Data Processing Pipeline with Azure Data Factory

Table of Contents

Project Statement:

Implement a serverless data processing pipeline where Azure Data Factory orchestrates data workflows, and Azure Databricks is used as a serverless processing engine for on-demand analytics and transformations.

Project Overview:

This project implements a comprehensive healthcare data analytics solution on Microsoft Azure. The pipeline processes raw healthcare data including patient information, doctor details, appointments, medical procedures, and billing records. The solution transforms this data into actionable insights through a multi-layered architecture that includes data ingestion, processing, transformation, and analytical reporting.

The system leverages Azure Data Factory for workflow orchestration and Azure Databricks as the serverless processing engine, creating an end-to-end serverless data pipeline that enables healthcare organizations to gain valuable insights into patient behavior, doctor performance, and operational efficiency.

Prerequisites:

1. **Azure Subscription:** Have an active Azure subscription for resource management.
2. **Azure Data Factory:** Create an Azure Data Factory instance.
3. **Azure Databricks:** Set up an Azure Databricks workspace for Spark processing.
4. **Azure Storage Account:** Create a storage account for CSV and Parquet files.
5. **Data Source:** Ensure the availability of the CSV file.
6. **Azure Access Key:** Obtain the Azure Access Key for the storage access
7. **Access Permissions:** Grant ADF permissions for storage access.
8. **Databricks Cluster:** Set up a Databricks cluster for Spark jobs.
9. **Libraries and Dependencies:** Install required libraries in both ADF and Databricks.
10. **Monitoring and Logging:** Set up monitoring in ADF and Databricks.

Azure Resources Used for this Project:

- Azure Blob Storage
- Azure Data Factory
- Azure Storage Account
- Databricks Notebook Activity
- Azure Delta Tables

Project Objectives:

- Establish automated healthcare data pipelines using Azure Data Factory.
- Convert raw CSV healthcare data into optimized Parquet format.
- Implement multi-layered data architecture (Raw → Processed → Analyzed).
- Perform comprehensive data transformations and quality checks.
- Generate actionable healthcare insights through advanced analytics.
- Optimize data processing performance using partitioning and compression techniques.
- Create maintainable and documented data processing workflows.

Tools Used:

- **Azure Data Factory (Orchestration):** Pipeline management and workflow coordination.
- **Azure Databricks (Transformation Engine):** PySpark-based data processing.
- **Azure Blob Storage (Data Storage):** Multi-container architecture for different data layers.
- **Delta Lake Format:** For optimized storage and query performance.
- **PySpark Libraries:** Data transformation, analysis, and optimization.
- **SQL Analytics:** Business intelligence and reporting queries.

Execution Overview:

1. Data Storage Architecture

- **Raw:** CSV healthcare data stored in Azure Blob (raw container).
- **Source:** Cleaned & Processed Parquet files stored in source container.
- **Destination:** Transformed results are stored in the destination container for reporting.

2. Orchestration with Azure Data Factory

- **Three Separate Pipelines:** Ingestion, Transformation, and Analysis were created individually.
- **Master Pipeline:** These pipelines are combined using Execute Pipeline activity for end-to-end orchestration.
- **Linked Services:** ADF securely connects to Azure Databricks and Blob Storage.
- **Execution Flow:** Pipelines run sequentially with dependency management.
- **Monitoring & Logging:** Execution status, performance, and errors are tracked for all runs.

3. Data Processing in Azure Databricks:

Notebook Execution: Multiple Python notebooks are executed within Databricks workspace:

- **Data Ingestion Notebooks:** Read CSV files from the raw container, perform data cleaning, validation, and store as Parquet files in the source container.
- **Data Transformation Notebooks:** Read processed data from the source container, apply business logic transformation, and store final results in the destination container.
- **Data Writing:** Store cleaned data as Parquet files in the source container and final transformed results in the destination container.

4. Scheduling and Monitoring

- **Pipeline Execution:** Pipelines are triggered manually.
- **Monitoring:** ADF and Databricks dashboards track execution status and performance.
- **Error Handling:** Retry and alert mechanisms ensure reliability.
- **Optimization:** Performance is reviewed to identify improvements.

Implementation-Tasks Performed:

1. Define Data Sources and Storage Architecture

- Identified healthcare CSV files (patients, doctors, appointments, procedures, billing) stored in the raw container of Azure Blob Storage.
- Designed a multi-layer storage model:

  - Source container has cleaned & processed Parquet files.
  - Destination container has final transformed datasets.
- Defined clear directory structures within containers for better organization.

2. Set Up Azure Data Factory (ADF)

- Built three separate pipelines:
  - **Ingestion:** Process raw CSV and store them as Parquet in source container.
  - **Transformation:** Perform aggregations and apply transformation functions, then store the final transformed data in the destination container.
  - **Analysis:** Run analytical SQL queries on the transformed data.
- Combined them into a master pipeline using Execute Pipeline activity.
- Configured linked services for Blob Storage, Databricks and access keys.
- Implemented dependency management to ensure correct execution order.

3. Develop Databricks Notebooks

- **Ingestion notebooks:**
  - Read raw healthcare CSV files (patients, doctors, appointments, billing, medical procedures) from the raw container, applied cleaning (standardization, formatting, text normalization), validation (null checks, schema enforcement), deduplication, and added audit columns like ingestion timestamp.
  - Saved the processed datasets in both Delta tables (healthcare_processed) for query optimization and Parquet format in the source container, partitioned where relevant (e.g., appointments by year).

- **Transformation notebooks:**
  - Applied business logic by joining patients, doctors, appointments, billing, and procedures, and created insights such as day type, day bucket, first appointment flag, revenue trends, specialization demand, workload classification, and doctor rankings.
  - Stored the curated datasets (patient_appointment_summary, specialization_summary) in the destination container as Parquet and Delta tables for further analysis and reporting.

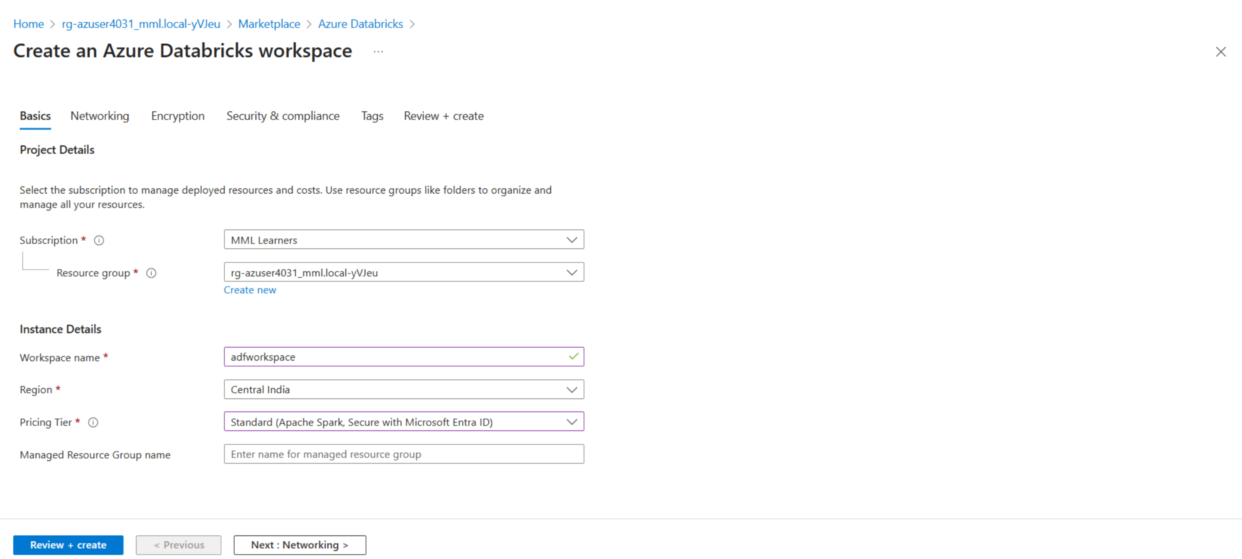
- **Analysis notebooks:**
  - Developed SQL-based analytical notebooks in Databricks on top of curated Delta/Parquet datasets. Generated key aggregations such as appointment distribution by day/week/month, first-time vs. returning patients, payment analysis for top patients, and demand trends across medical specializations.
  - Produced business insights including top-earning doctors, specialization workload and revenue ranking, and demand categorization. Final analytical results were structured and prepared for reporting and downstream decision-making.

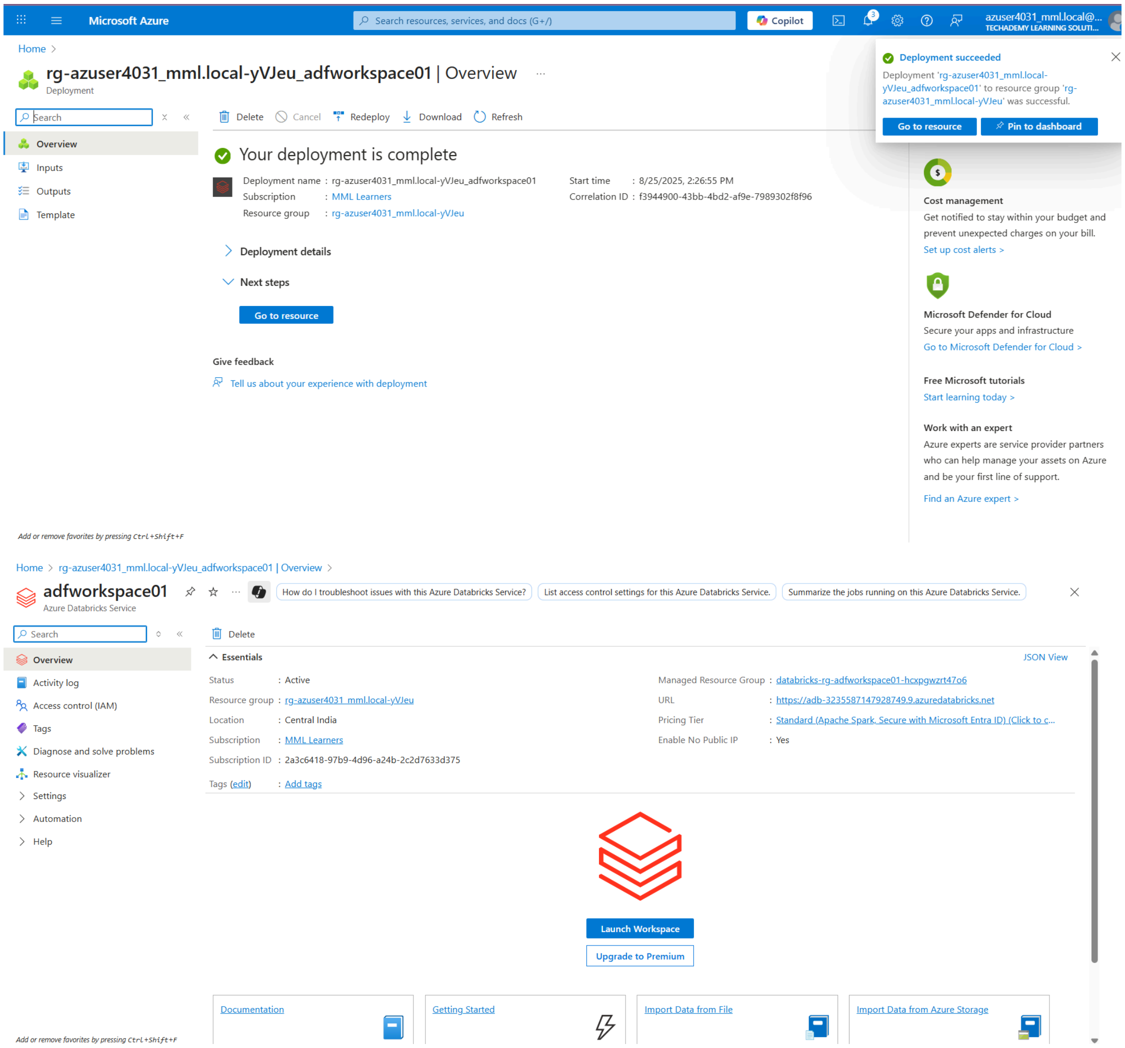
4. Optimization & Monitoring

- **Performance tuning:** Optimized Spark queries, memory usage, and cluster configs for efficiency.
- **Monitoring:** Used ADF and Databricks dashboards to track execution times, resource usage, and detect failures.
- **Error handling:** Added retry & alert mechanisms.

