

# PROJECT REPORT ON

## MEDICAL CLAIM DATASET

Prepared by:

Associate Name: Chandu Ragiri

GUIDED BY :

Trainer

Guru Aravinda Senapathy

Sr. Manager- Projects

AIA MSFT Data

Mentor

Kavin Kumar Govindaraj

Sr. Associate

AIA Cloud Data Integration

PROJECT DEMONSTRATION:

[WATCH IT](#)

# Index

Content	Page Numbers
Introduction	3
System Software	3
Functional Requirement 1 (Splitting Dataset Into 6 subsets)	4
Functional Requirement 2 (Log In to Azure and setting ADF)	6
Functional Requirement 3 (Converting .csv file into .parquet file)	10
Functional Requirement 4 (Splitting the data into Fact and Dimension Tables)	13
Functional Requirement 5 (Archiving and Purging the old files )	16
Conclusion	20

## INTRODUCTION :

The **Incremental Load Project on Medical Claim Datasets** was implemented using **Azure Data Factory (ADF)** and **Azure Data Lake Storage (ADLS)** to efficiently handle ETL processes. ADF facilitated **incremental data ingestion, transformation, and storage** while ensuring seamless integration with fact and dimension tables. The project involved three structured pipelines: **CSV to Parquet conversion, Parquet to fact and dimension tables transformation, and archival management**. By automating claim data processing, the system optimized **data accessibility, integrity, and performance**, enhancing analytical capabilities in healthcare claim management.

## SYSTEM SOFTWARE

The "Medical Claim Dataset" project successfully leveraged the robust capabilities of Azure Cloud to meet critical business requirements, specifically focusing on efficient and scalable data integration and processing. Azure, a leading cloud computing platform, proved to be an optimal choice due to its comprehensive suite of services for data storage, transformation, and analytics. This project capitalized on Azure's power to manage the complexities inherent in large-scale medical datasets, ensuring that all data operations were handled with high performance and reliability, aligning perfectly with the demands of modern healthcare data management.

A core component of this initiative was the development of the "Medical Claim Dataset" system itself, designed to significantly streamline healthcare data processing. This system achieves its efficiency by meticulously organizing patient records, claim transactions, and medical procedures. Azure Data Factory played a pivotal role in this architecture, orchestrating the ingestion and transformation of vast quantities of medical claim data. Its advanced features enabled crucial functionalities such as incremental data loads, which optimize processing time and

resource utilization, as well as sophisticated surrogate key generation for data integrity and lifecycle management, all contributing to the system's overall reliability and scalability.

Beyond data processing, the project also prioritized secure and accessible data storage and advanced analytics. Azure Blob Storage was strategically employed to provide a secure and scalable repository for both raw and processed medical claim data, ensuring data integrity and availability. Furthermore, Azure Synapse Analytics was integrated to facilitate advanced querying and reporting on the processed patient claims. This powerful combination empowers healthcare providers and insurers with the ability to derive actionable insights from their data, ultimately leading to improved operational efficiency, better patient outcomes, and more informed decision-making within the healthcare ecosystem.

## **FUNCTIONAL REQUIREMENT 1**

### **Dividing Main dataset into 6 subsets**

I started by using Google Colab as my development environment because of its ease of use with big datasets and cloud-based execution. I made sure to use pip to install the necessary Python packages, such as kagglehub to download the dataset straight from Kaggle and pandas for data manipulation.

