

EMPLOYEE ATTRITION SYSTEM

1. Introduction

In today's competitive business environment, human capital is one of the most valuable assets an organization can possess. Employee attrition — the gradual loss of employees over time — poses a critical challenge to workforce stability and organizational efficiency. High attrition rates can disrupt operations, increase recruitment costs, and impact team morale and productivity. To mitigate these risks, businesses are increasingly adopting data-driven strategies to predict and manage employee attrition effectively.

This project, titled Employee Attrition System, aims to develop a scalable and automated data pipeline for ingesting, processing, and analyzing HR data, and then building a machine learning model to predict the likelihood of employee attrition. By integrating Azure services such as Azure SQL Database, Azure Data Factory, and Azure Databricks, along with ML components like XGBoost and MLflow, this project demonstrates an end-to-end solution for attrition prediction within a modern data engineering and machine learning ecosystem.

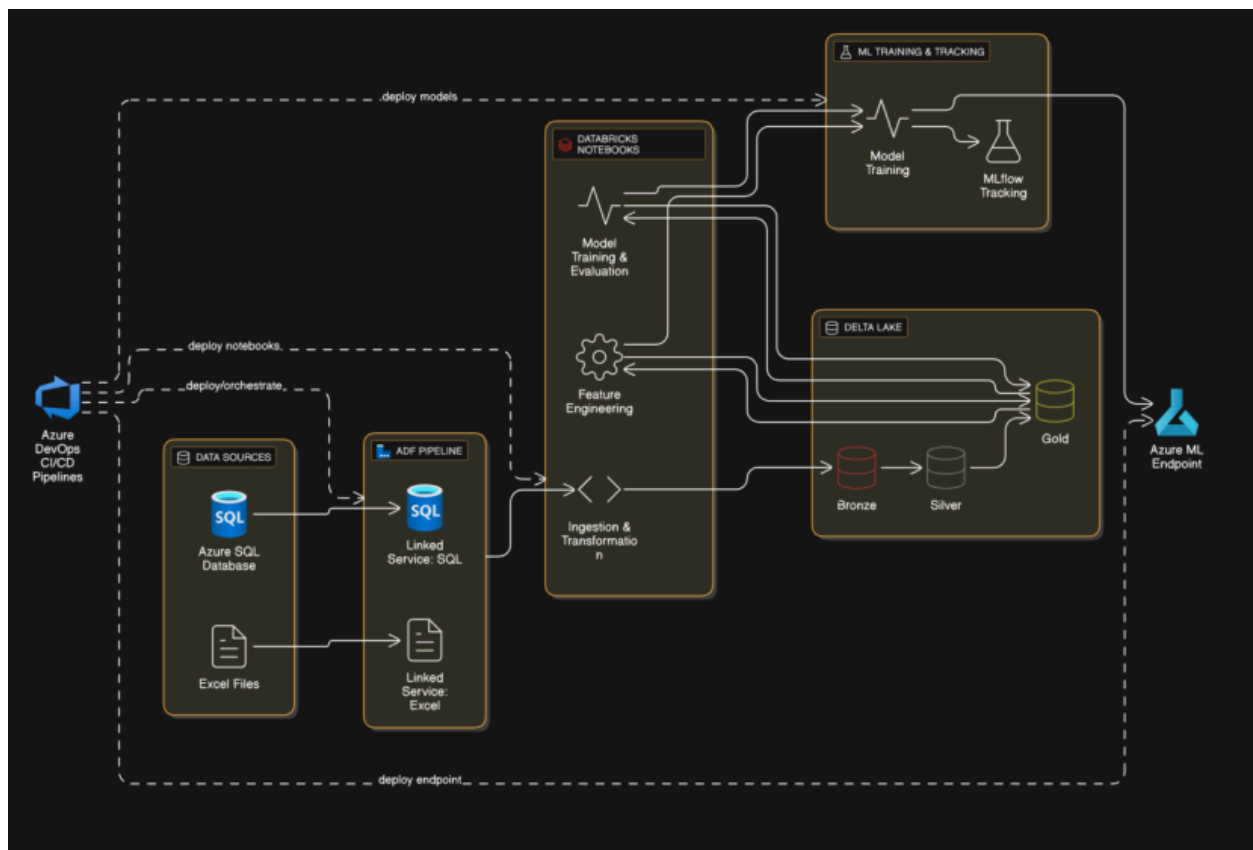
The solution has been designed using the Medallion Architecture (Bronze → Silver → Gold) pattern to ensure modularity, scalability, and maintainability of data transformations. The final ML model is tracked and versioned using MLflow, enabling reproducibility, governance, and traceability.

2. Problem Statement

Employee attrition is a significant concern for organizations striving to retain talent and maintain operational continuity. When employees leave unexpectedly, it leads to disruptions, recruitment expenses, and loss of organizational knowledge. Therefore, proactively identifying employees at risk of attrition is crucial for Human Resources (HR) departments to implement preventive strategies.

This project tackles two interconnected problems: ETL (Extract, Transform, Load) and ML (Machine Learning), framed under a unified system called the Employee Attrition System.

3. Architecture Diagram



- **Data Sources**

- **Azure SQL Database:** Contains structured HR data such as employee records, attrition status, job role, and demographics.
- **Excel Files in ADLS Gen2:** Semi-structured data like employee evaluations, survey results, and satisfaction metrics.

- **Azure Data Factory (ADF)**

- Orchestrates data movement from SQL and Excel sources.
- Uses Linked Services to securely connect to both data sources.
- Launches Databricks notebooks for processing.

- **Azure Databricks Notebooks**

- Runs Spark-based notebooks for ETL.
- Implements the **Medallion Architecture**:
 - **Bronze Layer:** Raw ingested data.
 - **Silver Layer:** Cleaned and joined data.
 - **Gold Layer:** Enriched and feature-engineered dataset for ML.

- **Delta Lake**

- Provides ACID-compliant, scalable storage for each ETL stage (Bronze → Silver → Gold).
- Enables efficient time-travel and versioning.

- **MLflow (within Databricks)**

- Tracks all machine learning experiments.
- Logs parameters, metrics (accuracy, AUC), and model versions.

- **Azure Machine Learning**

- Consumes the registered model from MLflow.

- Deploys it as a REST endpoint for real-time inference.
- **Azure DevOps**
 - Automates notebook deployment to Databricks.
 - Triggers ADF pipeline and Azure ML model deployment using CI/CD YAML pipelines.

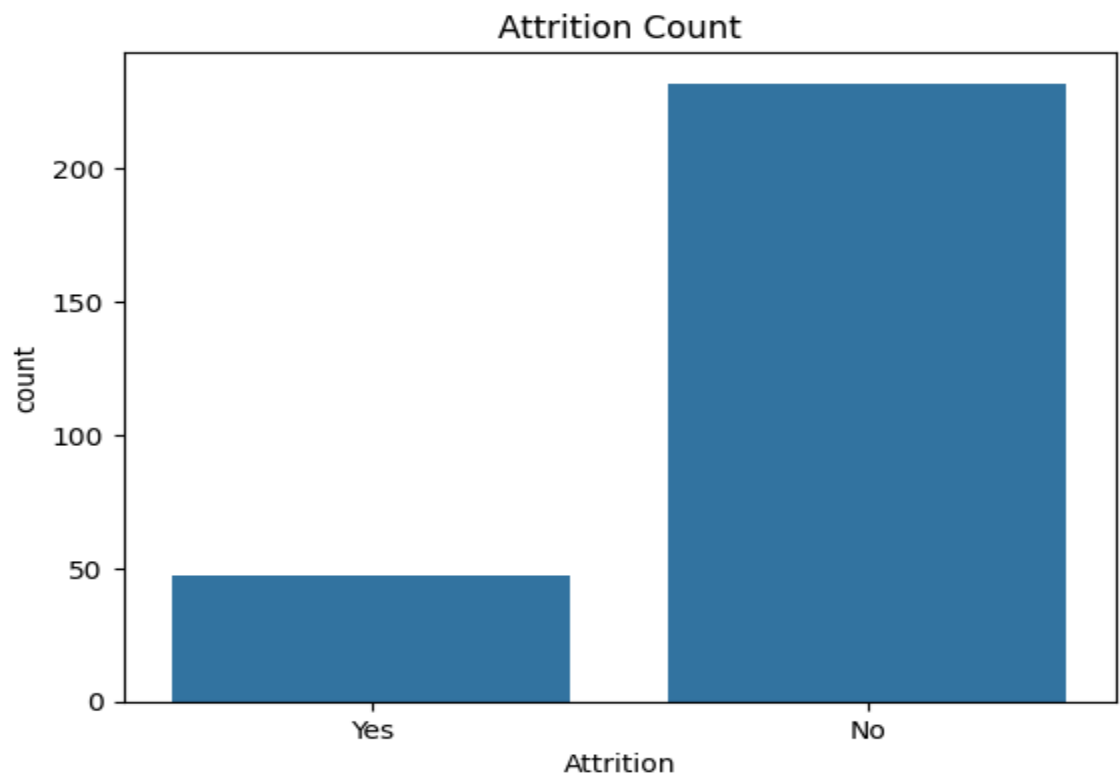
4. Dataset Overview

The HR Employee Attrition dataset used in this project provides comprehensive information about employees across various attributes including personal details, job roles, work-life balance, compensation, and attrition status. This dataset serves as the foundation for both data transformation and machine learning tasks in the Employee Attrition System.