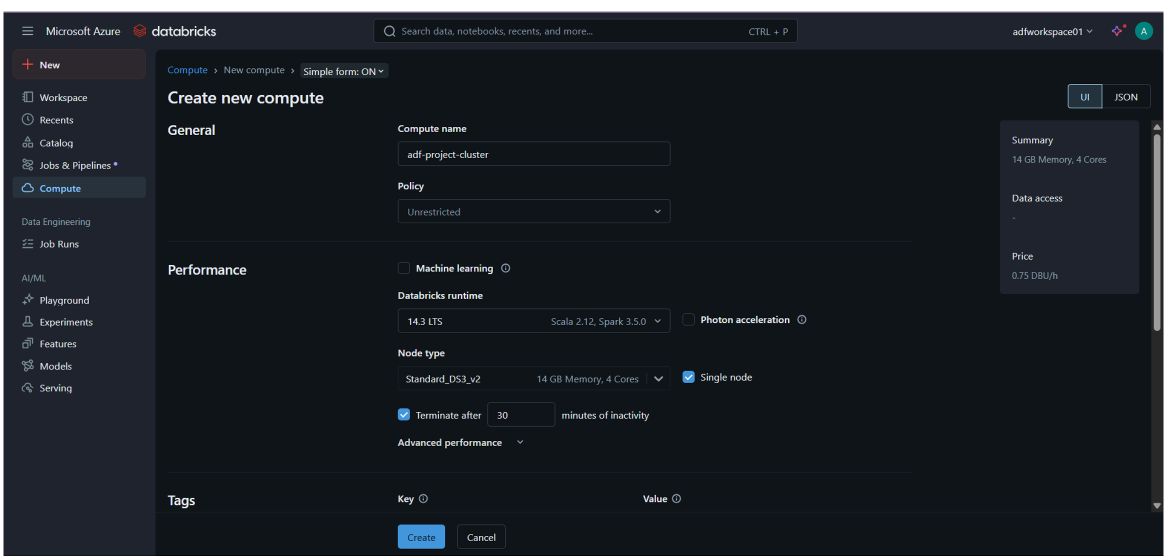
Practical Implementation on Azure Portal

Step 1: Set Up Azure Databricks



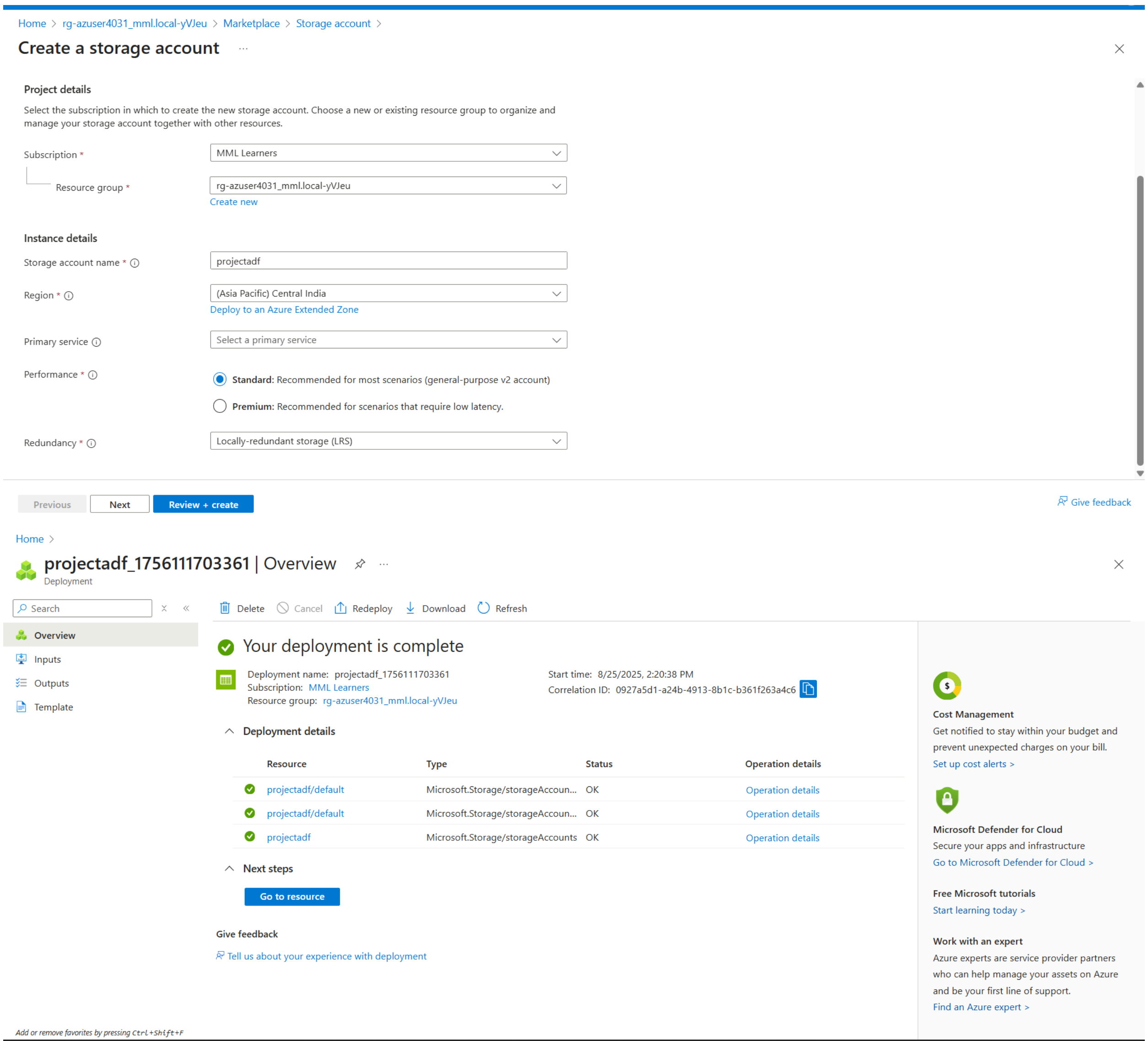


Create a Databricks workspace and a cluster in the Azure Portal.



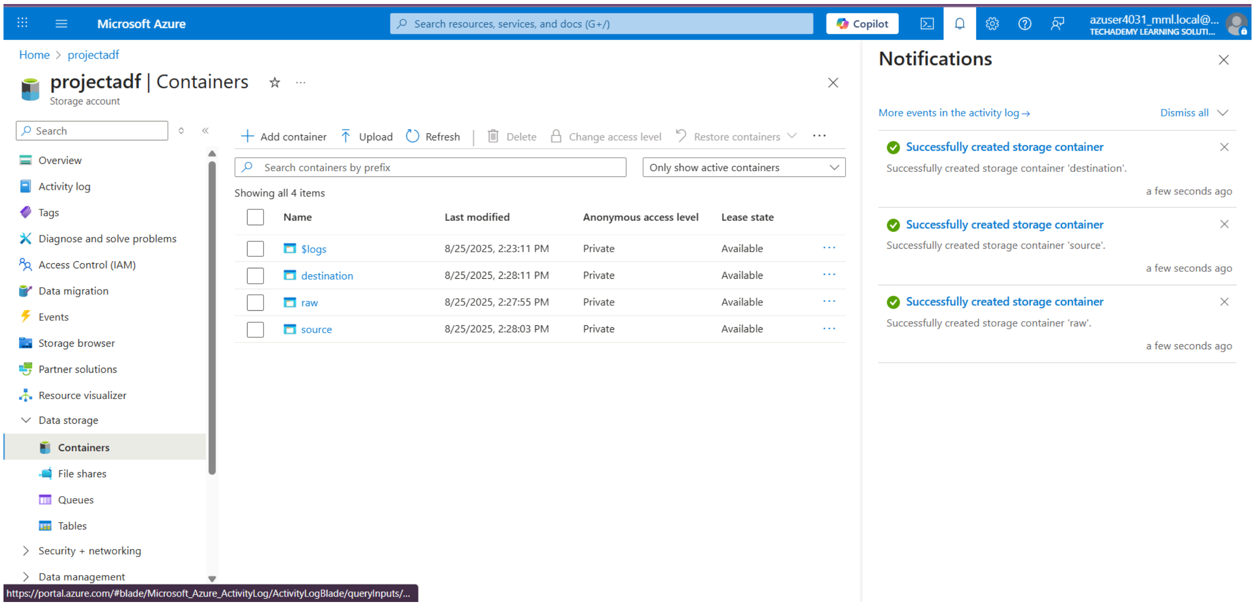
Step -2 Creating Azure Storage Account

Create an Azure Storage Account to store our files.



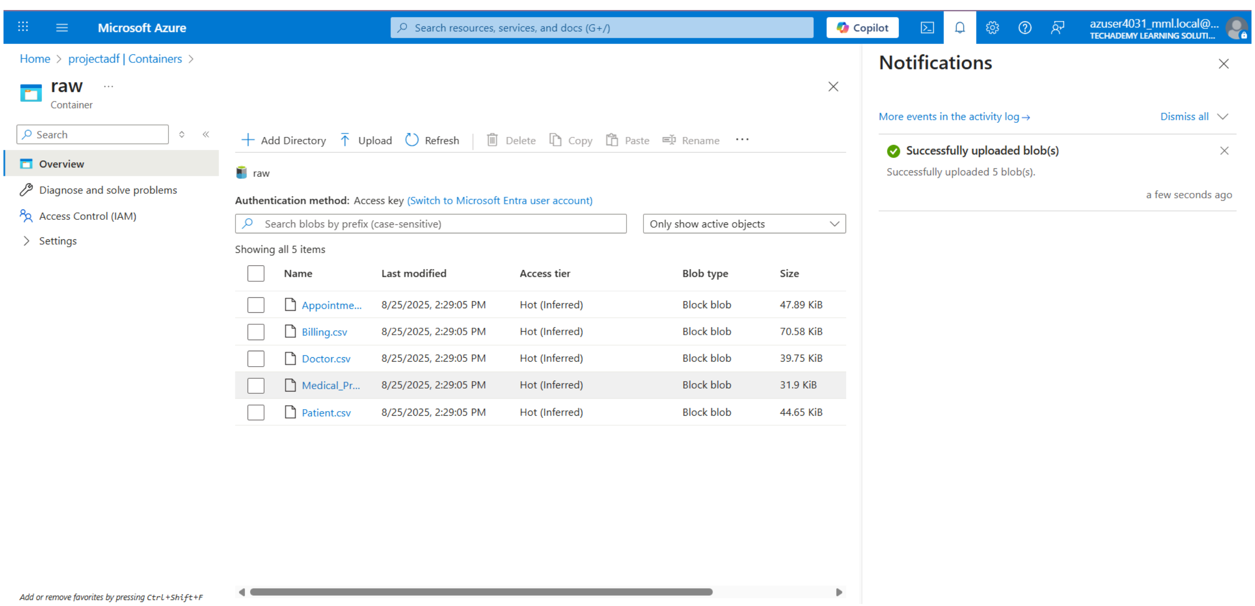
Create three containers

- Raw container holds the csv files
- Source container holds the cleaned and processed data
- Destination container holds the transformed data

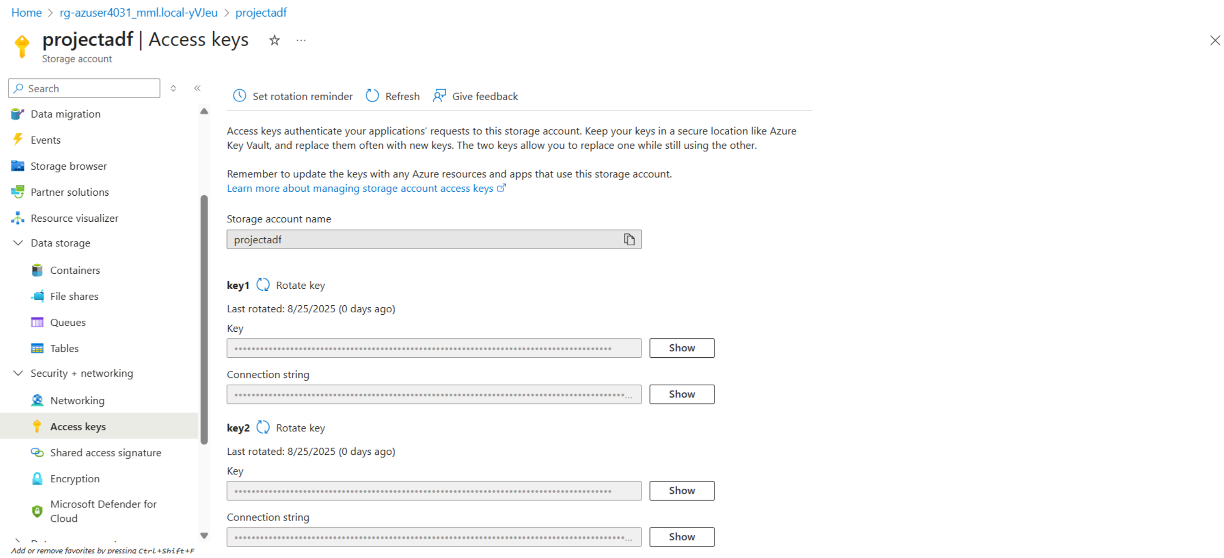


Uploading csv to the raw container

We used [Healthcare Management System dataset]() from kaggle

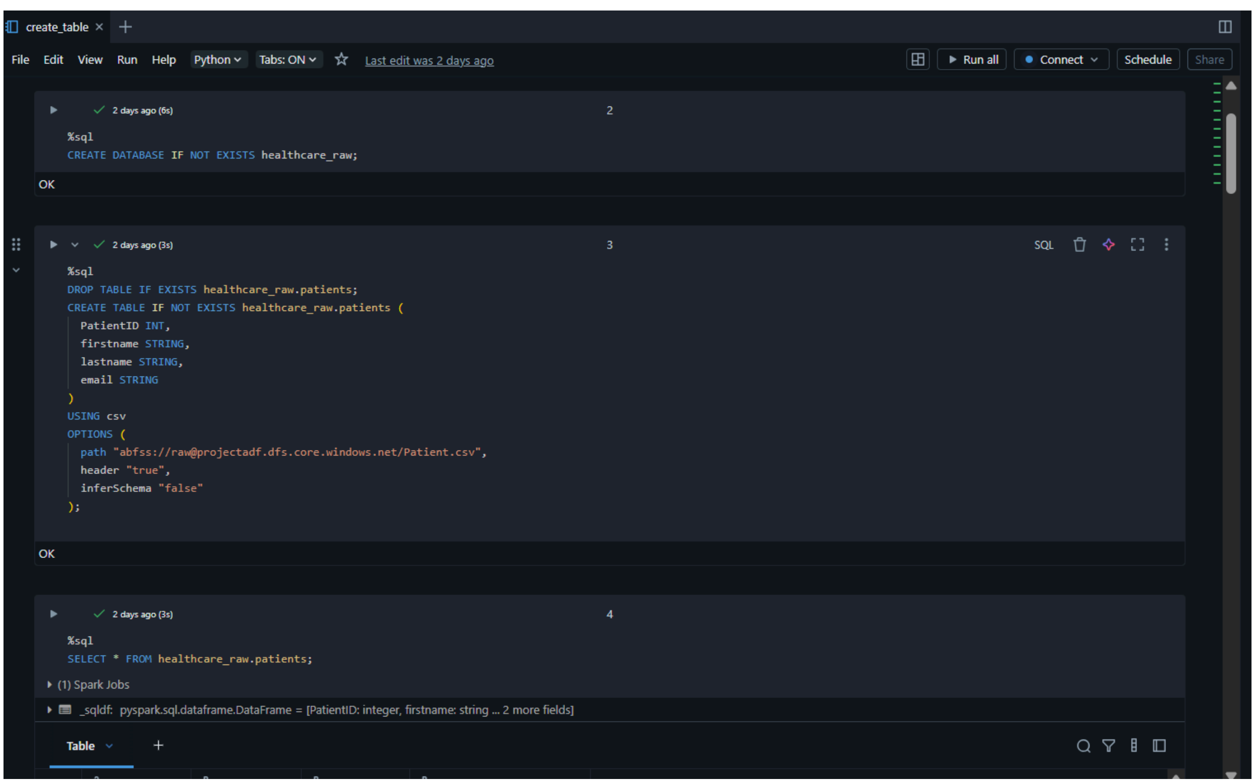


Get Access Key from Storage account to access the containers we have created



In the DataBricks WorkSpace, Create a new Notebook for creating the table for all the csv files we have in the raw container.