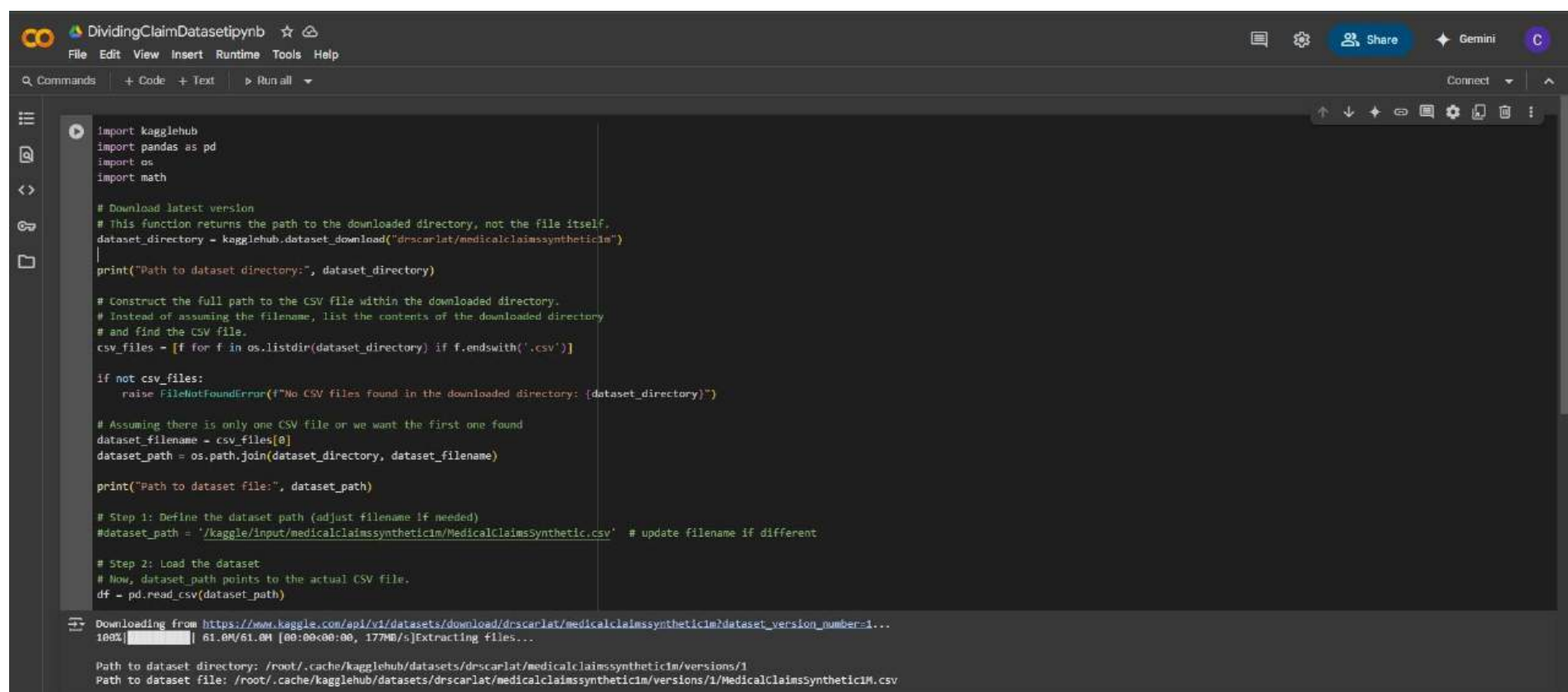
I obtained access to the dataset directory containing the.csv file by using kagglehub.dataset\_download to download the Medical Claims Synthetic Dataset (1M rows). Instead of hardcoding the file name, I dynamically fetched the CSV file path and listed the directory contents to guarantee reproducibility and automate file detection. I created a distinct MD5 hash for every row in the dataset before splitting it up. All of a row's column values were combined into a string to create this hash, and Python's hashlib library was then used to calculate its MD5 digest. A new column called md5\_hash was

created and used to store the hash values produced. In subsequent processes, this step facilitated the tracking and preservation of row uniqueness.

After the hashing process was finished, I divided the total number of rows (1,000,000) by 6 to determine how many rows each subset had, which came out to be 166,666 rows. After that, I created a loop that divided the DataFrame into six equal sections.

Each subset was saved as a separate CSV file, named sequentially as subset\_1.csv through subset\_6.csv. For convenience during additional processing, these files were kept in the working directory. In addition to making the data manageable, this methodical approach got it ready for batch-wise ingestion into the Azure Data Factory pipeline.