The dataset contains 1,470 records and 35 columns. Each record represents an employee's profile with details on demographics, job role, salary, performance, and attrition status.

Category	Columns
Personal Info	Age, Gender, MaritalStatus, Over18, EducationField
Job Details	JobRole, Department, JobLevel, JobInvolvement, EnvironmentSatisfaction, PerformanceRating

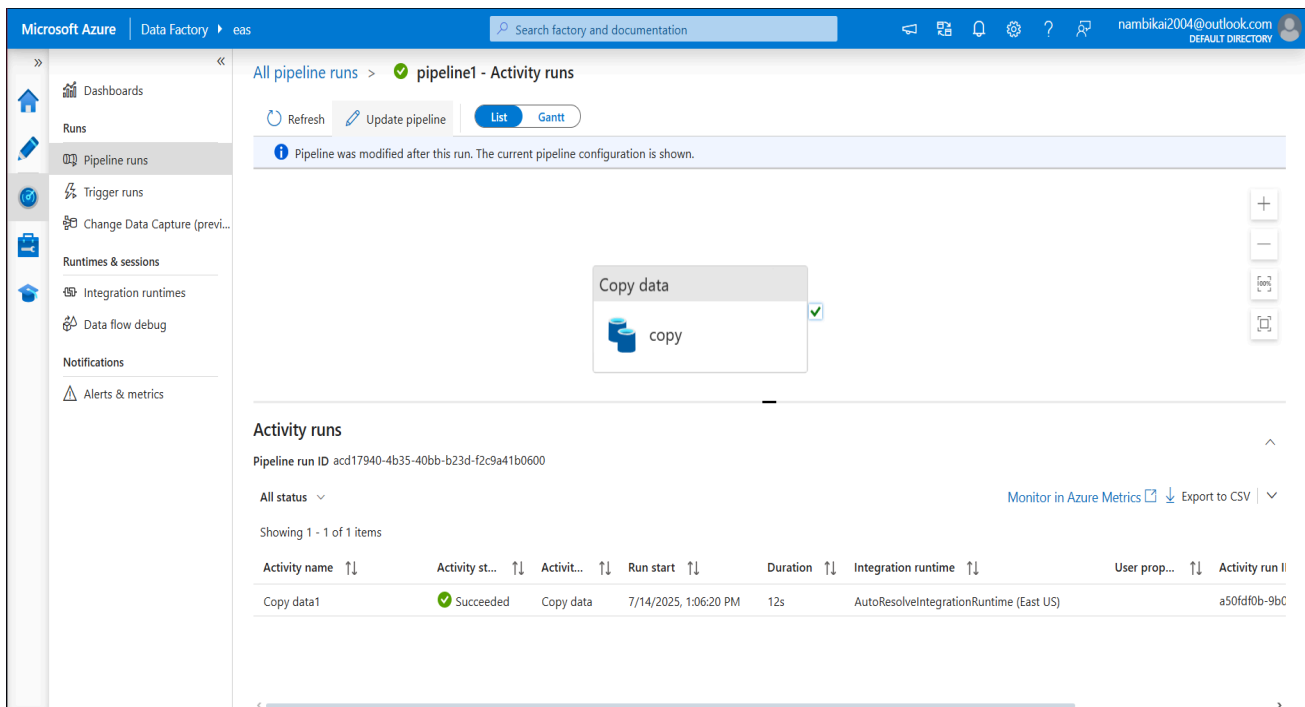
Compensation	MonthlyIncome, HourlyRate, PercentSalaryHike, StockOptionLevel
Work-Life Balance	WorkLifeBalance, BusinessTravel, OverTime, DistanceFromHome
Employment Duration	YearsAtCompany, YearsInCurrentRole, TotalWorkingYears, YearsSinceLastPromotion
Attrition	Attrition (Target variable: Yes/No)



5. Data Ingestion using Azure Data Factory (ADF)

The first step in our Employee Attrition System project involved building an ETL (Extract, Transform, Load) pipeline using Azure Data Factory (ADF). ADF was used to ingest HR data from two distinct sources:

- Azure SQL Database (for structured transactional HR data)
- Azure Data Lake Storage Gen2 (for Excel/CSV flat files)



The screenshot displays the Microsoft Azure Data Factory interface. The left sidebar contains navigation options: Dashboards, Runs, Pipeline runs (selected), Trigger runs, Change Data Capture (previous), Runtimes & sessions, Integration runtimes, Data flow debug, Notifications, and Alerts & metrics. The main pane shows the 'All pipeline runs' view for 'pipeline1 - Activity runs'. A notification states: 'Pipeline was modified after this run. The current pipeline configuration is shown.' Below this, a 'Copy data' activity is highlighted with a green checkmark. The 'Activity runs' section shows a table with one row of data for a successful run.

Activity name	Activity st...	Activit...	Run start	Duration	Integration runtime	User prop...	Activity run ID
Copy data1	✓ Succeeded	Copy data	7/14/2025, 1:06:20 PM	12s	AutoResolveIntegrationRuntime (East US)		a50fd0b-9b0

- These ingested dataset are then copied from the Azure Data Lake Storage Gen2 to the Azure Blob storage.

Microsoft Azure | Upgrade | Search resources, services, and docs (G+/) | Copilot | nambikai2004@outlook... | DEFAULT DIRECTORY (NAMBICAL...)

Home > easfinal1 | Containers >

dataset Container

Search

+ Add Directory | Upload | Change access level | Refresh | Delete | Copy | Paste | Rename | Acquire lease | Break lease | Edit columns

dataset

Authentication method: Access key (Switch to Microsoft Entra user account)

Add filter

Search blobs by prefix (case-sensitive) | Only show active blobs

Showing all 1 items

<input type="checkbox"/>	Name	Last modified	Access tier	Blob type	Size	Lease state
<input type="checkbox"/>	WA_Fn-UseC_-HR-Employee-Attrition.csv	7/14/2025, 6:33:25 PM	Hot (Inferred)	Block blob	222.63 KiB	Available

Add or remove favorites by pressing Ctr+L+Shift+F

- Then the Blob storage is mounted on the Bronze Layer of Delta Lake in Azure Databricks.

EAS | main | Python | Tabs: OFF | ☆

File Edit View Run Help | Last edit was 5 hours ago

Run all | Terminated | Schedule (1) | Share

```
dbutils.fs.unmount('/mnt/finalproject')
```

/mnt/finalproject has been unmounted.

True

```
dbutils.fs.mount(
    source="wasbs://dataset@easfinal1.blob.core.windows.net/",
    mount_point="/mnt/finalproject",
    extra_configs={f"fs.azure.account.key.easfinal1.blob.core.windows.net": "qhsZASaH5HRwLQPdU70zkRwDY3An7/MQC3xK8lXmQT1ptuT7SJNYKcz/Uzh0o8tgBwDMLQFazLX0+ASTKQsu0Q=="})
```

True

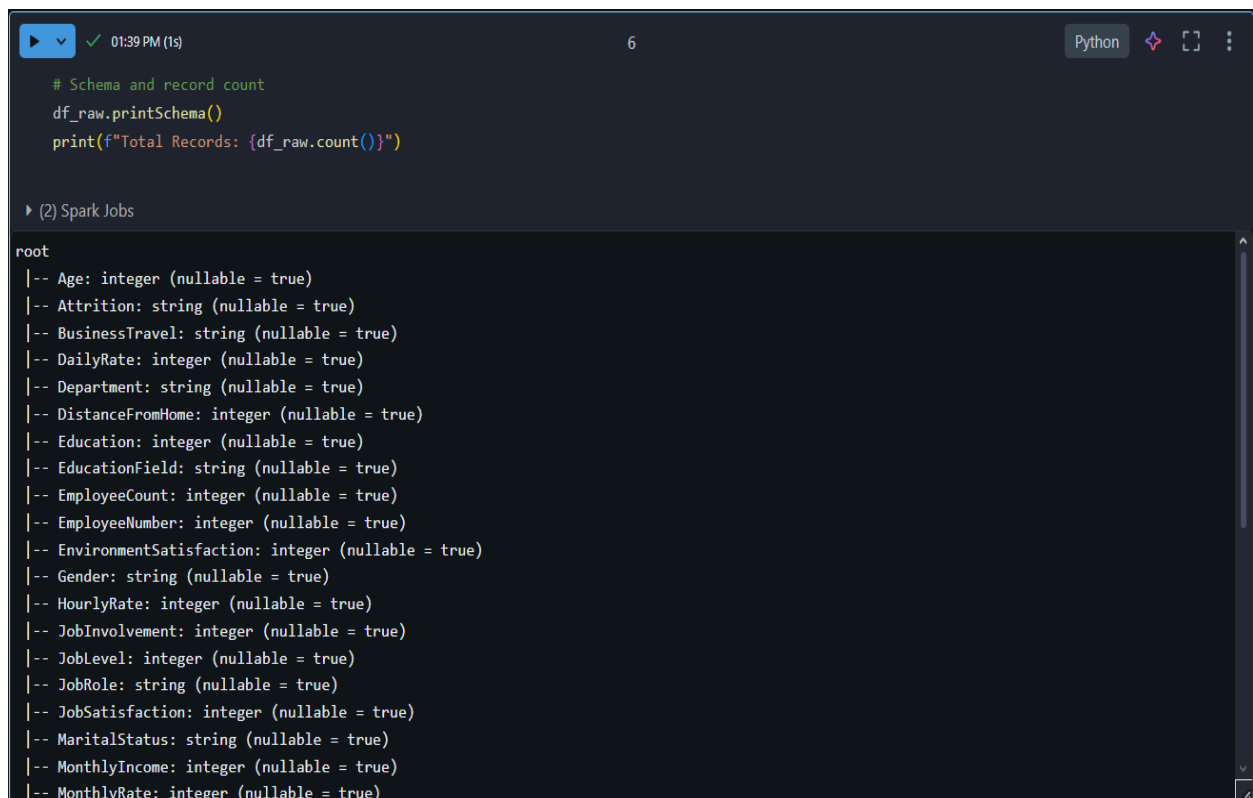
6. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) was conducted on the cleaned and enriched data (Silver Layer) to uncover patterns, identify relationships between features, and understand class imbalance in the attrition variable.