tables/create_table



create_table

File Edit View Run Help Python Tabs: ON Last edit was 2 days ago Run all Connect Schedule

2 days ago (1s) 7

```sql
%sql
DROP TABLE IF EXISTS healthcare_raw.appointments;
CREATE TABLE IF NOT EXISTS healthcare_raw.appointments (
  AppointmentID INT,
  Date DATE,
  Time STRING,
  PatientID INT,
  DoctorID INT
)
USING csv
OPTIONS (
  path "abfss://raw@projectadf.dfs.core.windows.net/Appointment.csv",
  header "true",
  inferSchema "false"
);
```

OK

2 days ago (1s) 8 SQL

```sql
%sql
SELECT * FROM healthcare_raw.appointments;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [AppointmentID: integer, Date: date ... 3 more fields]

Table

	AppointmentID	Date	Time	PatientID	DoctorID
1	639	2022-04-08	2023-12-23T14:33:46.40...	109	462
2	404	2023-04-10	2023-12-23T14:33:46.40...	823	774
3	281	2022-12-09	2023-12-23T14:33:46.40...	465	226

create_table

File Edit View Run Help Python Tabs: ON Last edit was 2 days ago Run all Connect Schedule

2 days ago (1s) 9 SQL

```sql
%sql
DROP TABLE IF EXISTS healthcare_raw.medical_procedure;
CREATE TABLE IF NOT EXISTS healthcare_raw.medical_procedure (
  ProcedureID INT,
  ProcedureName STRING,
  AppointmentID INT
)
USING csv
OPTIONS (
  path "abfss://raw@projectadf.dfs.core.windows.net/Medical_Procedure.csv",
  header "true",
  inferSchema "false"
);
```

OK

2 days ago (1s) 10

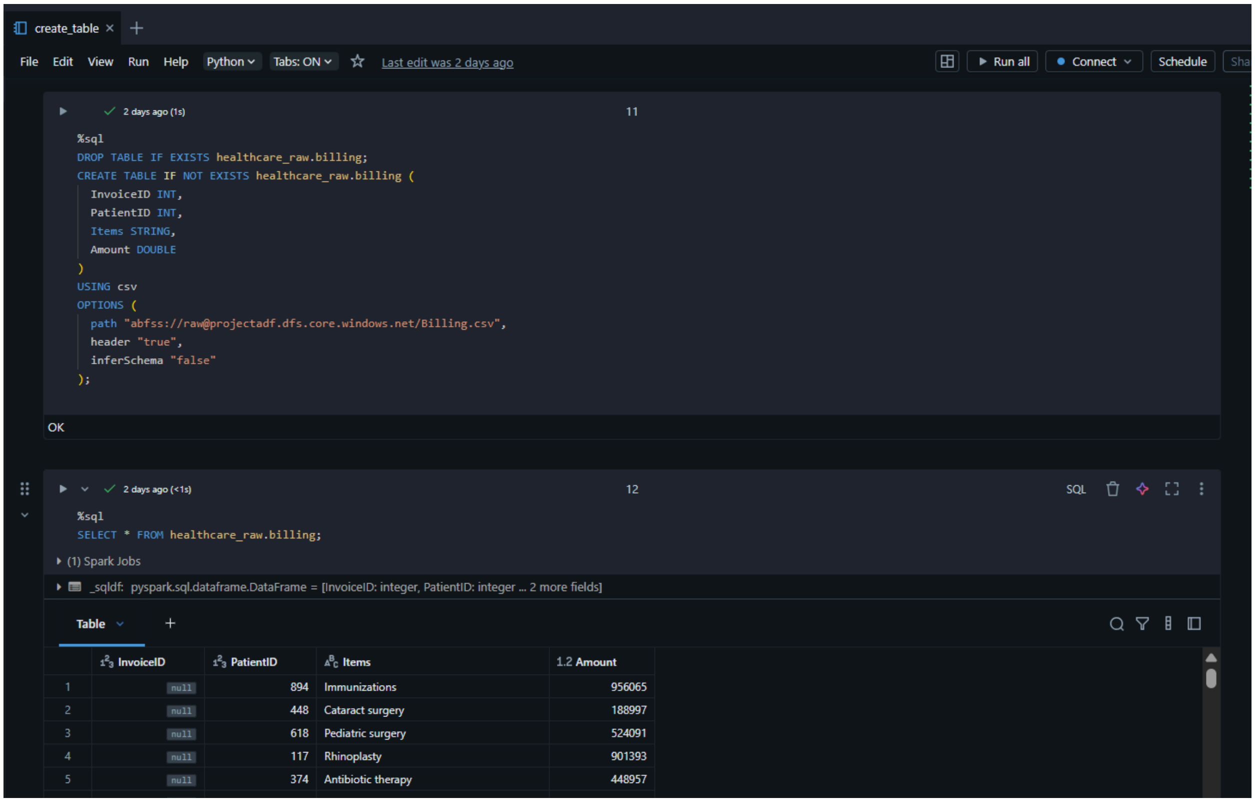
```sql
%sql
SELECT * FROM healthcare_raw.medical_procedure;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [ProcedureID: integer, ProcedureName: string ... 1 more field]

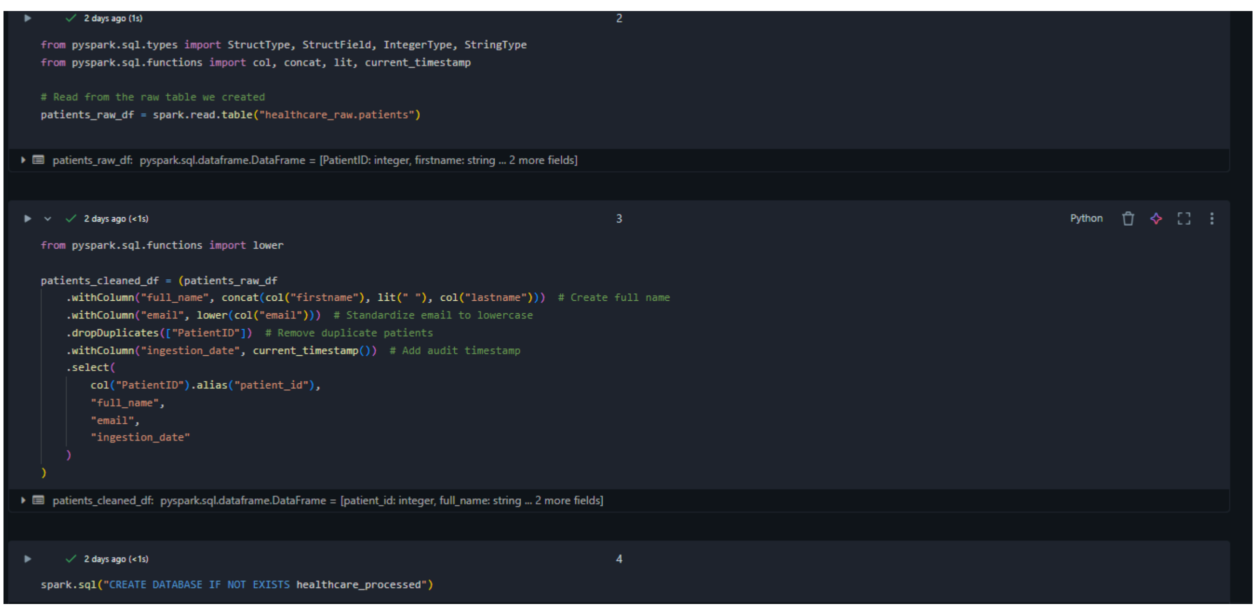
Table

	ProcedureID	ProcedureName	AppointmentID
1	432	Kidney transplant	955
2	574	Allergy testing	701
3	854	Psychotherapy	363
4	818	Emotional and spiritual support	959
5	672	Hormone replacement therapy	439
6	869	Coronary artery bypass surgery	805



Create 5 more notebooks called ingest_patients, ingest_billing, ingest_appointments, ingest_medical_procedure, ingest_doctor for ingesting the data to the source container.

ingestion/ingest_patient



✓ 2 days ago (17s) 5

```python
patients_cleaned_df.write.mode("overwrite").format("delta").saveAsTable("healthcare_processed.patients")

patients_cleaned_df.write.mode("overwrite").parquet(f"abfss://source@projectadf.dfs.core.windows.net/patients")
```

(9) Spark Jobs

✓ 2 days ago (2s) 6 Python

```python
display(spark.read.table("healthcare_processed.patients"))
print("Patients ingestion completed successfully!")
```

(2) Spark Jobs

Table

	patient_id	full_name	email	ingestion_date
1	101	Ira Eachern	ira.eachern@yopmail.com	2025-08-25T09:23:19.083+00:...
2	103	Lusa Alisia	lusa.alisia@yopmail.com	2025-08-25T09:23:19.083+00:...
3	104	Joane Yam	joane.yam@yopmail.com	2025-08-25T09:23:19.083+00:...
4	106	Petronia Tamsky	petronia.tamsky@yopmail.com	2025-08-25T09:23:19.083+00:...
5	108	Antonietta Claudine	antonietta.claudine@yopmail.com	2025-08-25T09:23:19.083+00:...
6	111	Leontine Zachary	leontine.zachary@yopmail.com	2025-08-25T09:23:19.083+00:...
7	112	Elbertina Dosia	elbertina.dosia@yopmail.com	2025-08-25T09:23:19.083+00:...
8	115	Blondelle Riordan	blondelle.riordan@yopmail.com	2025-08-25T09:23:19.083+00:...
9	116	Babita Felecia	babita.felecia@yopmail.com	2025-08-25T09:23:19.083+00:...

ingestion/ingest_doctor

✓ 2 days ago (<1s) 2

```python
from pyspark.sql.functions import col, trim, lower, initcap, current_timestamp
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

doctors_raw_df = spark.read.table("healthcare_raw.doctors")
```

doctors_raw_df: pyspark.sql.dataframe.DataFrame = [DoctorID: integer, DoctorName: string ... 2 more fields]

✓ 2 days ago (<1s) 3 Python

```python
doctors_cleaned_df = (doctors_raw_df
    # Clean text fields
    .withColumn("doctor_name", trim(initcap(col("DoctorName"))))  # Capitalize first letters and trim
    .withColumn("specialization", trim(lower(col("Specialization"))))  # Standardize to lowercase
    .withColumn("contact_info", trim(col("DoctorContact")))

    # Data quality: Remove doctors without name or specialization
    .filter(col("DoctorName").isNotNull() & col("Specialization").isNotNull())

    # Remove duplicates
    .dropDuplicates(["DoctorID"])

    # Add audit columns
    .withColumn("ingestion_date", current_timestamp())

    # Select and rename final columns
    .select(
        col("DoctorID").alias("doctor_id"),
        "doctor_name",
        "specialization",
        "contact_info",
        "ingestion_date"
    )
)
```

2 days ago (5s) 4 Python

```python
doctors_cleaned_df.write.mode("overwrite").format("delta").saveAsTable("healthcare_processed.doctors")

doctors_cleaned_df.write.mode("overwrite").parquet(f"abfss://source@projectadf.dfs.core.windows.net/doctors")
```

(9) Spark Jobs

2 days ago (1s) 5

```python
print("Total doctors processed:", doctors_cleaned_df.count())
display(doctors_cleaned_df.limit(5))
```

(5) Spark Jobs

```
Total doctors processed: 600
```

Table

	doctor_id	doctor_name	specialization	contact_info	ingestion_date
1	100	Thalia	emergency medicine	.@yopmail.com	2025-08-25T09:25:29.365+00:...
2	101	Mireielle	allergists	.@yopmail.com	2025-08-25T09:25:29.365+00:...
3	103	Elie	endocrinologist	.@yopmail.com	2025-08-25T09:25:29.365+00:...
4	104	Cacilie	emergency medicine	.@yopmail.com	2025-08-25T09:25:29.365+00:...
5	105	Lynea	emergency medicine	.@yopmail.com	2025-08-25T09:25:29.365+00:...

ingest/ingest_appointment

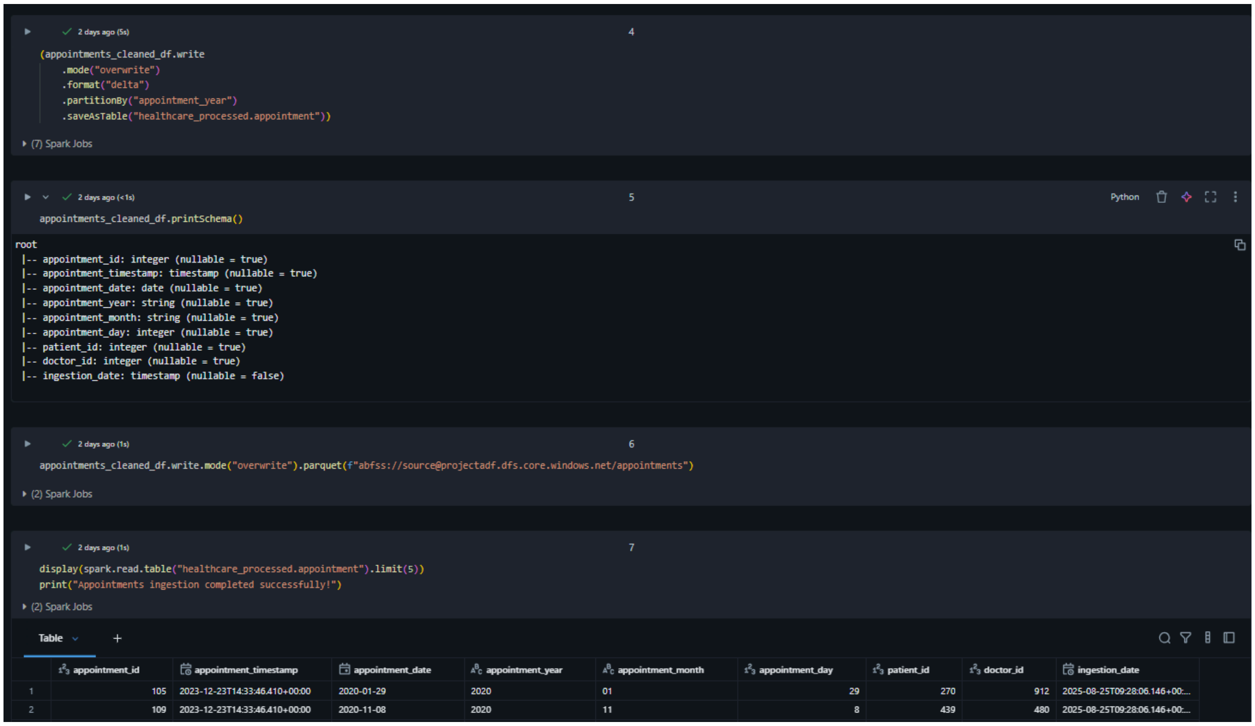
2 days ago (<1s) 2

```python
from pyspark.sql.functions import col, to_timestamp, concat, date_format, current_timestamp
appointments_raw_df = spark.read.table("healthcare_raw.appointments")
```