Script :



```
import kagglehub
import pandas as pd
import os
import math

# Download latest version
# This function returns the path to the downloaded directory, not the file itself.
dataset_directory = kagglehub.dataset_download("drscarlat/medicalclaimssynthetic1m")
print("Path to dataset directory:", dataset_directory)

# Construct the full path to the CSV file within the downloaded directory.
# Instead of assuming the filename, list the contents of the downloaded directory
# and find the CSV file.
csv_files = [f for f in os.listdir(dataset_directory) if f.endswith('.csv')]

if not csv_files:
    raise FileNotFoundError("No CSV files found in the downloaded directory: {dataset_directory}")

# Assuming there is only one CSV file or we want the first one found
dataset_filename = csv_files[0]
dataset_path = os.path.join(dataset_directory, dataset_filename)

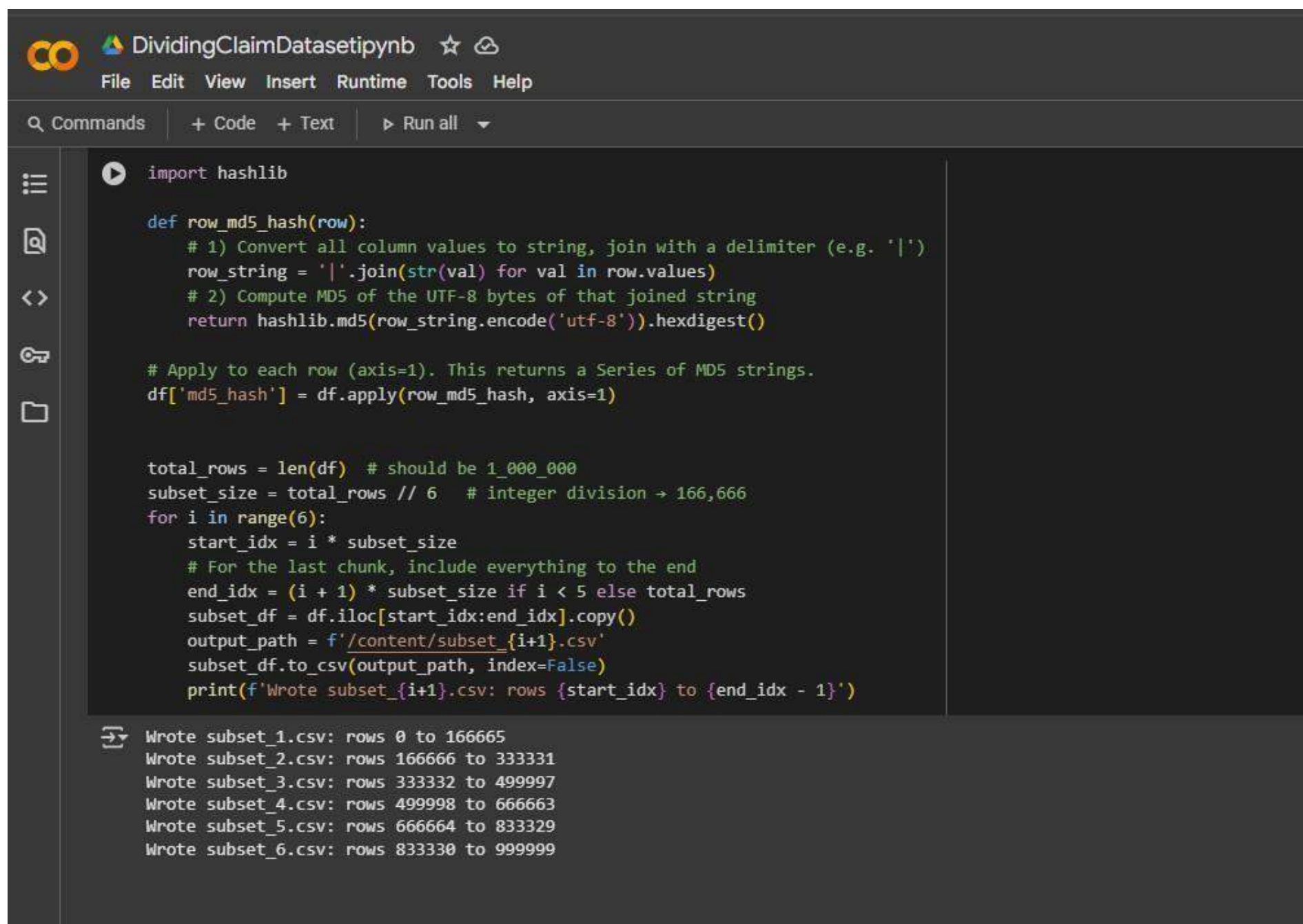
print("Path to dataset file:", dataset_path)

# Step 1: Define the dataset path (adjust filename if needed)
#dataset_path = '/kaggle/input/medicalclaimssynthetic1m/MedicalClaimsSynthetic.csv' # update filename if different

# Step 2: Load the dataset
# Now, dataset_path points to the actual CSV file.
df = pd.read_csv(dataset_path)
```