We used Apache Spark (PySpark) to perform EDA in a distributed and scalable way.

6.1 Data Overview

We examined the basic structure, schema, and summary statistics of the dataset.



```
# Schema and record count
df_raw.printSchema()
print(f"Total Records: {df_raw.count()}")
```

▶ (2) Spark Jobs

```
root
|-- Age: integer (nullable = true)
|-- Attrition: string (nullable = true)
|-- BusinessTravel: string (nullable = true)
|-- DailyRate: integer (nullable = true)
|-- Department: string (nullable = true)
|-- DistanceFromHome: integer (nullable = true)
|-- Education: integer (nullable = true)
|-- EducationField: string (nullable = true)
|-- EmployeeCount: integer (nullable = true)
|-- EmployeeNumber: integer (nullable = true)
|-- EnvironmentSatisfaction: integer (nullable = true)
|-- Gender: string (nullable = true)
|-- HourlyRate: integer (nullable = true)
|-- JobInvolvement: integer (nullable = true)
|-- JobLevel: integer (nullable = true)
|-- JobRole: string (nullable = true)
|-- JobSatisfaction: integer (nullable = true)
|-- MaritalStatus: string (nullable = true)
|-- MonthlyIncome: integer (nullable = true)
|-- MonthlyRate: integer (nullable = true)
```



```
df_raw.groupBy("Attrition").count().show()
```

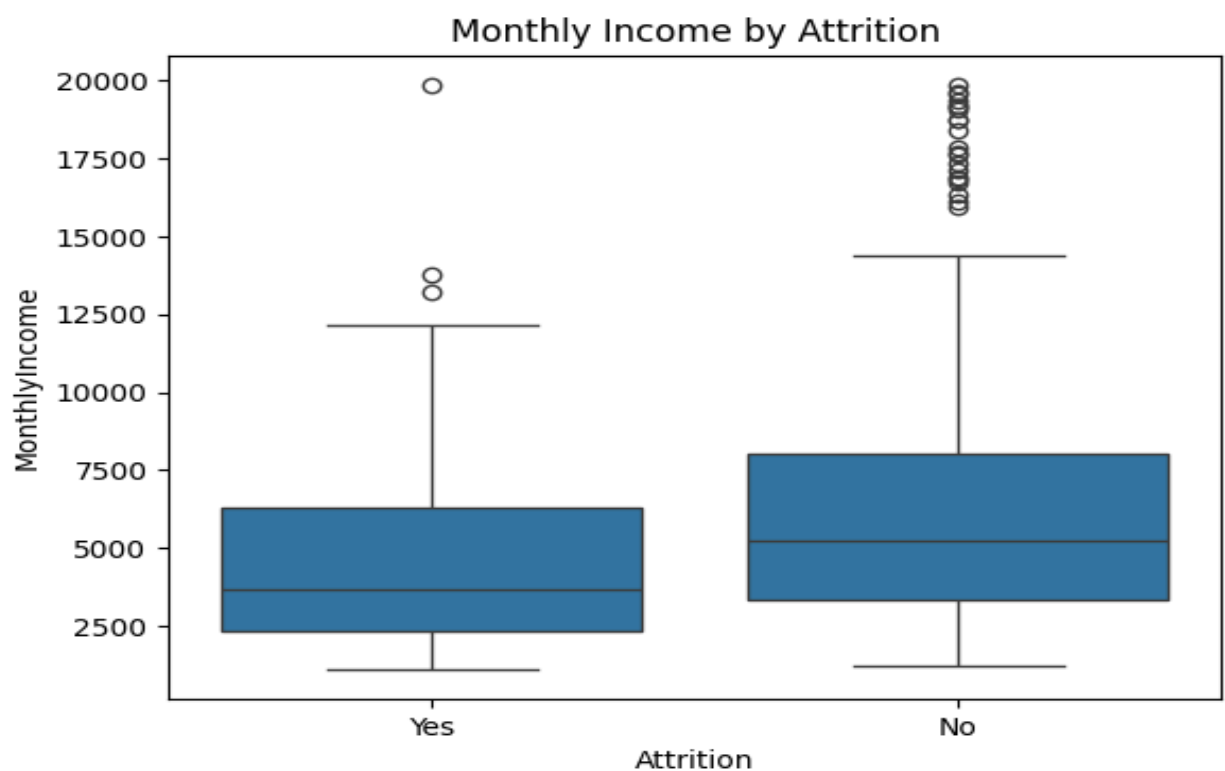
(2) Spark Jobs

Attrition	count
No	1233
Yes	237

```
df_raw.select("Age", "MonthlyIncome", "YearsAtCompany", "DistanceFromHome", "JobSatisfaction").describe().show()
```

(2) Spark Jobs

summary	Age	MonthlyIncome	YearsAtCompany	DistanceFromHome	JobSatisfaction
count	1470	1470	1470	1470	1470
mean	36.923809523809524	6502.931292517007	7.0081632653061225	9.19251700680272	2.7285714285714286
stddev	9.135373489136734	4707.956783097995	6.126525152403566	8.106864435666084	1.1028461230547149
min	18	1009	0	1	1
max	60	19999	40	29	4



Observation:

- The dataset is imbalanced, with more employees not leaving (AttritionFlag = 0) than leaving (AttritionFlag = 1).

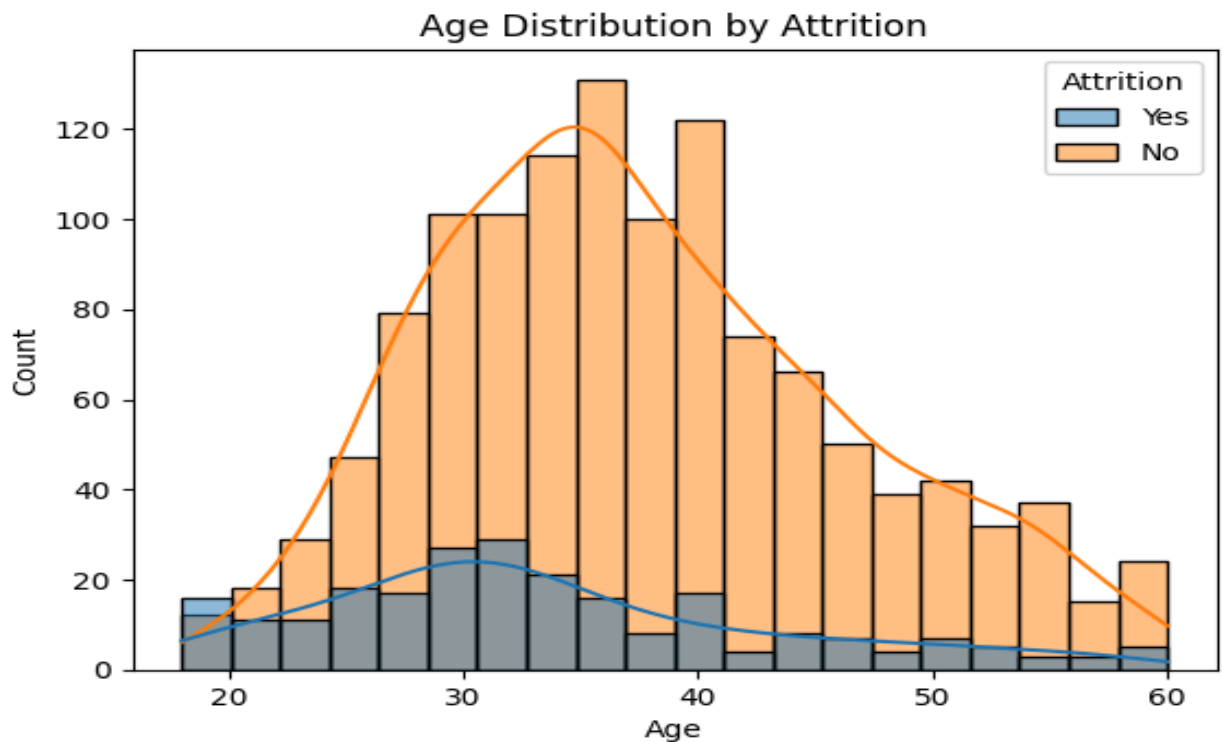
6.2 Feature Distributions

We visualized key numerical features to understand distributions.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Convert Spark to Pandas
pdf = df_raw.select("Age", "MonthlyIncome", "DistanceFromHome", "Attrition").toPandas()