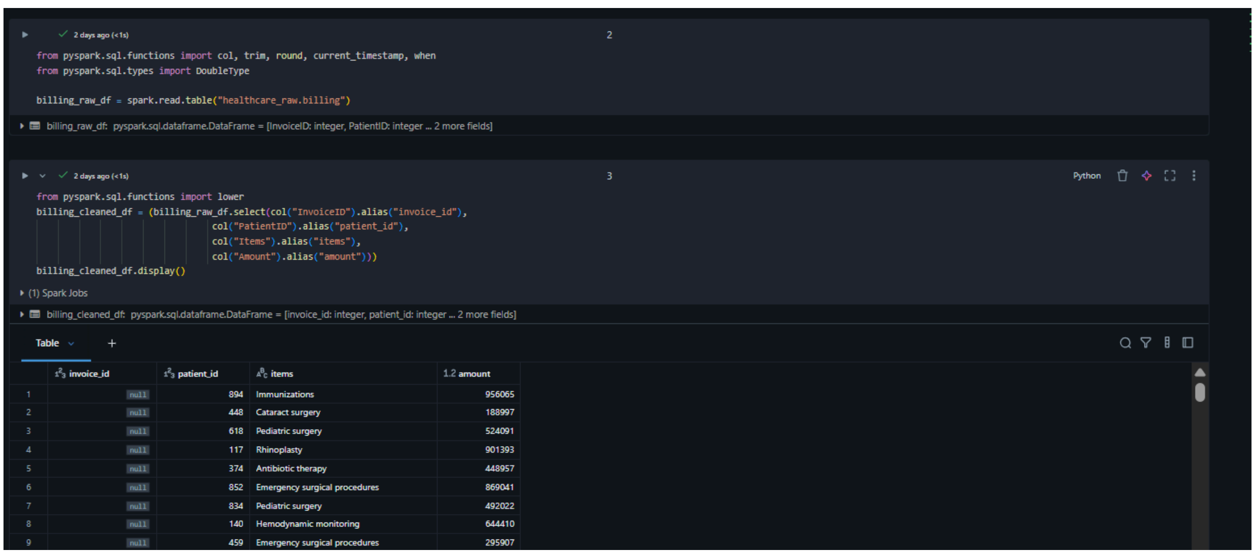
appointments_raw_df: pyspark.sql.dataframe.DataFrame = [AppointmentID: integer, Date: date ... 3 more fields]

2 days ago (<1s) 3 Python

```python
from pyspark.sql.functions import col, to_date, to_timestamp, date_format, dayofmonth, current_timestamp

appointments_cleaned_df = (appointments_raw_df
    # Keep Date column as-is (string from CSV)
    .withColumn("appointment_date", col("Date"))

    # Convert to proper date for extracting parts
    .withColumn("date_parsed", to_date(col("Date"), "dd-MM-yyyy"))

    # Extract year, month, and day
    .withColumn("appointment_year", date_format(col("date_parsed"), "yyyy"))
    .withColumn("appointment_month", date_format(col("date_parsed"), "MM"))
    .withColumn("appointment_day", dayofmonth(col("date_parsed")))

    # Parse Time column (ISO timestamp)
    .withColumn("appointment_timestamp", to_timestamp(col("Time"), "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"))

    # Add ingestion timestamp
    .withColumn("ingestion_date", current_timestamp())

    # Deduplicate
    .dropDuplicates(["AppointmentID"])

    # Final selection
    .select(
        col("AppointmentID").alias("appointment_id"),
        "appointment_timestamp",
        "appointment_date",       # original string from CSV
        "appointment_year",
        "appointment_month",
        "appointment_day",        # new day column (int 1-31)
        col("PatientID").alias("patient_id"),
        col("DoctorID").alias("doctor_id"),
        "ingestion_date"
    )
)
```



ingestion/ingest_billing



Step-3 : Create a Data Factory

Create a new Azure Data Factory for performing the file conversion.

```python
(billing_cleaned_df.write
    .mode("overwrite")
    .format("delta")
    .saveAsTable("healthcare_processed.billing"))

# Write to cloud storage
billing_cleaned_df.write.mode("overwrite").parquet(f"abfss://source@projectadf.dfs.core.windows.net/billing")
```

(7) Spark Jobs

```python
from pyspark.sql.functions import sum, count,avg

summary_df = billing_cleaned_df.agg(
    count("*").alias("total_invoices"),
    sum("amount").alias("total_revenue"),
    round(avg("amount"), 2).alias("average_invoice_amount")
)

print("Billing Summary:")
display(summary_df)
```

(2) Spark Jobs

summary_df: pyspark.sql.dataframe.DataFrame = [total_invoices: long, total_revenue: double ... 1 more field]

Billing Summary:

	total_invoices	total_revenue	average_invoice_amount
1	1000	510275708	510275.71

ingestion/ingest_medical_procedure

```python
from pyspark.sql.functions import col, trim, lower, initcap, current_timestamp

procedures_raw_df = spark.read.table("healthcare_raw.medical_procedure")
```

procedures_raw_df: pyspark.sql.dataframe.DataFrame = [ProcedureID: integer, ProcedureName: string ... 1 more field]

```python
procedures_cleaned_df = (procedures_raw_df
    # Clean text fields
    .withColumn("procedure_name", trim(initcap(col("ProcedureName"))))

    # Data validation: Ensure required fields are present
    .filter(col("ProcedureName").isNotNull() & col("AppointmentID").isNotNull())

    # Remove duplicates (same procedure for same appointment shouldn't happen)
    .dropDuplicates(["ProcedureID", "AppointmentID"])

    # Add audit column
    .withColumn("ingestion_date", current_timestamp())

    # Select and rename final columns
    .select(
        col("ProcedureID").alias("procedure_id"),
        "procedure_name",
        col("AppointmentID").alias("appointment_id"),
        "ingestion_date"
    )
)
```

procedures_cleaned_df: pyspark.sql.dataframe.DataFrame = [procedure_id: integer, procedure_name: string ... 2 more fields]

```python
procedures_cleaned_df.write.mode("overwrite").format("delta").saveAsTable("healthcare_processed.medical_procedures")

# Write to cloud storage
procedures_cleaned_df.write.mode("overwrite").parquet(f"abfss://source@projectadf.dfs.core.windows.net/medical_procedures")
```

(9) Spark Jobs

To perform transformation, inside transformation folder create two notebooks

transformation/patient_appointment_summary

Read Source Tables

2 days ago (1s) 3 Python

```python
# Read appointments data
appointments_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/appointments")
patients_df =  spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/patients")
doctors_df =  spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/doctors")
billing_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/billing")
procedures_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/medical_procedures")
```

(5) Spark Jobs

- appointments_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 7 more fields]
- billing_df: pyspark.sql.dataframe.DataFrame = [invoice_id: integer, patient_id: integer ... 2 more fields]
- doctors_df: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, doctor_name: string ... 3 more fields]
- patients_df: pyspark.sql.dataframe.DataFrame = [patient_id: integer, full_name: string ... 2 more fields]
- procedures_df: pyspark.sql.dataframe.DataFrame = [procedure_id: integer, procedure_name: string ... 2 more fields]

Build Base Data (Joins)

2 days ago (1s) 5

```python
billing_df.printSchema()
billing_df.show(20, truncate=False)
billing_df.select("patient_id").distinct().show()
```

(3) Spark Jobs

```
|       471|
|       496|
|       623|
|       858|
|       897|
|       243|
```

2 days ago (<1s) 6 Python

```python
# Join appointments with patients
appointment_patient_df = appointments_df.join(
    patients_df,
    appointments_df.patient_id == patients_df.patient_id,
    "inner"
).select(
    appointments_df["*"],
    patients_df["full_name"].alias("patient_name"),
    patients_df["email"].alias("patient_email")
)
# Join with doctors
appointment_doctor_df = appointment_patient_df.join(
    doctors_df,
    appointment_patient_df.doctor_id == doctors_df.doctor_id,
    "inner"
).select(
    appointment_patient_df["*"],
    doctors_df["doctor_name"],
    doctors_df["specialization"],
    doctors_df["contact_info"].alias("doctor_contact")
)
# Join with procedures
appointment_procedure_df = appointment_doctor_df.join(
    procedures_df,
    appointment_doctor_df.appointment_id == procedures_df.appointment_id,
    "left"
).select(
    appointment_doctor_df["*"],
    procedures_df["procedure_name"],
    procedures_df["procedure_id"]
)
# Join with billing (aggregate billing by appointment)
from pyspark.sql.functions import sum, coalesce, lit

billing_agg_df = billing_df.groupBy("patient_id").agg(
    sum("amount").alias("total_amount"),
    sum(coalesce(col("amount"), lit(0))).alias("total_billing")
)
appointment_summary_base_df = appointment_procedure_df.join(
    billing_agg_df,
    appointment_procedure_df.patient_id == billing_agg_df.patient_id,
    "left"
).drop(billing_agg_df.patient_id)
```

Add Day Name (Text Format of Date)

2 days ago (2s) 8 Python

```python
from pyspark.sql.functions import col, date_format

appointment_summary_df =appointment_summary_base_df.withColumn(
    "day_name", date_format(col("appointment_date"), "EEEE")
)
appointment_summary_df.display()
```

(6) Spark Jobs

appointment_summary_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 17 more fields]

Table +

	appointment_id	appointment_timestamp	appointment_date	appointment_year	appointment_month	appointment_day	patient_id	doctor_id	ingestion_date	
1	531	2023-12-23T14:33:46.412+00:00	2022-06-13	2022	06	13	103	291	2025-08-25T09:28:10.877+00:...	Lus
2	455	2023-12-23T14:33:46.410+00:00	2023-03-15	2023	03	15	103	388	2025-08-25T09:28:10.877+00:...	Lus
3	656	2023-12-23T14:33:46.414+00:00	2023-07-25	2023	07	25	112	612	2025-08-25T09:28:10.877+00:...	Elbe

First Appointment Flag

2 days ago (1s) 10 Python

```python
from pyspark.sql import Window
from pyspark.sql.functions import row_number, when,col
# Transformation 3: Flag is_first_appointment for each patient
# Create window specification by patient_id ordered by appointment timestamp
patient_window = Window.partitionBy("patient_id").orderBy("appointment_timestamp")

appointment_summary_df = appointment_summary_df.withColumn(
    "appointment_rank",
    row_number().over(patient_window)
).withColumn(
    "is_first_appointment",
    when(col("appointment_rank") == 1, True).otherwise(False)
).drop("appointment_rank")
appointment_summary_df.display()
```

(7) Spark Jobs

appointment_summary_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 18 more fields]

Table +

	appointment_id	appointment_timestamp	appointment_date	appointment_year	appointment_month	appointment_day	patient_id	doctor_id	ingestion_date	
1	455	2023-12-23T14:33:46.410+00:00	2023-03-15	2023	03	15	103	388	2025-08-25T09:28:10.877+00:...	Lus
2	531	2023-12-23T14:33:46.412+00:00	2022-06-13	2022	06	13	103	291	2025-08-25T09:28:10.877+00:...	Lus
3	443	2023-12-23T14:33:46.408+00:00	2022-11-27	2022	11	27	112	210	2025-08-25T09:28:10.877+00:...	Elbe
4	619	2023-12-23T14:33:46.412+00:00	2022-06-16	2022	06	16	112	926	2025-08-25T09:28:10.877+00:...	Elbe

Day Bucket (Early / Mid / Late Month)

2 days ago (1s) 12 Python

```python
from pyspark.sql.functions import dayofmonth

appointment_summary_df = appointment_summary_df.withColumn(
    "appointment_day_num", dayofmonth("appointment_date")
).withColumn(
    "day_bucket",
    when(col("appointment_day_num") <= 10, "Early Month")
    .when((col("appointment_day_num") > 10) & (col("appointment_day_num") <= 20), "Mid Month")
    .otherwise("Late Month")
)
appointment_summary_df.display()
```

(7) Spark Jobs

appointment_summary_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 20 more fields]

Table +

	appointment_id	appointment_timestamp	appointment_date	appointment_year	appointment_month	appointment_day	patient_id	doctor_id	ingestion_date	
1	455	2023-12-23T14:33:46.410+00:00	2023-03-15	2023	03	15	103	388	2025-08-25T09:28:10.877+00:...	Lus
2	531	2023-12-23T14:33:46.412+00:00	2022-06-13	2022	06	13	103	291	2025-08-25T09:28:10.877+00:...	Lus
3	443	2023-12-23T14:33:46.408+00:00	2022-11-27	2022	11	27	112	210	2025-08-25T09:28:10.877+00:...	Elbe

Day Type (Weekday vs Weekend)

2 days ago (1s) 14 Python

```
from pyspark.sql.functions import dayofweek

appointment_summary_df = appointment_summary_df.withColumn(
    "day_of_week_num", dayofweek("appointment_date")
).withColumn(
    "day_type",
    when((col("day_of_week_num") == 1) | (col("day_of_week_num") == 7), "Weekend")
    .otherwise("Weekday")
)
appointment_summary_df.display()
```

(7) Spark Jobs

appointment_summary_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 22 more fields]