Downloading from [https://www.kaggle.com/api/v1/datasets/download/drscarlat/medicalclaimssynthetic1m?dataset\\_version\\_number=1...](https://www.kaggle.com/api/v1/datasets/download/drscarlat/medicalclaimssynthetic1m?dataset_version_number=1...)  
100%|██████████| 61.0M/61.0M [00:00<00:00, 177MB/s]Extracting files...

Path to dataset directory: /root/.cache/kagglehub/datasets/drscarlat/medicalclaimssynthetic1m/versions/1  
Path to dataset file: /root/.cache/kagglehub/datasets/drscarlat/medicalclaimssynthetic1m/versions/1/MedicalClaimsSynthetic1M.csv



```
import hashlib

def row_md5_hash(row):
    # 1) Convert all column values to string, join with a delimiter (e.g. '|')
    row_string = '|'.join(str(val) for val in row.values)
    # 2) Compute MD5 of the UTF-8 bytes of that joined string
    return hashlib.md5(row_string.encode('utf-8')).hexdigest()

# Apply to each row (axis=1). This returns a Series of MD5 strings.
df['md5_hash'] = df.apply(row_md5_hash, axis=1)

total_rows = len(df) # should be 1_000_000
subset_size = total_rows // 6 # integer division -> 166,666
for i in range(6):
    start_idx = i * subset_size
    # For the last chunk, include everything to the end
    end_idx = (i + 1) * subset_size if i < 5 else total_rows
    subset_df = df.iloc[start_idx:end_idx].copy()
    output_path = f'/content/subset_{i+1}.csv'
    subset_df.to_csv(output_path, index=False)
    print(f'Wrote subset_{i+1}.csv: rows {start_idx} to {end_idx - 1}')

Wrote subset_1.csv: rows 0 to 166665
Wrote subset_2.csv: rows 166666 to 333331
Wrote subset_3.csv: rows 333332 to 499997
Wrote subset_4.csv: rows 499998 to 666663
Wrote subset_5.csv: rows 666664 to 833329
Wrote subset_6.csv: rows 833330 to 999999
```

## FUNCTIONAL REQUIREMENT 2

### Creating the azure service

Azure Portal using Azure ID, gaining access to a centralized platform for managing cloud resources. The portal provides an intuitive interface that simplifies the deployment, configuration, and monitoring of various services within Microsoft's cloud infrastructure. It serves as a control panel, allowing users to efficiently organize and maintain their cloud-based operations. As part of the initial setup, I created a Resource Group to organize all project-related services. Resource groups play a crucial role in structuring cloud environments, ensuring that resources remain logically grouped for better management and accessibility. With the resource group established, I proceeded to create an Azure Storage



Account, which serves as a secure and scalable storage solution.

This account provides the flexibility to store various types of data, including Blob Storage, File Storage, Queue Storage, and Table Storage, making it a vital component for cloud-based data management. And created the azure data factory which we can do all the data transformation and data copy and control flow activities.