# Plot histograms
sns.histplot(data=pdf, x="Age", hue="Attrition", kde=True, bins=20)
plt.title("Age Distribution by Attrition")
plt.show()
```



Insights:

- Younger employees had a higher attrition rate.
- Distance from home also showed some influence on attrition.

6.3 Attrition by Categorical Features

We visualized how categorical features affect attrition:

```
▶ ✓ Just now (3s) 11

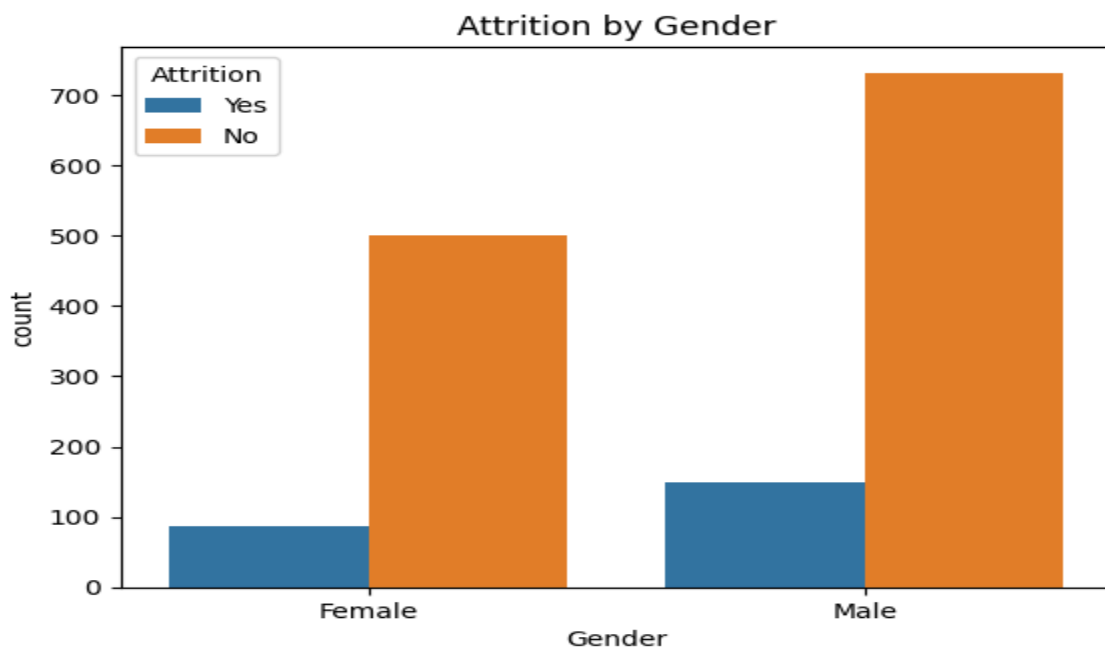
# Convert PySpark DataFrame to Pandas
pdf = df_raw.select("Gender", "OverTime", "Attrition").toPandas()

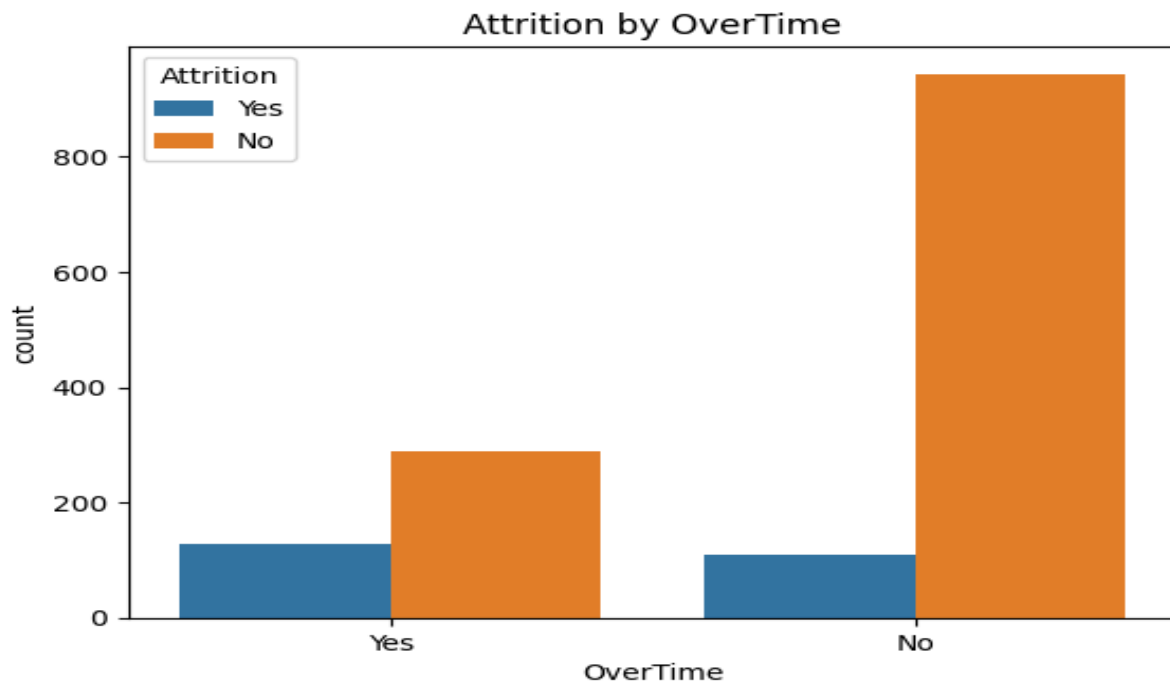
# Now use Seaborn to plot
import seaborn as sns
import matplotlib.pyplot as plt

sns.countplot(data=pdf, x="Gender", hue="Attrition")
plt.title("Attrition by Gender")
plt.show()

sns.countplot(data=pdf, x="OverTime", hue="Attrition")
plt.title("Attrition by OverTime")
plt.show()

▶ (1) Spark Jobs
```





Insights:

- Employees who worked overtime were more likely to leave.
- Attrition was slightly higher among males in this dataset.

6.4 Class Imbalance Check

```
Just now (3s) 12
df_raw.groupBy("Attrition").count().show()
(2) Spark Jobs
+-----+-----+
|Attrition|count|
+-----+-----+
|    No   | 1233|
|    Yes  |  237|
+-----+-----+
```

Observation:

- Class imbalance is significant (majority class = 0), so techniques like stratified sampling, balanced class weights, or SMOTE may be considered during modeling.

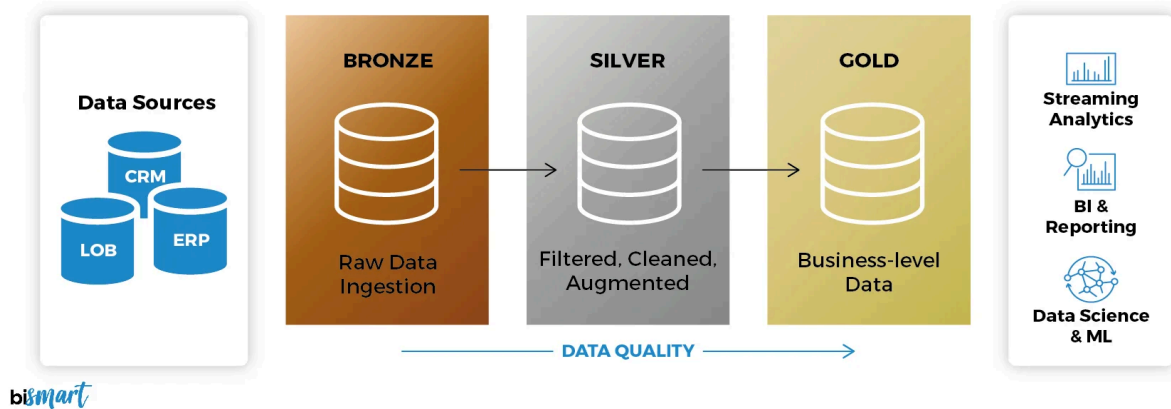
7. Data Processing in Azure Databricks

After ingestion into the Bronze Layer via Azure Data Factory, the next step was to clean, join, and enrich the HR data using **Azure Databricks**. The goal was to create a refined dataset suitable for analytics and machine learning tasks.

7.1 Medallion Architecture Processing

We adopted the **Medallion Architecture (Bronze → Silver → Gold)**:

Layer	Purpose
Bronze	Raw ingested data from source systems
Silver	Cleaned and joined data for further processing
Gold	Aggregated and transformed data ready for analytics/ML



7.2 Bronze Layer: Raw Data Loading

The Bronze Layer is the first stage in the Medallion Architecture used in data lakehouses. It is responsible for capturing and storing raw, unprocessed data from various source systems. This layer plays a crucial role in maintaining data lineage and traceability while serving as the foundation for all downstream data processing.

Purpose of the Bronze Layer

- Capture data as-is from source systems without transformations.
- Preserve the original state of the data for audit and recovery purposes.
- Enable reprocessing in case of pipeline failures or logic updates.
- Maintain a single source of truth for historical raw data.

The screenshot shows the EAS interface with a Python script being executed. The script reads a CSV file from a Delta Lake and writes it to a new table. The output shows a preview of the data with columns: Age, Attrition, BusinessTravel, DailyRate, Department, DistanceFromHome, Education, and Life Stage.