Table +

	appointment_id	appointment_timestamp	appointment_date	appointment_year	appointment_month	appointment_day	patient_id	doctor_id	ingestion_date	
1	455	2023-12-23T14:33:46.410+00:00	2023-03-15	2023	03	15	103	388	2025-08-25T09:28:10.877+00:...	Lus
2	531	2023-12-23T14:33:46.412+00:00	2022-06-13	2022	06	13	103	291	2025-08-25T09:28:10.877+00:...	Lus
3	443	2023-12-23T14:33:46.408+00:00	2022-11-27	2022	11	27	112	210	2025-08-25T09:28:10.877+00:...	Elbe

2 days ago (1s) 15 Python

```
# Build the final curated DataFrame
final_summary_df = appointment_summary_df.select(
    "appointment_id",
    "appointment_timestamp",
    "appointment_date",
    "appointment_year",
    "appointment_month",
    "patient_id",
    "patient_name",
    "patient_email",
    "doctor_id",
    "doctor_name",
    "specialization",
    "doctor_contact",
    "procedure_id",
    "procedure_name",
    "total_amount",
    "day_name",
    "appointment_day_num",
    "day_bucket",
    "day_of_week_num",
    "day_type",
    "is_first_appointment",
    "ingestion_date"
)

# Preview
final_summary_df.display()
```

(7) Spark Jobs

final_summary_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 20 more fields]

Table +

	appointment_id	appointment_timestamp	appointment_date	appointment_year	appointment_month	patient_id	patient_name	patient_email	doctor_id	do
15	860	2023-12-23T14:33:46.416+00:00	2023-08-30	2023	08	131	Jaclyn Euridice	jaclyn.euridice@yopmail.com	944	Elena
16	464	2023-12-23T14:33:46.408+00:00	2023-04-29	2023	04	133	Mariele Emanuel	mariele.emanuel@yopmail.com	505	Jobi
17	464	2023-12-23T14:33:46.408+00:00	2023-04-29	2023	04	133	Mariele Emanuel	mariele.emanuel@yopmail.com	505	Jobi

2 days ago (1s) 16

```
%sql
create database if not exists healthcare_analysis;
```

OK

2 days ago (6s) 17 Python

```
final_summary_df.write.mode("overwrite").format("delta").saveAsTable(
    "healthcare_analysis.patient_appointment_summary"
)
```

(12) Spark Jobs

2 days ago (2s) 18

```
final_summary_df.write.mode("overwrite").parquet(
    f"abfss://destination@projectadf.dfs.core.windows.net/patient_appointment_summary"
)
```

(7) Spark Jobs

Transformation/specialization_summary

Read required processed data

2 days ago (<1s) 3

```
appointments_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/appointments")
doctors_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/doctors")
billing_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/billing")
```

(3) Spark Jobs

appointments_df: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 7 more fields]
billing_df: pyspark.sql.dataframe.DataFrame = [invoice_id: integer, patient_id: integer ... 2 more fields]
doctors_df: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, doctor_name: string ... 3 more fields]

Transformation 1: Number of doctors in each specialization

2 days ago (1s) 5 Python

```
# Transformation 1: Number of doctors in each specialization
from pyspark.sql.functions import count
doctors_per_specialization = doctors_df.groupBy("specialization").agg(
count("doctor_id").alias("number_of_doctors")
)
print("Number of doctors per specialization:")
doctors_per_specialization.show()
```

(2) Spark Jobs

doctors_per_specialization: pyspark.sql.dataframe.DataFrame = [specialization: string, number_of_doctors: long]

```
|      geriatrician|                20|
|  gastroenterology|                33|
```

Transformation 2: Revenue per year by doctor and specialization

2 days ago (1s) 7 Python

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

# Join billing with appointments using patient_id to get doctor_id and year, then with doctors for specialization
revenue_by_doctor_year = billing_df.join(
    appointments_df.select("patient_id", "doctor_id", "appointment_year"),
    "patient_id",
    "inner"
).join(
    doctors_df.select("doctor_id", "doctor_name", "specialization"),
    "doctor_id",
    "inner"
).groupBy("doctor_id", "doctor_name", "specialization", "appointment_year").agg(
    sum("amount").alias("yearly_revenue")
)

print("Revenue by doctor and year:")
revenue_by_doctor_year.orderBy("appointment_year", "yearly_revenue", ascending=False).show()
```

(4) Spark Jobs

revenue_by_doctor_year: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, doctor_name: string ... 3 more fields]

```
Revenue by doctor and year:
+---------+-----------+-------------------+----------------+--------------+
|doctor_id|doctor_name|     specialization|appointment_year|yearly_revenue|
+---------+-----------+-------------------+----------------+--------------+
|      667|     Gloria|       geriatrician|            2023|     4035149.0|
|      801|     Brooks|        dermatology|            2023|     3436703.0|
|      452|      Minne|critical care med...|            2023|     3436703.0|
|      189|        Max|    family medicine|            2023|     3283643.0|
```

Transformation 3: Total revenue for each doctor by year

2 days ago (1s) 9

```
total_revenue_by_doctor_year = revenue_by_doctor_year.groupBy("doctor_id", "doctor_name", "specialization", "appointment_year").agg(
    sum("yearly_revenue").alias("total_yearly_revenue")
)

print("Total revenue per doctor by year:")
total_revenue_by_doctor_year.orderBy("appointment_year", "total_yearly_revenue", ascending=False).show()
```

(4) Spark Jobs

total_revenue_by_doctor_year: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, doctor_name: string ... 3 more fields]

```
Total revenue per doctor by year:
+---------+-----------+-------------------+----------------+--------------------+
|doctor_id|doctor_name|     specialization|appointment_year|total_yearly_revenue|
+---------+-----------+-------------------+----------------+--------------------+
|      667|     Gloria|       geriatrician|            2023|           4035149.0|
|      801|     Brooks|        dermatology|            2023|           3436703.0|
|      452|      Minne|critical care med...|            2023|           3436703.0|
|      189|        Max|    family medicine|            2023|           3283643.0|
```

Transformation 4: Yearly specilization count for each year

2 days ago (1s) 11 Python

```python
from pyspark.sql.functions import col, count

# Ensure year is integer
appointments_clean = appointments_df.withColumn(
    "year_int", col("appointment_year").cast("int")
)

# Join with doctors to bring specialization info
yearly_specialization_appointments = (
    appointments_clean.join(
        doctors_df.select("doctor_id", "specialization"),
        "doctor_id",
        "inner"
    )
    .groupBy("specialization", "year_int")
    .agg(count("appointment_id").alias("yearly_appointment_count"))
    .orderBy("year_int", "yearly_appointment_count", ascending=False)
)

print("Yearly appointment count per specialization:")
yearly_specialization_appointments.show()
```

(3) Spark Jobs

appointments_clean: pyspark.sql.dataframe.DataFrame = [appointment_id: integer, appointment_timestamp: timestamp ... 8 more fields]

yearly_specialization_appointments: pyspark.sql.dataframe.DataFrame = [specialization: string, year_int: integer ... 1 more field]

```
|      ophthalmology|    2023|                       6|
|         nephrology|    2023|                       5|
|         allergists|    2023|                       5|
```

Transformation 5: Yearly revenue trend for each specialization

2 days ago (1s) 16

```python
# Transformation 8: Yearly revenue trend for each specialization
specialization_revenue_trend = revenue_by_doctor_year.groupBy("specialization", "appointment_year").agg(
    sum("yearly_revenue").alias("total_yearly_revenue_specialization"),
    count("doctor_id").alias("number_of_doctors")
).withColumn(
    "avg_revenue_per_doctor", col("total_yearly_revenue_specialization") / col("number_of_doctors")
)

print("Specialization revenue trend:")
specialization_revenue_trend.orderBy("appointment_year", "total_yearly_revenue_specialization", ascending=False).show()
```

(5) Spark Jobs

specialization_revenue_trend: pyspark.sql.dataframe.DataFrame = [specialization: string, appointment_year: string ... 3 more fields]

```
|obstetric anesthe...|            2023|                           5815321.0|                4|          1453830.25|
|        geriatrician|            2023|                           4844910.0|                2|           2422455.0|
|          pediatrics|            2023|                           4644340.0|                3|  1548113.3333333333|
|     family medicine|            2023|                           4568693.0|                2|           2284346.5|
|         dermatology|            2023|                           4316799.0|                3|           1438933.0|
|       ophthalmology|            2023|                           4219545.0|                4|          1054886.25|
|          allergists|            2023|                           4035233.0|                5|            807046.6|
```

Find the most common procedure

2 days ago (<1s)

```
procedures_df = spark.read.format("parquet").load(f"abfss://source@projectadf.dfs.core.windows.net/medic
```

(1) Spark Jobs

procedures_df: pyspark.sql.dataframe.DataFrame = [procedure_id: integer, procedure_name: string ... 2 more fields]

2 days ago (<1s)

```
from pyspark.sql.functions import count

common_procedure = (
    procedures_df
    .groupBy("procedure_name")
    .agg(count("*").alias("procedure_count"))
    .orderBy("procedure_count", ascending=False)
)

common_procedure.show()
```

(2) Spark Jobs

common_procedure: pyspark.sql.dataframe.DataFrame = [procedure_name: string, procedure_count: long]

```
|Comprehensive Ger...|             24|
|Interventional Ra...|             23|
|Laser Therapy For...|             22|
|Sedation For Mino...|             22|
|     General Surgery|             22|
|Dermatologic Surgery|             22|
```

Rank by specialization

2 days ago (1s) 24

```python
from pyspark.sql.window import Window
from pyspark.sql.functions import rank, col, sum

# Base: doctors with career revenue (before joining everything else)
doctor_revenue_base = total_revenue_by_doctor_year.groupBy("doctor_id").agg(
    sum("total_yearly_revenue").alias("total_revenue")
).join(
    doctors_df.select("doctor_id", "doctor_name", "specialization"),
    "doctor_id",
    "inner"
)

# Define ranking window by specialization
specialization_rank_window = Window.partitionBy("specialization").orderBy(col("total_revenue").desc())

# Add rank column
doctor_performance_df = doctor_revenue_base.withColumn(
    "specialization_rank",
    rank().over(specialization_rank_window)
)

# Preview top 20
doctor_performance_df.limit(20).display()
```

(7) Spark Jobs

doctor_performance_df: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, total_revenue: double ... 3 more fields]

doctor_revenue_base: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, total_revenue: double ... 2 more fields]

Table +

	doctor_id	total_revenue	doctor_name	specialization	specialization_rank
1	975	2173939	Hettie	allergists	1
2	619	1987601	Shannah	allergists	2
3	430	1691230	Eolanda	allergists	3
4	742	908837	Gui	allergists	4

2 days ago (<1s) 25

```python
from pyspark.sql.functions import coalesce, lit, sum, current_timestamp

# Start with doctor-level base info
final_summary_df = doctors_df.select(
    "doctor_id",
    "doctor_name",
    "specialization",
    "contact_info"
)

# Skip doctor_total_appointments join since you don't need it

# Join with revenue by year (career total revenue)
final_summary_df = final_summary_df.join(
    total_revenue_by_doctor_year.groupBy("doctor_id")
    .agg(sum("total_yearly_revenue").alias("total_revenue")),
    "doctor_id",
    "left"
).withColumn("total_revenue", coalesce(col("total_revenue"), lit(0.0)))

# Join with specialization demand classification
final_summary_df = final_summary_df.join(
    specialization_demand_yearly.select(
        "specialization",
        "specialization_demand_level"
    ),
    "specialization",
    "left"
).withColumn("specialization_demand_level", coalesce(col("specialization_demand_level"), lit("Unknown")))

# Join with specialization workload classification
final_summary_df = final_summary_df.join(
    specialization_workload_classified.select(
        "specialization",
        col("total_appointments").alias("specialization_total_appointments"),
        "workload_level"
    ),
    "specialization",
    "left"
).withColumn("specialization_total_appointments", coalesce(col("specialization_total_appointments"), lit(0))) \
 .withColumn("workload_level", coalesce(col("workload_level"), lit("Unknown")))

# Add lineage timestamp
final_summary_df = final_summary_df.withColumn("ingestion_date", current_timestamp())
```

2 days ago (1s) 26

```sql
%sql
drop table if exists healthcare_analysis.specialization_summary
```

OK

2 days ago (7s) 27

```python
final_summary_df.write.mode("overwrite").format("delta").saveAsTable("healthcare_analysis.specialization_summary")
```

(18) Spark Jobs

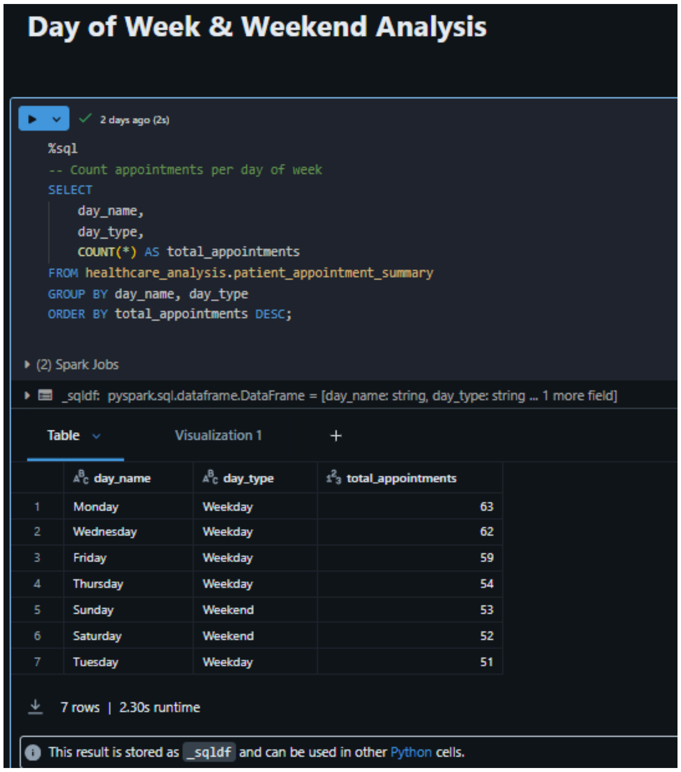
2 days ago (2s) 28

```python
final_summary_df.write.mode("overwrite").parquet(f"abfss://destination@projectadf.dfs.core.windows.net/specialization_summary")
```

(13) Spark Jobs

To visualize the transformed data, create two more notebooks inside the data analysis folder

Data_analysis/Patient_analysis



Day of Week & Weekend Analysis

2 days ago (2s)

