 'gold' pipeline configuration with a 'Pipeline assets' section containing 'dlt' (Data Lake Transformation) files: DimDate.py, DimTrack.py, DimUser.py, FactStream.py, and README.md. The main workspace shows the 'DimDate.py' file open, which contains Python code for creating streaming tables and autoCDC flows. The right side features a 'Pipeline graph' window showing the data flow between various streaming tables like dimdate_stg, dimtrack_stg, dimuser_stg, factstream_stg, and their corresponding dimdate, dimtrack, dimuser, and factstream tables. A status bar at the bottom indicates 8 Tables and 8 Performance.

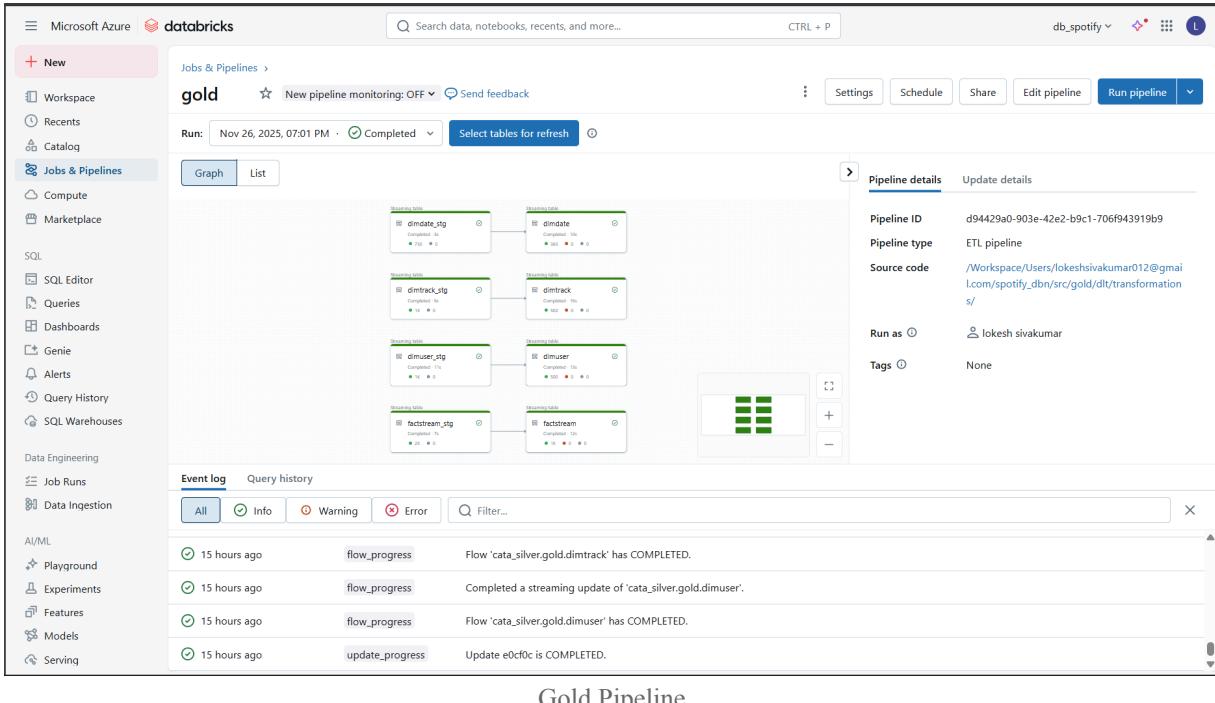
```
1 import dlt
2
3 @dlt.table
4 def dimdate_stg():
5     df = spark.readStream.table("cata_silver.silver.dimdate")
6     return df
7
8 dlt.create_streaming_table("dimdate")
9
10 dlt.create_auto_cdc_flow(
11     target = "dimdate",
12     source = "dimdate_stg",
13     keys = ["date"],
14     sequence_by = "date",
15     stored_as_scd_type = 1,
16     track_history_except_column_list = None,
17     name = None,
18     once = False
19 )
20
```

DimDate

This screenshot shows the same Databricks Pipeline Editor interface, but with the 'DimTrack' pipeline selected. The left sidebar shows the 'gold' pipeline configuration with the 'transformations' section expanded, revealing DimDate.py and DimTrack.py. The main workspace shows the 'DimTrack.py' file open, containing Python code for creating streaming tables and autoCDC flows. The right side shows the 'Pipeline graph' with data flow between streaming tables like dimtrack_stg, dimuser_stg, factstream_stg, and their corresponding dimtrack, dimuser, and factstream tables. A status bar at the bottom indicates 8 Tables and 8 Performance.

```
1 import dlt
2
3 @dlt.table
4 def dimtrack_stg():
5     df = spark.readStream.table("cata_silver.silver.dimtrack")
6     return df
7
8 dlt.create_streaming_table("dimtrack")
9
10 dlt.create_auto_cdc_flow(
11     target = "dimtrack",
12     source = "dimtrack_stg",
13     keys = ["track_id"],
14     sequence_by = "updated_at",
15     stored_as_scd_type = 2,
16     track_history_except_column_list = None,
17     name = None,
18     once = False
19 )
20
```

DimTrack



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.