```

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

df_raw = spark.read.option("header", True).option("inferSchema", True).csv("dbfs:/mnt/finalproject/WA_Fn-UseC-HR-Employee-Attrition.csv")
df_raw.write.format("delta").mode("overwrite").saveAsTable("employee_attrition_bronze")

```

df_raw: pyspark.sql.dataframe.DataFrame = [Age: integer, Attrition: string ... 33 more fields]

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Life Stage
1	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sc
2	49	No	Travel_Frequently	279	Research & Developme...	8	1	Life Sc
3	37	Yes	Travel_Rarely	1373	Research & Developme...	2	2	Other
4	33	No	Travel_Frequently	1392	Research & Developme...	3	4	Life Sc
5	27	No	Travel_Rarely	591	Research & Developme...	2	1	Medic
6	32	No	Travel_Frequently	1005	Research & Developme...	2	2	Life Sc

7.3 Silver Layer: Data Cleaning and Transformation

The Silver Layer is the second stage in the Medallion Architecture, where data is cleaned, validated, and enriched. It transforms the raw data from the Bronze Layer into a structured and high-quality dataset that can be used for exploratory analysis and modeling.

Purpose of the Silver Layer

- Perform data cleansing (handle nulls, fix data types, remove duplicates).
- Apply business rules and standardizations.

- Create joins between different datasets (e.g., employee data + performance ratings).
- Serve as the base for feature engineering and analytics.

EAS main Python Tabs: OFF ☆

File Edit View Run Help Last edit was 6 hours ago Run all Terminated Schedule (1) Share

```

▶ 01:39 PM (2s) 11

# Read from Bronze
df_bronze = spark.read.table("employee_attrition_bronze")

# Clean and enrich
df_silver = df_bronze.dropDuplicates() \
    .withColumn("AttritionFlag", when(col("Attrition") == "Yes", 1).otherwise(0)) \
    .drop("EmployeeNumber", "Over18", "StandardHours", "EmployeeCount")

# Write to Silver
df_silver.write.format("delta").mode("overwrite").saveAsTable("employee_attrition_silver")

▶ (2) Spark Jobs

▶ df_bronze: pyspark.sql.dataframe.DataFrame = [Age: integer, Attrition: string ... 33 more fields]
▶ df_silver: pyspark.sql.dataframe.DataFrame = [Age: integer, Attrition: string ... 30 more fields]

```

▶ 01:39 PM (1s) 12 SQL SQL SQL SQL SQL

```

%sql
SELECT * FROM employee_attrition_silver;

▶ (1) Spark Jobs

▶ _sqldf: pyspark.sql.dataframe.DataFrame = [Age: integer, Attrition: string ... 30 more fields]

```

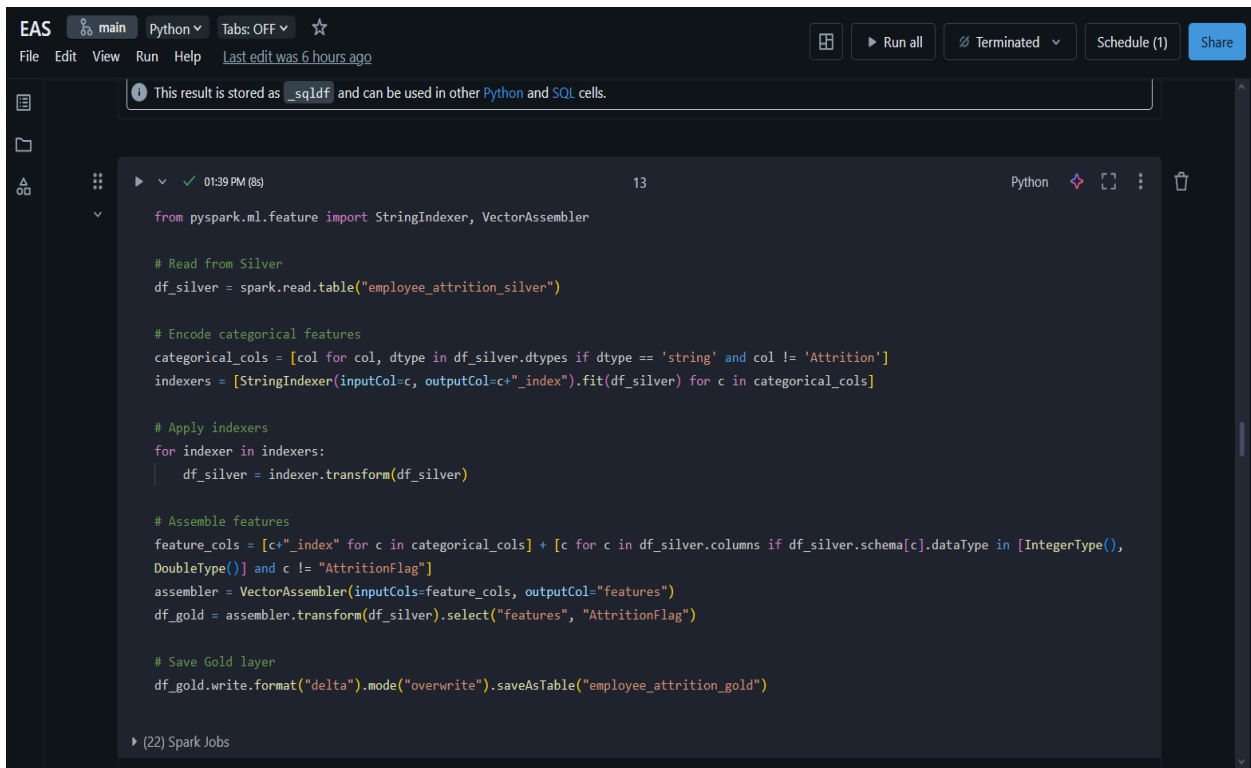
	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Life Sc
1	27	No	Travel_Rarely	608	Research & Developme...	1	2	Life Sc
2	41	No	Travel_Rarely	314	Human Resources	1	3	Humai
3	25	No	Travel_Rarely	141	Sales	3	1	Other
4	21	Yes	Travel_Rarely	1427	Research & Developme...	18	1	Other
5	56	No	Travel_Rarely	718	Research & Developme...	4	4	Techni
6	25	No	Travel_Rarely	949	Research & Developme...	1	3	Techni
7	27	No	Travel_Rarely	591	Research & Developme...	2	1	Medic
8	34	No	Non-Travel	1065	Sales	23	4	Marke
9	50	No	Travel_Rarely	328	Research & Developme...	1	3	Medic
10	25	No	Non-Travel	675	Research & Developme...	5	2	Life Sc
11	18	Yes	Travel_Frequently	1306	Sales	5	3	Marke
12	36	No	Travel_Rarely	1425	Research & Developme...	14	1	Life Sc

7.4 Feature Engineering for Gold Layer

The Gold Layer is the final stage of the Medallion Architecture. It contains analytics-ready, feature-engineered, and highly curated data used directly for reporting, visualization, and machine learning model training.

Purpose of the Gold Layer

- Provide a refined dataset with meaningful insights and transformed features.
- Serve as the single source of truth for business intelligence and ML models.
- Contain aggregated, joined, and feature-enriched data optimized for performance and accuracy.



The screenshot shows a Databricks workspace interface. At the top, there's a header with 'EAS', 'main', 'Python', and 'Tabs: OFF'. Below this is a menu bar with 'File', 'Edit', 'View', 'Run', and 'Help'. A status bar indicates 'Last edit was 6 hours ago'. On the right, there are buttons for 'Run all', 'Terminated', 'Schedule (1)', and 'Share'. A notification bar states: 'This result is stored as _sqldf and can be used in other Python and SQL cells.' The main area is a code editor with a dark theme, displaying a Python script. The script imports 'StringIndexer' and 'VectorAssembler' from 'pyspark.ml.feature'. It reads a table 'employee_attrition_silver' from 'Silver'. It then encodes categorical features by creating 'StringIndexer' objects for each categorical column and fitting them on the data. Next, it applies these indexers to the data. Finally, it assembles the features into a 'VectorAssembler', transforms the data, and saves the resulting 'df_gold' table to 'Delta' format, overwriting the existing 'employee_attrition_gold' table. The bottom of the editor shows '(22) Spark Jobs'.