```sql
%sql
-- Count appointments per day of week
SELECT
    day_name,
    day_type,
    COUNT(*) AS total_appointments
FROM healthcare_analysis.patient_appointment_summary
GROUP BY day_name, day_type
ORDER BY total_appointments DESC;
```

(2) Spark Jobs

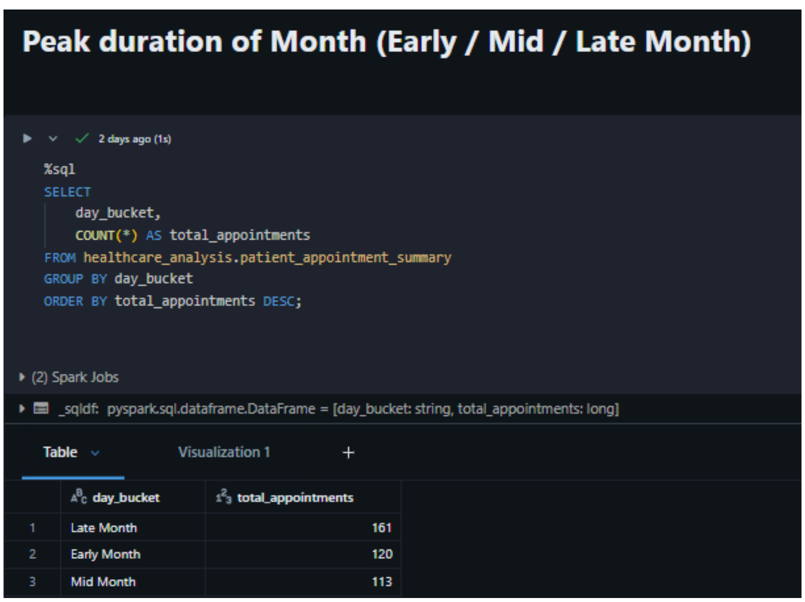
_sqldf: pyspark.sql.dataframe.DataFrame = [day_name: string, day_type: string ... 1 more field]

Table | Visualization 1 | +

	day_name	day_type	total_appointments
1	Monday	Weekday	63
2	Wednesday	Weekday	62
3	Friday	Weekday	59
4	Thursday	Weekday	54
5	Sunday	Weekend	53
6	Saturday	Weekend	52
7	Tuesday	Weekday	51

7 rows | 2.30s runtime

This result is stored as _sqldf and can be used in other Python cells.



Peak duration of Month (Early / Mid / Late Month)

2 days ago (1s)

```sql
%sql
SELECT
    day_bucket,
    COUNT(*) AS total_appointments
FROM healthcare_analysis.patient_appointment_summary
GROUP BY day_bucket
ORDER BY total_appointments DESC;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [day_bucket: string, total_appointments: long]

Table | Visualization 1 | +

	day_bucket	total_appointments
1	Late Month	161
2	Early Month	120
3	Mid Month	113

First vs Returning Patients

2 days ago (1s)

```sql
%sql
SELECT
    is_first_appointment,
    COUNT(*) AS total_appointments,
    COUNT(DISTINCT patient_id) AS unique_patients
FROM healthcare_analysis.patient_appointment_summary
GROUP BY is_first_appointment;
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [is_first_appointment: boolean, total_appointments: long ...

Table | Visualization 1 | +

	is_first_appointment	total_appointments	unique_patients
1	true	219	219
2	false	175	93

Top 10 patients based on their payment details

2 days ago (1s)

8

```sql
%sql
SELECT
    patient_id,
    patient_name,
    SUM(total_amount) AS total_payment
FROM healthcare_analysis.patient_appointment_summary
GROUP BY patient_id, patient_name
ORDER BY total_payment DESC
LIMIT 10;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [patient_id: integer, patient_name: string ... 1 more field]

Table | +

	patient_id	patient_name	total_payment
1	368	Stephanie Agle	27493624
2	779	Candi Shaver	13134572
3	381	Chickie Hazlett	7298356
4	961	Ruthe Garbe	5668260
5	694	Paola Swigart	5529024
6	409	Netty Moseley	5370324
7	339	Mellicent Viddah	4373994
8	546	Aimil Suzetta	4347878
9	994	Gui Turne	4312515
10	817	Trixi Skurnik	4038372

Data_analysis/Patient_analysis

Number of doctors in each specialization

2 days ago (2s)

```sql
%sql
SELECT specialization, COUNT(DISTINCT doctor_id) AS number_of_doctors
FROM healthcare_analysis.specialization_summary
GROUP BY specialization
ORDER BY number_of_doctors DESC;
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [specialization: string, number_of_doctors: long]

Table | Visualization 1 | +

	specialization	number_of_doctors
1	infectious disease	37
2	oncologist	35
3	gastroenterology	33
4	otolaryngologists	32
5	emergency medicine	31
6	dermatology	29
7	allergists	28
8	internists	28
9	pulmonologists	27
10	nephrology	26
11	ophthalmology	25
12	anesthesiology	25
13	neurology	24
14	obstetric anesthesiologis...	24
15	surgery	24

24 rows | 2.30s runtime

Doctors with highest revenue in each specialization

2 days ago (1s)

4

```sql
%sql
SELECT specialization, doctor_id, doctor_name, total_revenue
FROM (
    SELECT
        specialization,
        doctor_id,
        doctor_name,
        total_revenue,
        ROW_NUMBER() OVER (PARTITION BY specialization ORDER BY total_revenue DESC) AS rank
    FROM healthcare_analysis.specialization_summary
) ranked
WHERE rank = 1;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [specialization: string, doctor_id: integer ... 2 more fields]

Table | +

	specialization	doctor_id	doctor_name	total_revenue
1	allergists	975	Hettie	2173939
2	anesthesiology	792	Leanna	3680845
3	cardiology	237	Suzette	1830079
4	critical care medicine	452	Minne	3593294
5	dermatology	801	Brooks	3436703
6	emergency medicine	877	Vevay	2113058
7	endocrinologist	300	Roxane	1889420
8	family medicine	189	Max	3283643
9	gastroenterology	129	Aurelie	3204079
10	geriatrician	667	Gloria	5846564
11	hospice and palliative care	407	Bill	3436703
12	infectious disease	574	Ronna	2206483
13	internists	362	Charissa	1896039
14	nephrology	443	Renie	1354711
15	neurology	976	Netty	3283643

24 rows | 0.79s runtime

2 days ago (1s)

```sql
%sql
-- Highest demand specialization
SELECT specialization, specialization_demand_level
FROM healthcare_analysis.specialization_summary
WHERE specialization_demand_level = 'High Demand'
GROUP BY specialization, specialization_demand_level
ORDER BY COUNT(doctor_id) DESC
LIMIT 1;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [specialization: string, specialization_d

Table

	specialization	specialization_demand_level
1	infectious disease	High Demand

2 days ago (1s)

```sql
%sql
-- Lowest demand specialization
SELECT specialization, specialization_demand_level
FROM healthcare_analysis.specialization_summary
WHERE specialization_demand_level = 'Low Demand'
GROUP BY specialization, specialization_demand_level
ORDER BY COUNT(doctor_id) ASC
LIMIT 1;
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [specialization: string, specialization_den

Table

	specialization	specialization_demand_level
1	hospice and palliative care	Low Demand