```
from pyspark.ml.feature import StringIndexer, VectorAssembler

# Read from Silver
df_silver = spark.read.table("employee_attrition_silver")

# Encode categorical features
categorical_cols = [col for col, dtype in df_silver.dtypes if dtype == 'string' and col != 'Attrition']
indexers = [StringIndexer(inputCol=c, outputCol=c+"_index").fit(df_silver) for c in categorical_cols]

# Apply indexers
for indexer in indexers:
    df_silver = indexer.transform(df_silver)

# Assemble features
feature_cols = [c+"_index" for c in categorical_cols] + [c for c in df_silver.columns if df_silver.schema[c].dataType in [IntegerType(), DoubleType()] and c != "AttritionFlag"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
df_gold = assembler.transform(df_silver).select("features", "AttritionFlag")

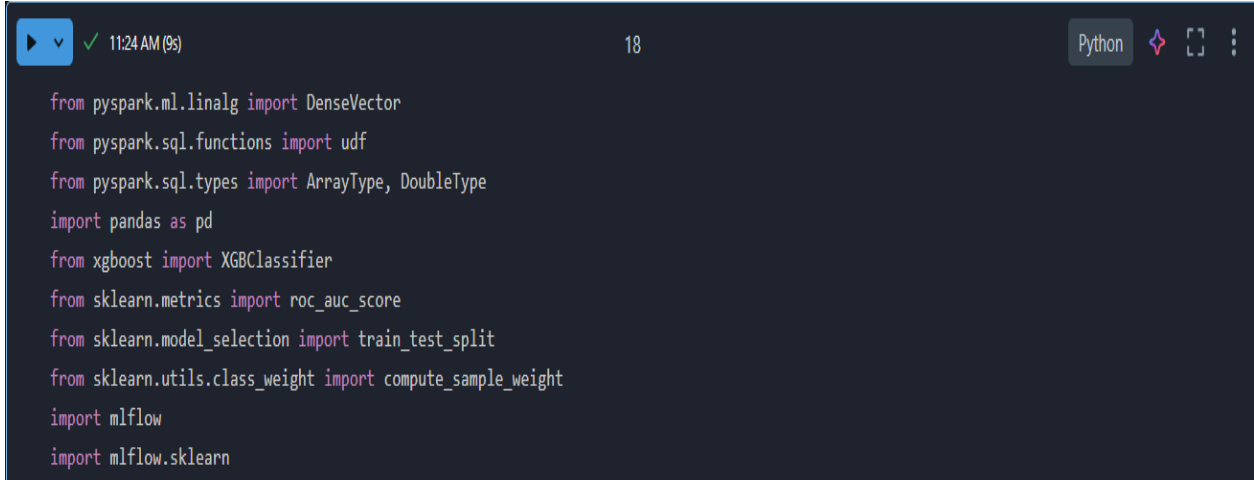
# Save Gold layer
df_gold.write.format("delta").mode("overwrite").saveAsTable("employee_attrition_gold")
```

▶ (22) Spark Jobs

8. XGBoost Model Training


This section explains how we train an XGBoost classifier to predict employee attrition using the features prepared from the Gold layer. The training is done using the scikit-learn API on top of pandas-converted Spark data, with proper handling of class imbalance and integration with MLflow.

8.1 Importing the necessary libraries



```
from pyspark.ml.linalg import DenseVector
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, DoubleType
import pandas as pd
from xgboost import XGBClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_sample_weight
import mlflow
import mlflow.sklearn
```

8.2 Read Gold Layer Data



```
# Load Gold data
df_gold = spark.read.table("employee_attrition_gold")
```

We read the feature-engineered data stored in the Gold layer.

8.3 Convert Features Vector to Array

```
vector_to_array_udf = udf(lambda vector: vector.toArray().toList(), ArrayType(DoubleType()))  
df_array = df_gold.withColumn("features_array", vector_to_array_udf("features"))
```

Since the features are stored as a vector, we convert them into arrays to extract individual columns.

8.4 Explode Feature Columns

```
feature_count = df_array.select("features_array").first()[0].__len__()  
feature_cols = [f"f{i}" for i in range(feature_count)]  
df_exploded = df_array.select(  
    "AttritionFlag", *[col("features_array")[i].alias(f"f{i}") for i in range(feature_count)]  
)
```

Generate (Ctrl + I)

We generate one column for each feature index (f0, f1, f2, ...) for modeling.

8.5 Convert to Pandas for Modeling

```
df_pd = df_exploded.toPandas()
```

The dataset is converted to pandas since XGBoost's scikit-learn API is used.

8.6 Train-Test Split

```
X = df_pd.drop(columns=["AttritionFlag"])
y = df_pd["AttritionFlag"]
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=42)
```

We split the dataset into training and testing sets while maintaining class proportions using stratify.

8.7 Handle Class Imbalance with Sample Weights

```
sample_weights = compute_sample_weight(class_weight="balanced", y=y_train)
```

This ensures that the model pays equal attention to both attrition classes despite any imbalance.

8.8 Train XGBoost Classifier

```
mlflow.set_experiment("/employee-attrition-experiment")
with mlflow.start_run(run_name="xgboost_sklearn_model"):

    xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
    xgb.fit(X_train, y_train, sample_weight=sample_weights)

    y_pred_proba = xgb.predict_proba(X_test)[:, 1]
    auc = roc_auc_score(y_test, y_pred_proba)

    mlflow.log_metric("test_auc", auc)
    mlflow.sklearn.log_model(xgb, "model")