Doctor who earns the highest in the hospital

+ Code

2 days ago (<1s)

```sql
%sql
SELECT doctor_id, doctor_name, specialization, total_revenue
FROM healthcare_analysis.specialization_summary
ORDER BY total_revenue DESC
LIMIT 1;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [doctor_id: integer, doctor_name: string ... 2 more fields]

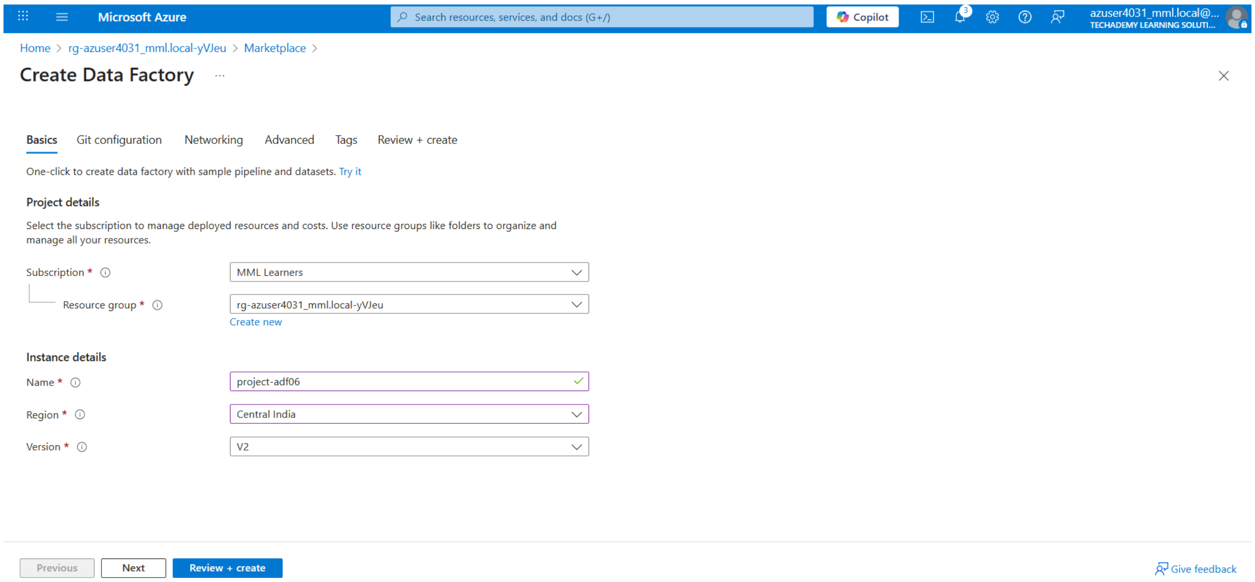
Table

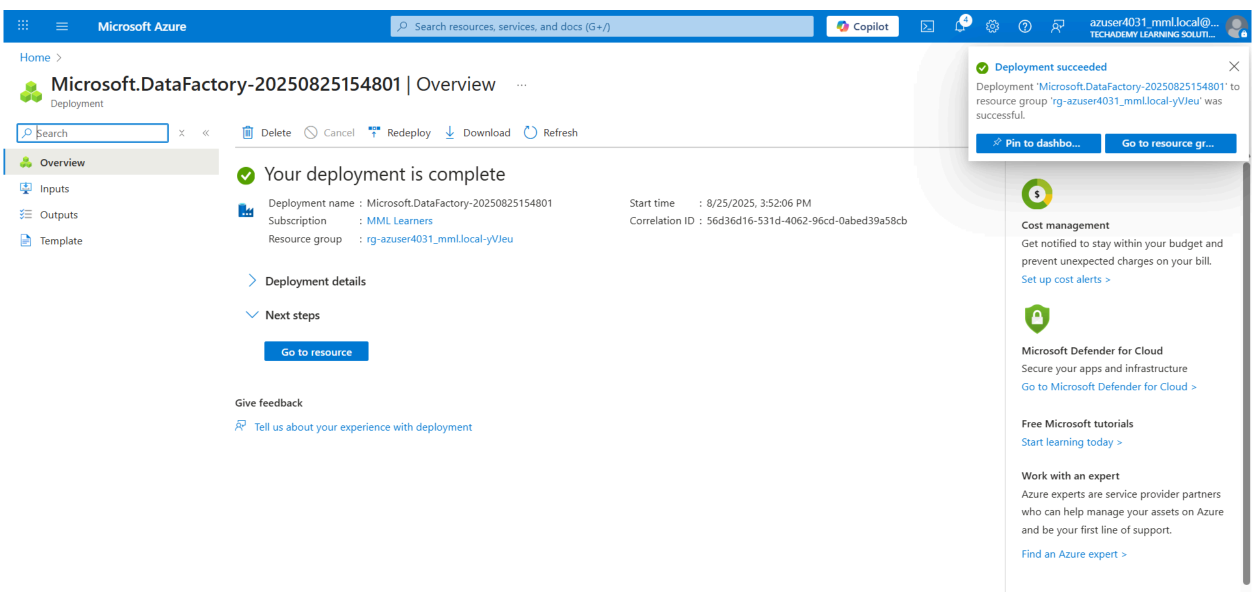
	doctor_id	doctor_name	specialization	total_revenue
1	667	Gloria	geriatrician	5846564

Here is the link of notebook files we have used in this project

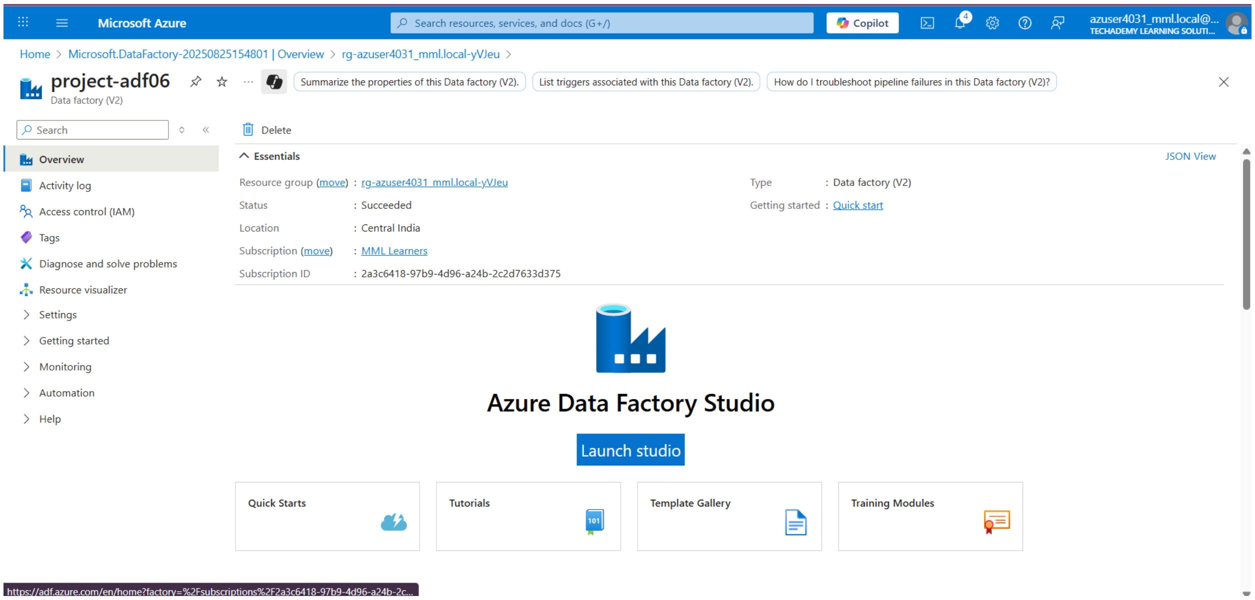
Step-3 : Create a Data Factory

Create a new Azure Data Factory to implement a serverless data processing pipeline.

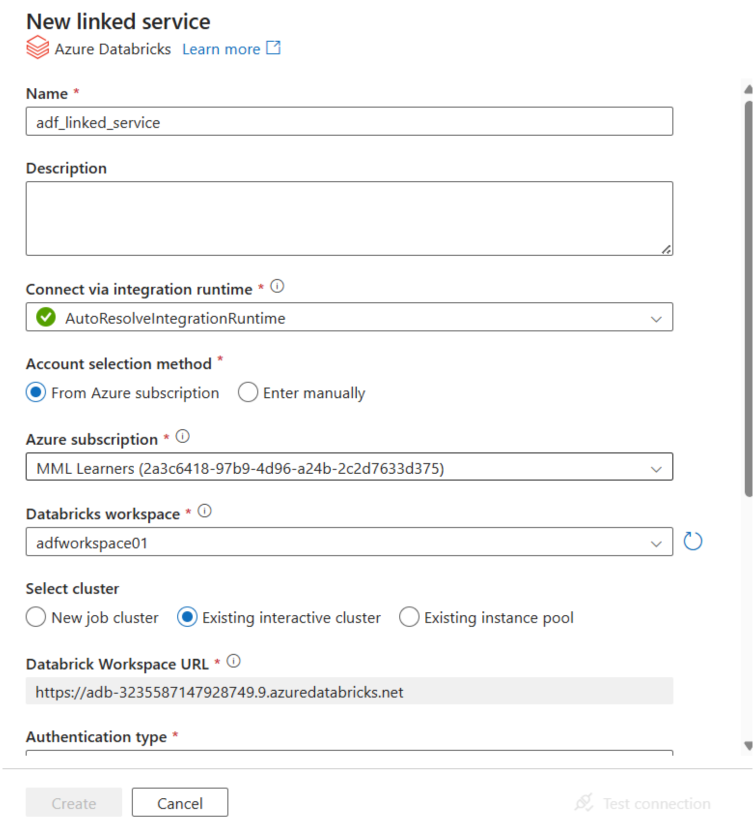


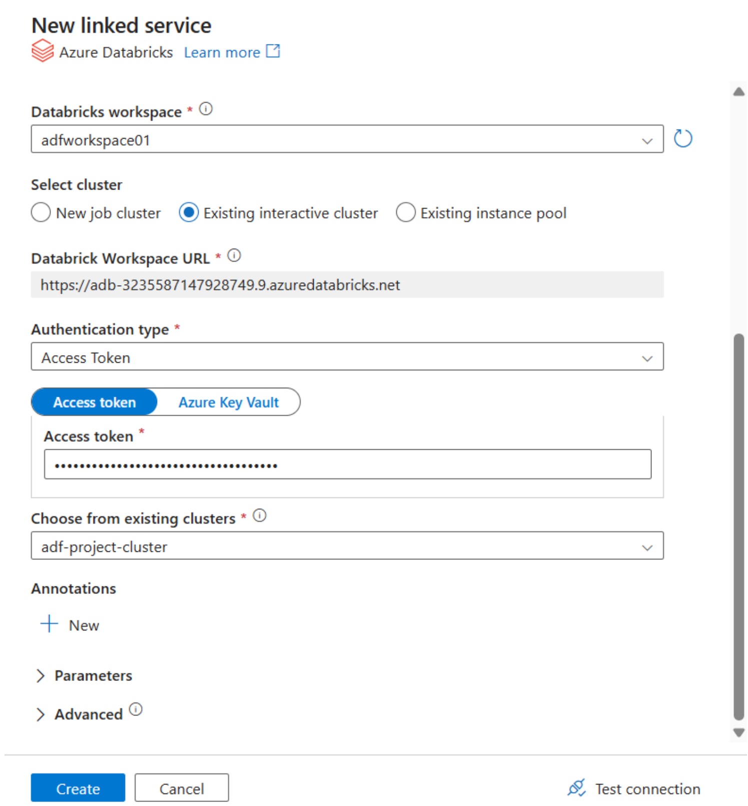


Next, launch the Azure Data Factory Studio that is created.



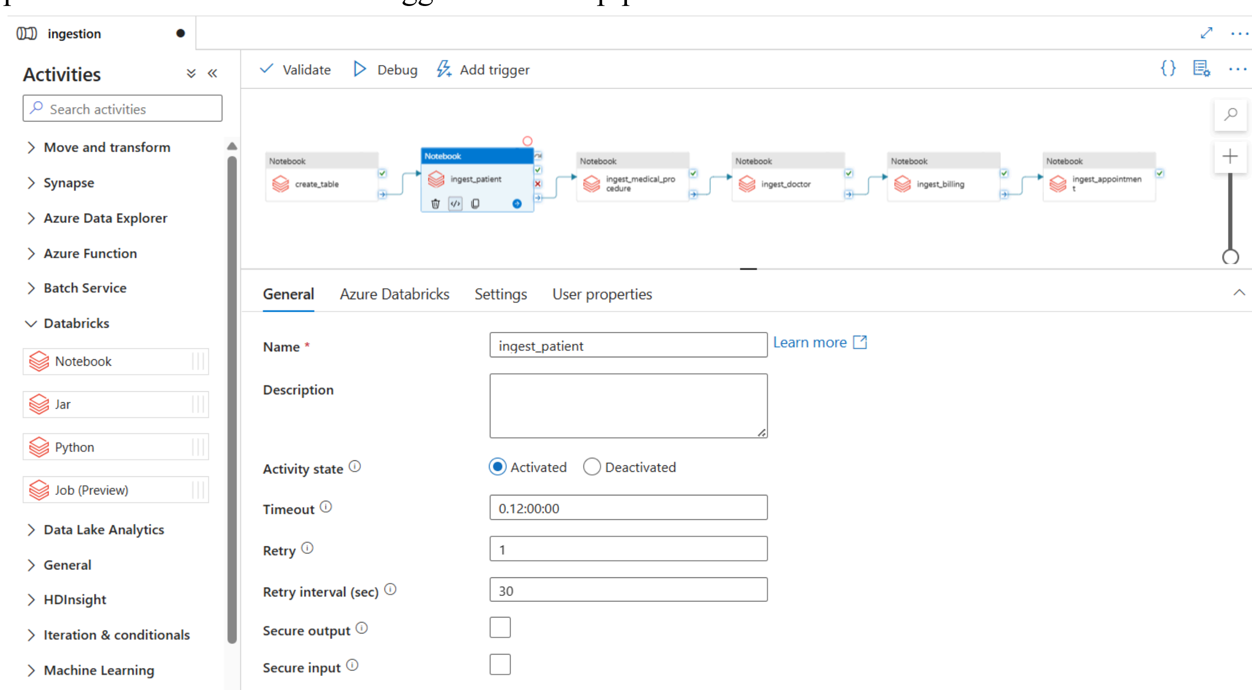
Define a linked service for the Azure Storage account, Below is the process of creating a linked service.

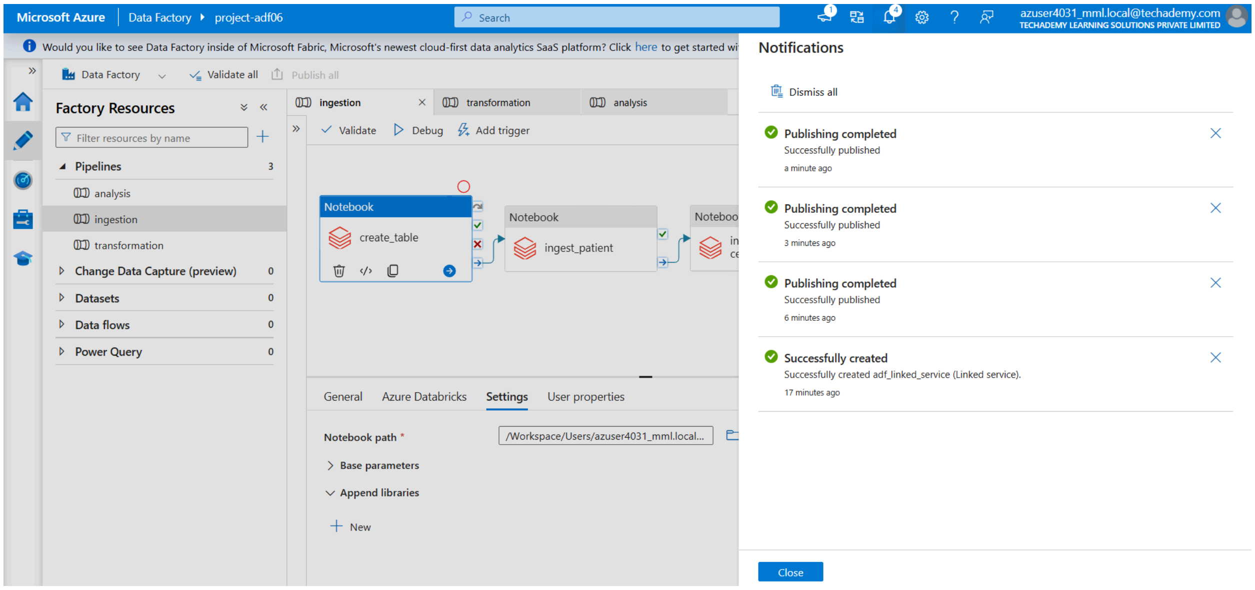




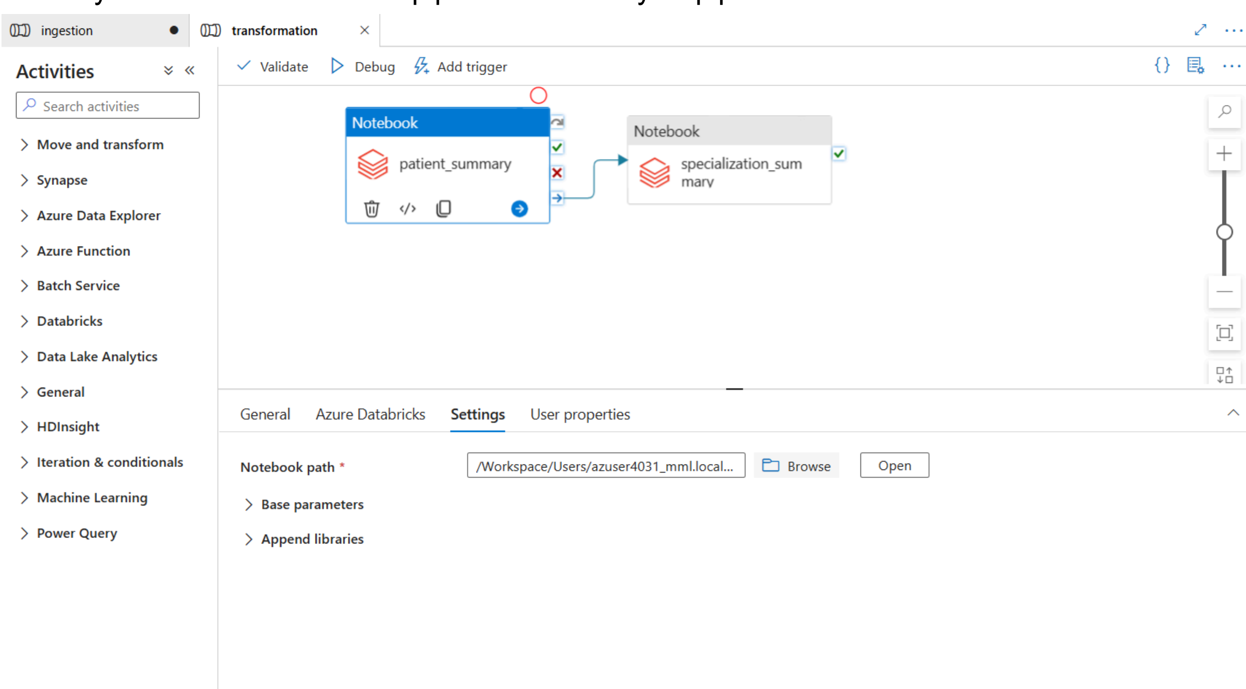
Creating Ingestion pipeline

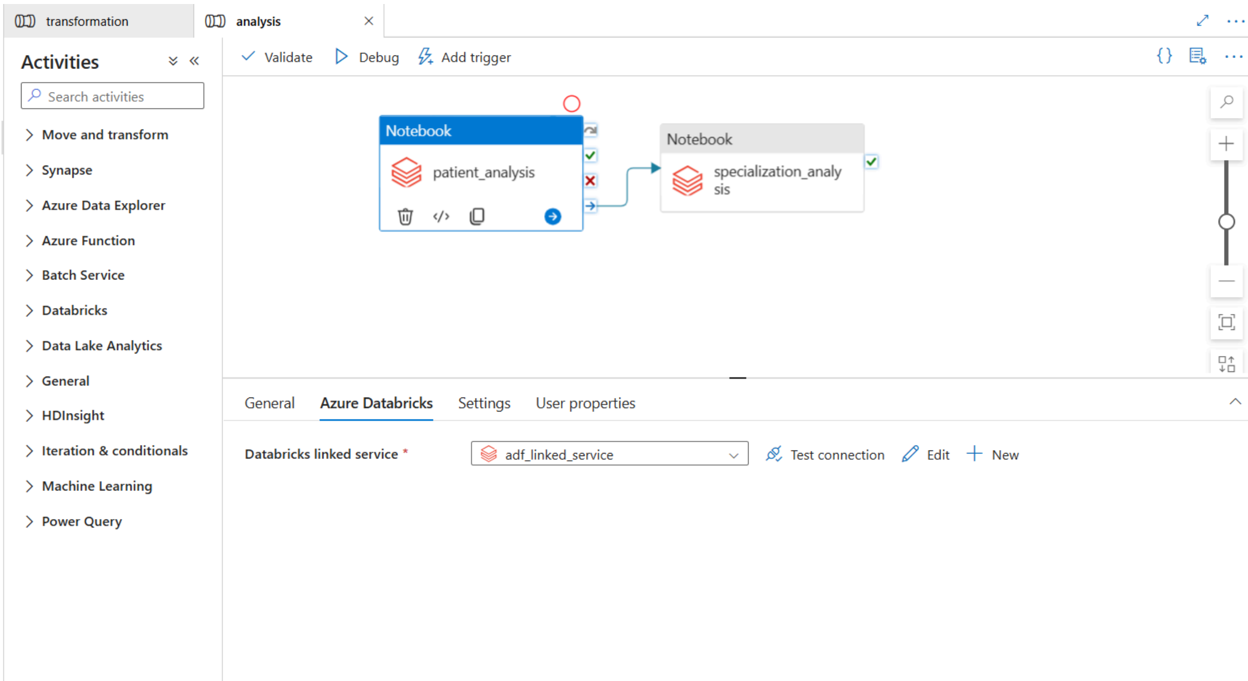
In the Activity tab, select databricks notebook activity and now add 6 notebook activity to create the pipeline and configure the activity for each notebook like below. Once configure click publish all then click on add trigger to run the pipeline



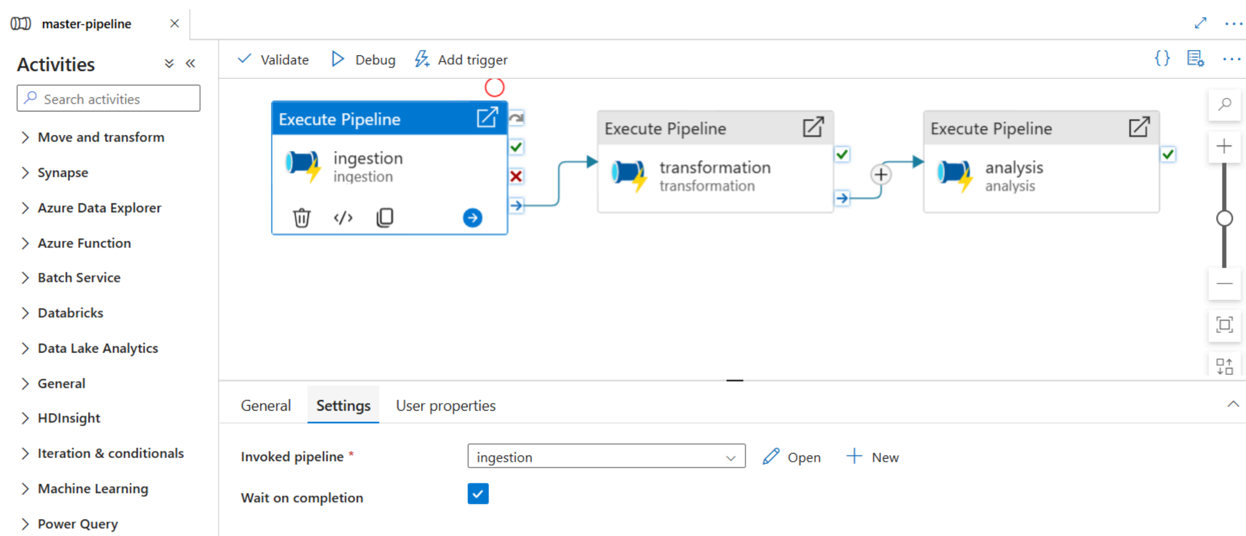


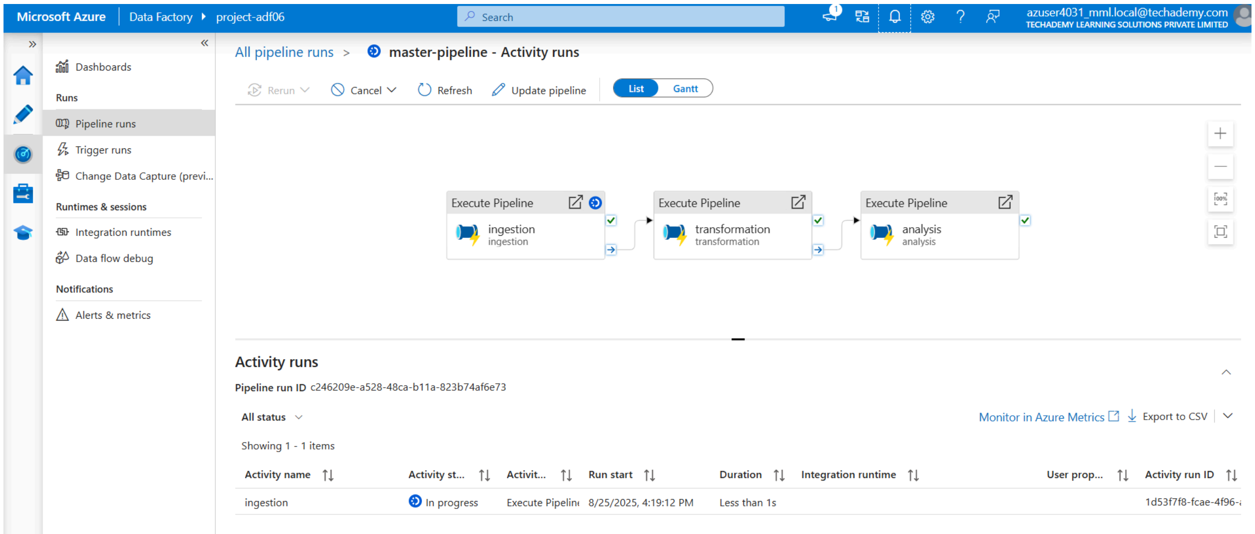
Similarly for the Transformation pipeline and Analysis pipeline we do the same.





Create Master Pipeline by selecting execute pipeline in the general activity section and map the ingestion, transformation and analysis pipeline one after the other to create the complete master pipeline.

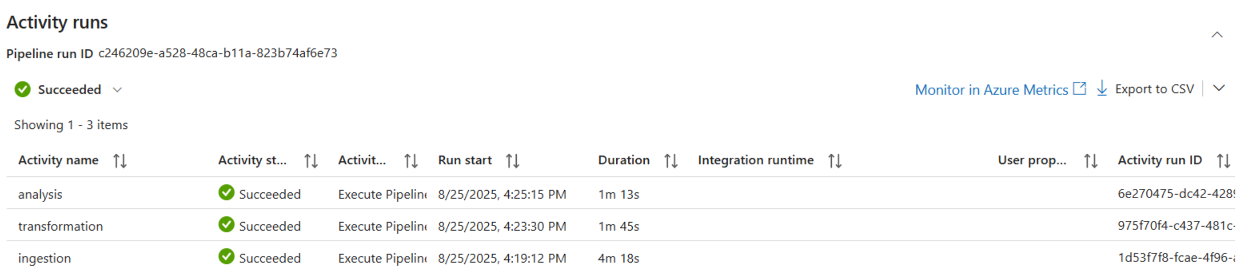




Output

Successful Output Generated:

- Validate and Debug the pipeline created to see the results of the execution.
- It shows the Activity Status as **Succeeded** which means our pipeline successfully ingested, transformed and analyzed the data.



Activity runs

Pipeline run ID c246209e-a528-48ca-b11a-823b74af6e73

Succeeded

Showing 1 - 3 items

Activity name	Activity st...	Activit...	Run start	Duration	Integration runtime	User prop...	Activity run ID
analysis	Succeeded	Execute Pipeline	8/25/2025, 4:25:15 PM	1m 13s			6e270475-dc42-428!
transformation	Succeeded	Execute Pipeline	8/25/2025, 4:23:30 PM	1m 45s			975f70f4-c437-481c-
ingestion	Succeeded	Execute Pipeline	8/25/2025, 4:19:12 PM	4m 18s			1d53f7f8-fcae-4f96-

In the source container we can see the processed data is stored source

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

Search blobs by prefix (case-sensitive)

Showing all 5 items

Name	Last modified	Access tier
appointments	8/25/2025, 2:58:11 PM	
billing	8/25/2025, 2:57:24 PM	
doctors	8/25/2025, 2:55:28 PM	
medical_procedures	8/25/2025, 2:54:35 PM	
patients	8/25/2025, 2:53:33 PM	

source > appointments

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

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

Showing all 8 items

Name	Last modified	Access tier	Blob type	Size	Lease state
[..]					
_SUCCESS	8/25/2025, 4:23:15 PM	Hot (Inferred)	Block blob	0	Available
_committed_5570965616819959164	8/25/2025, 4:23:10 PM	Hot (Inferred)	Block blob	233 B	Available
_committed_6198646254894929306	8/25/2025, 2:58:11 PM	Hot (Inferred)	Block blob	123 B	Available
_committed_9115898469978707106	8/25/2025, 4:23:15 PM	Hot (Inferred)	Block blob	223 B	Available
_committed_vacuum5847446971087808802	8/25/2025, 4:23:10 PM	Hot (Inferred)	Block blob	96 B	Available
_started_5570965616819959164	8/25/2025, 4:23:10 PM	Hot (Inferred)	Block blob	0	Available
_started_9115898469978707106	8/25/2025, 4:23:15 PM	Hot (Inferred)	Block blob	0	Available