    print(f"AUC: {auc}")
```

We train an XGBoost classifier with balanced class weights and log-loss as the evaluation metric.

8.9 Evaluation Metrics

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Convert predicted probabilities to binary predictions using a threshold of 0.5
y_pred = (y_pred_proba >= 0.5).astype(int)

# Metrics
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

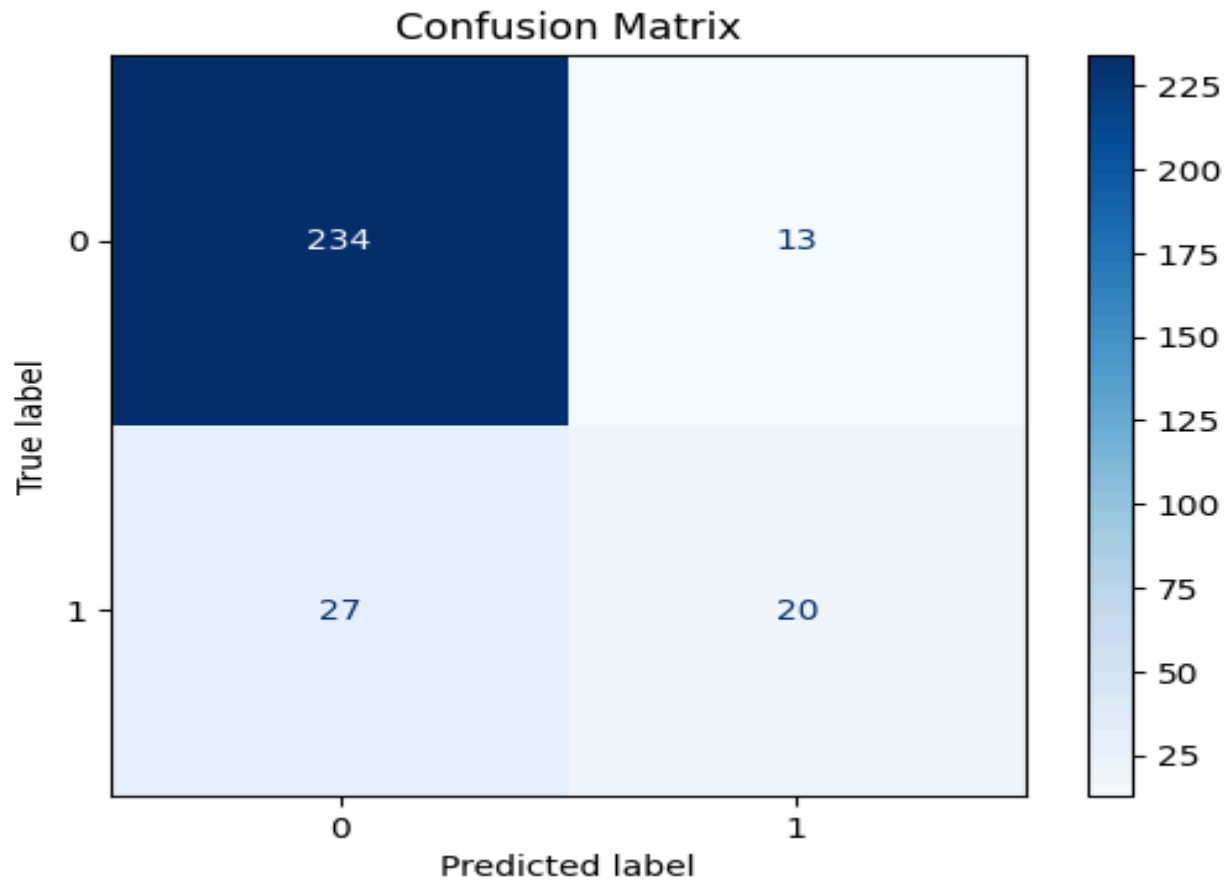
print(f"Accuracy: {acc:.3f}")
print(f"Precision: {prec:.3f}")
print(f"Recall: {rec:.3f}")
print(f"F1 Score: {f1:.3f}")

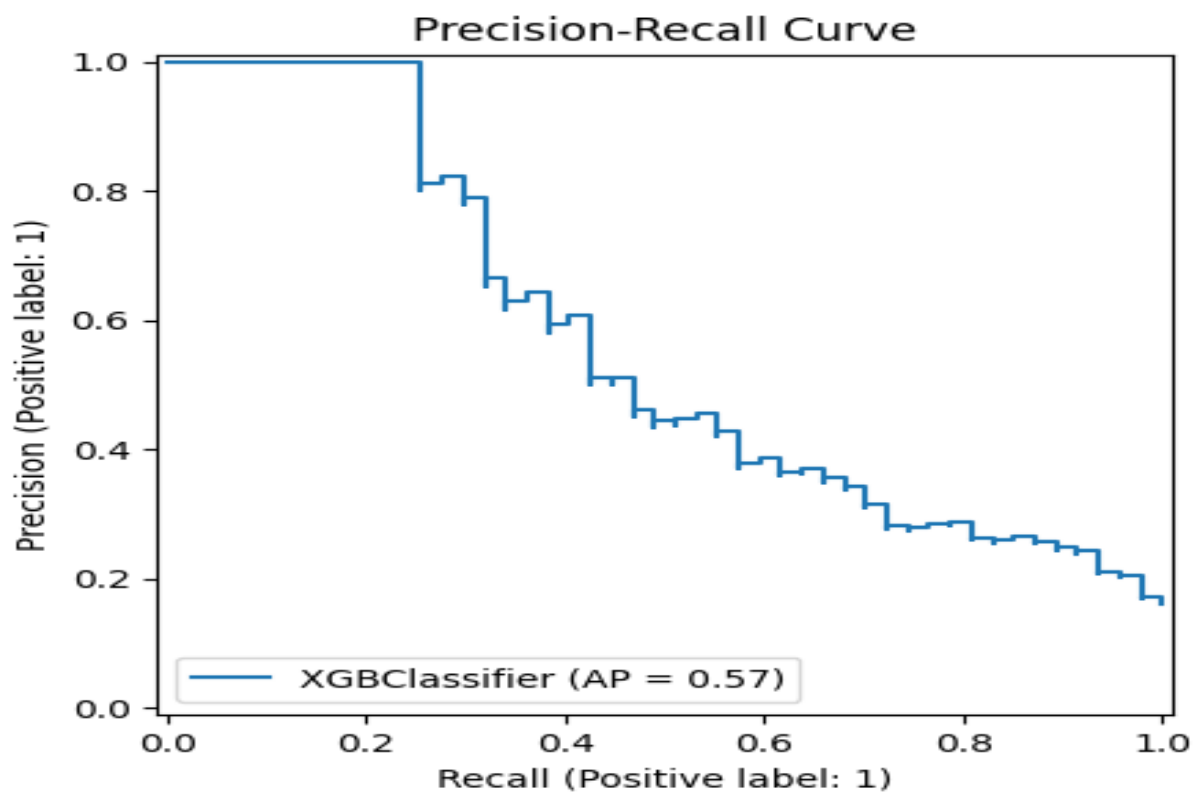
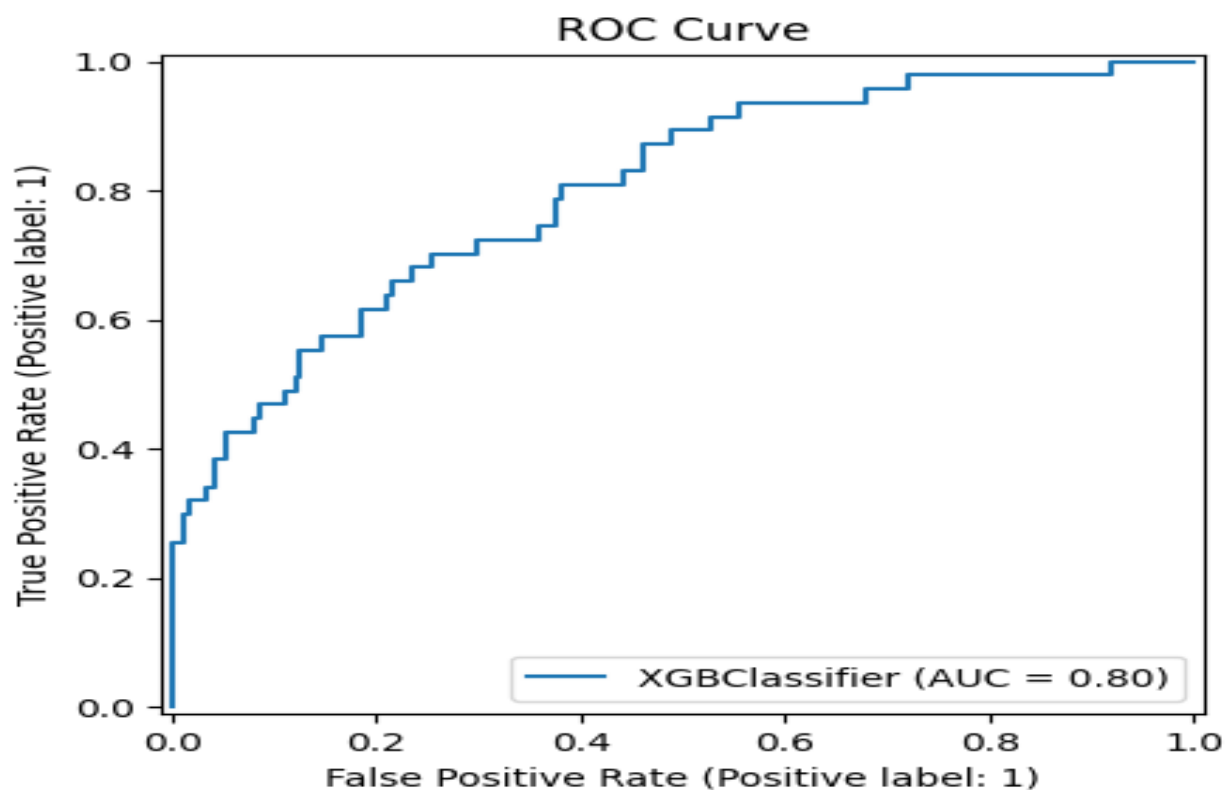
# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

From training the XGboost model we got the following metrics:

METRICS	PERCENTILE
Accuracy	0.864
Precision	0.606
Recall	0.426
F1 Score	0.500

8.10 Outputs and Graphs



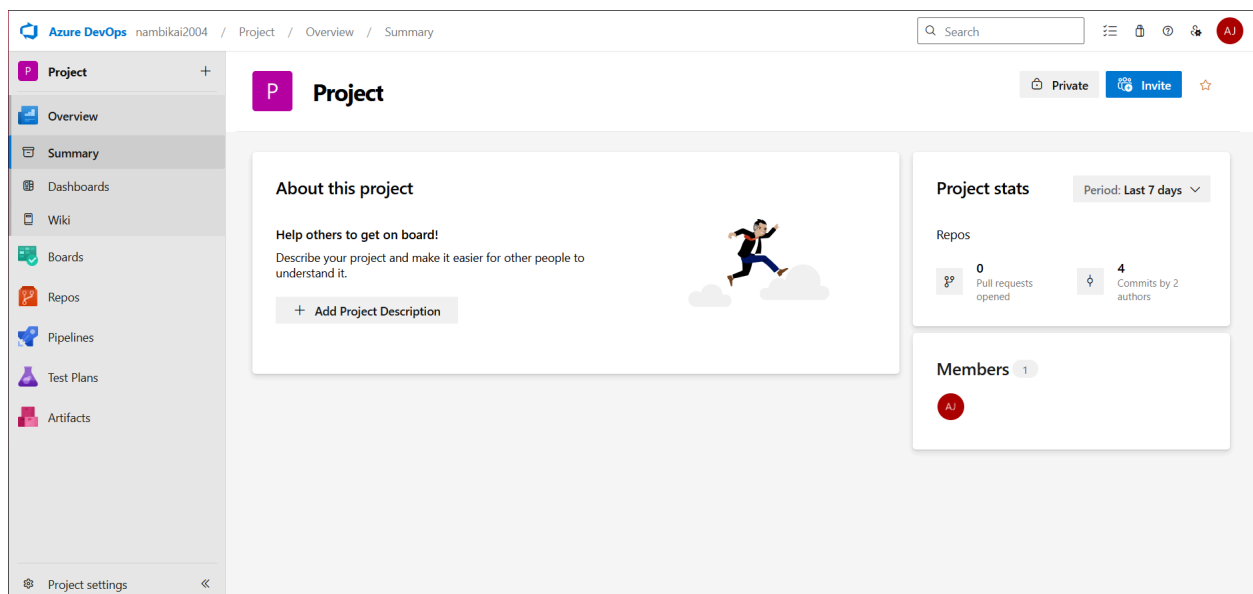


9. DevOps Version Control using Azure DevOps

To manage version control and collaboration during development, Azure DevOps was used for initializing a Git repository and maintaining notebook updates.

9.1 Project and Repository Setup

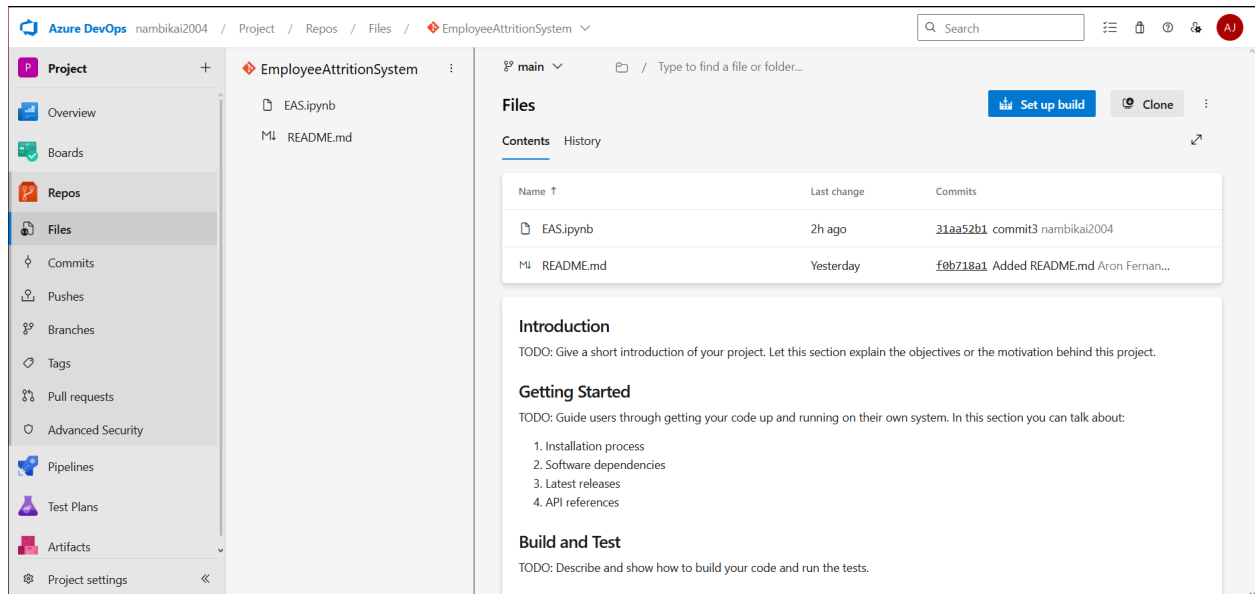
An Azure DevOps project was created under the user's organization. A Git repository was initialized within this project to store the code and notebooks related to the Employee Attrition System.



9.2 Cloning and Versioning

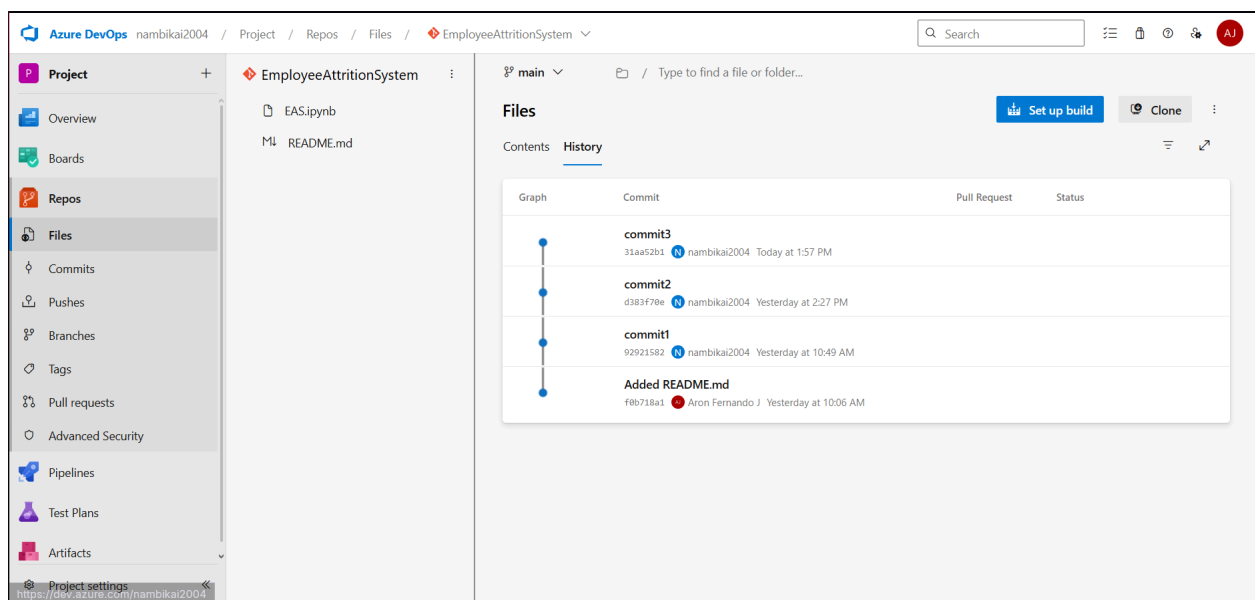
The repository was cloned into the development environment (e.g., Databricks or local Git). The main notebook (EAS.ipynb) and related files were added and committed using Git.

Regular commits were made to track changes during development. The repository helped in maintaining version history and supporting collaboration.



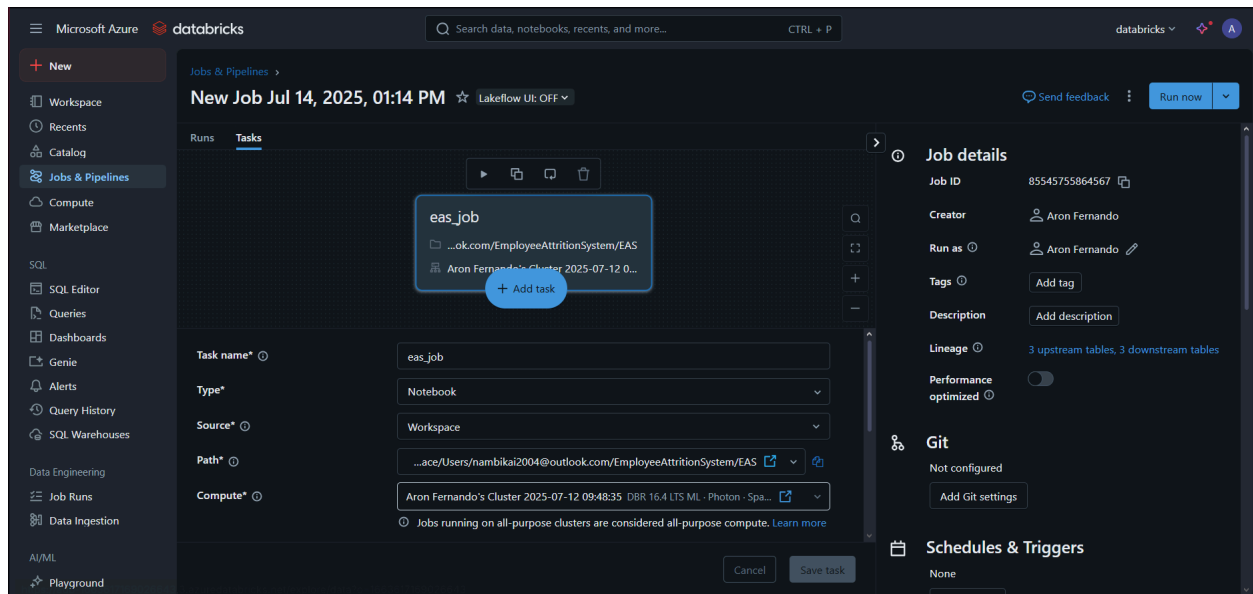
9.3 Git Operations

Basic Git commands like `git add`, `git commit`, and `git push` were used to sync local changes to the Azure DevOps repo. This ensured that all stages of the project (EDA, feature engineering, ML model) were consistently versioned.



10. Job Scheduling in Azure Databricks