part-00000-tid-9115898469978707106-fb2945ab-b10a-49e5-...	8/25/2025, 4:23:15 PM	Hot (Inferred)	Block blob	12.94 KiB	Available

Similarly in the destination container we can see our transformed data is stored for the further analysis

Add Directory | Upload | Refresh | Delete | Copy | Paste | Rename | Acquire lease | Break lease | Edit columns

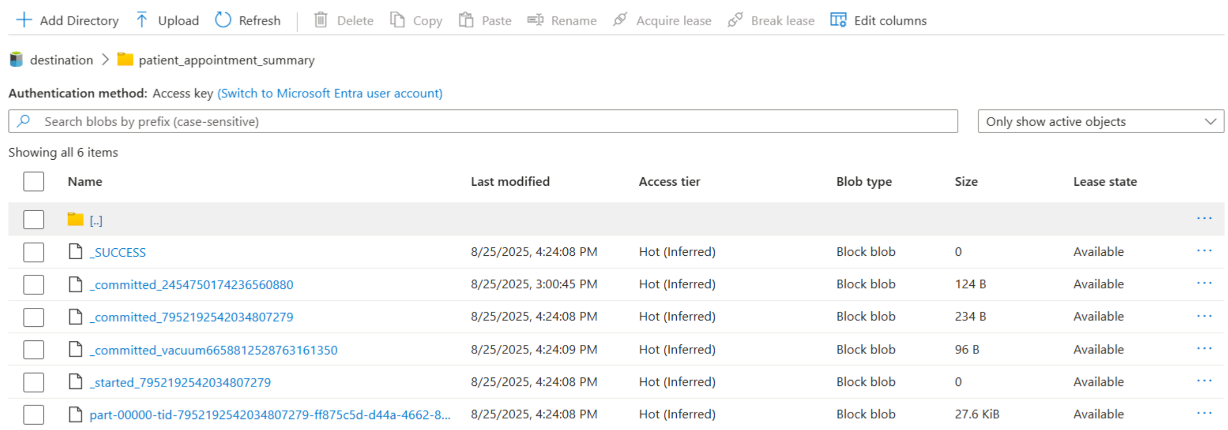
destination

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

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

Showing all 2 items

Name	Last modified	Access tier	Blob type	Size	Lease state
patient_appointment_summary	8/25/2025, 3:00:45 PM				
specialization_summary	8/25/2025, 3:06:27 PM				



Strategies for Implementing a Serverless Data Processing Pipeline with Azure Data Factory and Azure Databricks

1. Use Serverless and On-Demand Compute

Leverage serverless Databricks SQL warehouses or auto-terminating clusters to avoid idle costs. Trigger pipeline execution only when needed using ADF triggers.

2. Decouple Storage and Compute

Store raw, intermediate, and processed data in Azure Data Lake Gen2 or Blob Storage. Use Delta Lake for schema enforcement, transactional consistency, and performance.

3. Modular Pipeline Design

Design pipelines as independent stages for ingestion, validation, transformation, and loading. Use parameters and variables in ADF to make them environment-agnostic.

4. Databricks for Heavy Transformations

Utilize Spark for large-scale ETL, streaming, machine learning, and graph processing. Use Delta Live Tables (DLT) for reliable declarative ETL.

5. Data Layout Optimization

Apply partitioning, Z-ordering, compression, and caching in Delta Lake to enhance query speed and storage efficiency.

6. Security and Governance

Use Managed Identity for secure access to storage. Enable Unity Catalog for centralized data governance, access control, and lineage tracking.

7. Monitoring and Logging

Monitor pipelines through Azure Monitor and Log Analytics. Track Spark UI logs, job performance, and cluster utilization, with alerts for errors or performance drops.

8. Cost Management

Enable auto-termination for clusters, right-size based on workload, and use spot instances where possible. Track costs with Azure Cost Management and budgets.

9. CI/CD and DevOps Integration
Integrate ADF pipelines and Databricks notebooks with GitHub or Azure DevOps. Use Databricks Repos for version control and CI/CD pipelines for deployment automation.

10. Error Handling and Retry Mechanisms
Implement retry policies in ADF, configure dead-letter queues for bad records, and log errors for monitoring and alerting.

Conclusion

This project successfully implemented a serverless data processing pipeline using Azure Data Factory (ADF) and Azure Databricks, enabling seamless data ingestion, transformation, and analysis. The integration of these Azure services provided a scalable, cost-efficient, and modular solution for handling large-scale data workflows.

Successful Orchestration of Pipelines

Azure Data Factory served as the central orchestrator, coordinating three separate pipelines for ingestion, transformation, and analysis. By combining these pipelines into a master pipeline, the project ensured a clear workflow structure and efficient execution with dependency management.

Efficient Data Processing with Databricks

Azure Databricks provided the computational power for complex transformations and analytical tasks. Its distributed Spark engine enabled large-scale data processing, while Delta Lake ensured transactional consistency, schema enforcement, and optimized query performance.

Monitoring and Control

Although the pipeline was executed through manual triggers, monitoring and logging were effectively handled via ADF run history and Databricks job metrics. This allowed for effective tracking, debugging, and performance evaluation.