To automate the execution of data pipelines and model training notebooks, Databricks Jobs were utilized. This enabled reliable and repeatable workflow scheduling directly within the Databricks environment.



Step 1: Navigate to Jobs Interface

From the Databricks workspace, select "Workflows" > "Jobs" in the left-hand navigation pane. Click on "Create Job" to start defining a new scheduled task.

Step 2: Configure Job Details

- Job Name: Provide a clear name (e.g., Employee Attrition Pipeline).
- Task Type: Select Notebook as the task type.

- Notebook Path: Browse and select the main notebook (e.g., /Repos/user/EAS.ipynb).
- Cluster: Choose an existing cluster or define a new one for job execution.

Step 3: Set Job Parameters

If the notebook accepts parameters (like file paths, flags), configure them under the "Parameters" section.

Step 4: Define Job Schedule

- Set a schedule frequency using cron syntax or built-in presets (e.g., daily, weekly).
- Configure start time and time zone as needed.

Step 5: Configure Notifications

Set up email or webhook alerts for success, failure, or skipped runs.

Step 6: Create and Run the Job

Click “Create” to save the job. You can manually trigger the job or wait for the scheduled run. The job status, output logs, and run history are visible in the Jobs UI.

Microsoft Azure

databricks

Search data, notebooks, recent, and more...CTRL + P

databricks

A

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

Jobs & Pipelines

New Job Jul 14, 2025, 01:14 PM

Lakeflow UI: OFF

Send feedback

Run now

RunsTasks

Runs

Include time in queue

Started before

< Previous

Next >

Run total duration

6m 40s

3m 20s

Jul 14

eas_job

Tasks

Go to the latest successful run

Cancel runs

Start time	Run ID	Launched	Duration	Spark	Status	Error code	Run param...	
Jul 14, 2025, 0...	3035075...	Manually	1m 10s	Spark UI / Logs / Metrics	✔ Succ...			
Jul 14, 2025, 0...	7106634...	Manually	4m 53s	Logs	❌ Failed	InvalidRunC...		

Job details

Job ID

85545755864567

Creator

Aron Fernando

Run as

Aron Fernando

Tags

Add tag

Description

Add description

Lineage

3 upstream tables, 3 downstream tables

Performance optimized

Git

Not configured

Add Git settings

Schedules & Triggers

None