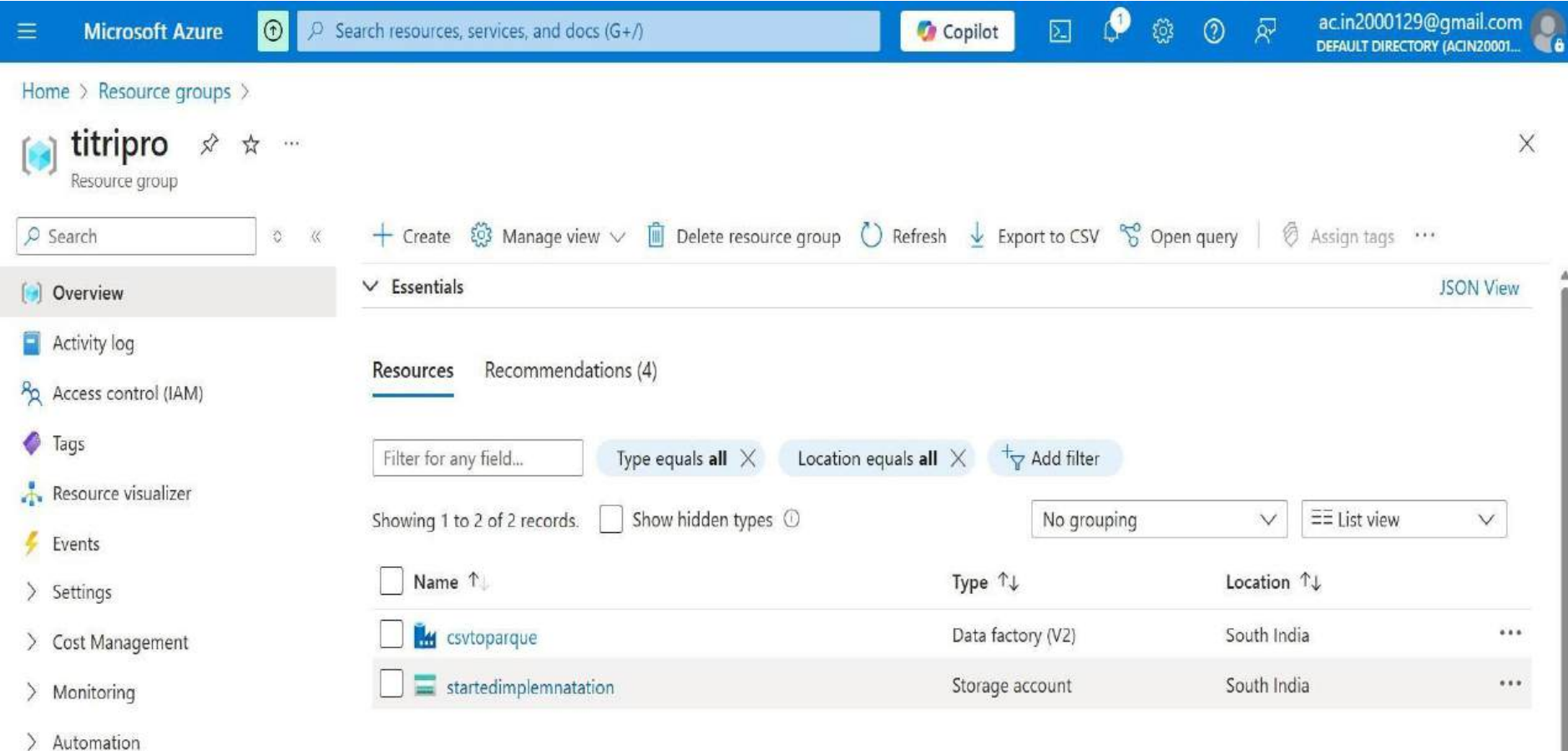


Fig 1: Resource Group Page of Azure Portal

In the **Azure Storage resource**, I set up a **container** to organize and manage stored data efficiently. Within this container, I created multiple folders to categorize different stages of data processing. The folders were named **Stage, Preprocess, and Sink**, each serving a distinct function in the data workflow.

The **Stage** folder acted as the initial storage location for incoming data before undergoing any processing. The **Preprocess** folder was used to store data that required transformation, cleaning, or enrichment before moving forward in the pipeline. The **Sink** folder served as the final destination for processed data, ready for further analysis or integration.

To enhance organization, I also created **archive folders** within these main directories to maintain historical data versions and backup important files. This structure ensures efficient data management and seamless processing within the Azure environment.

**Folder Structure:**

- stg/           # Raw CSV subsets
- preprocess/   # Parquet files
- sink/           # Final fact/dimension tables
- stg/archive/# For archived data
- preprocess/archive/       # For purged data
- sink/archive/       # For archived data



Fig 2: Folder Structure of ADLS in Storage Account

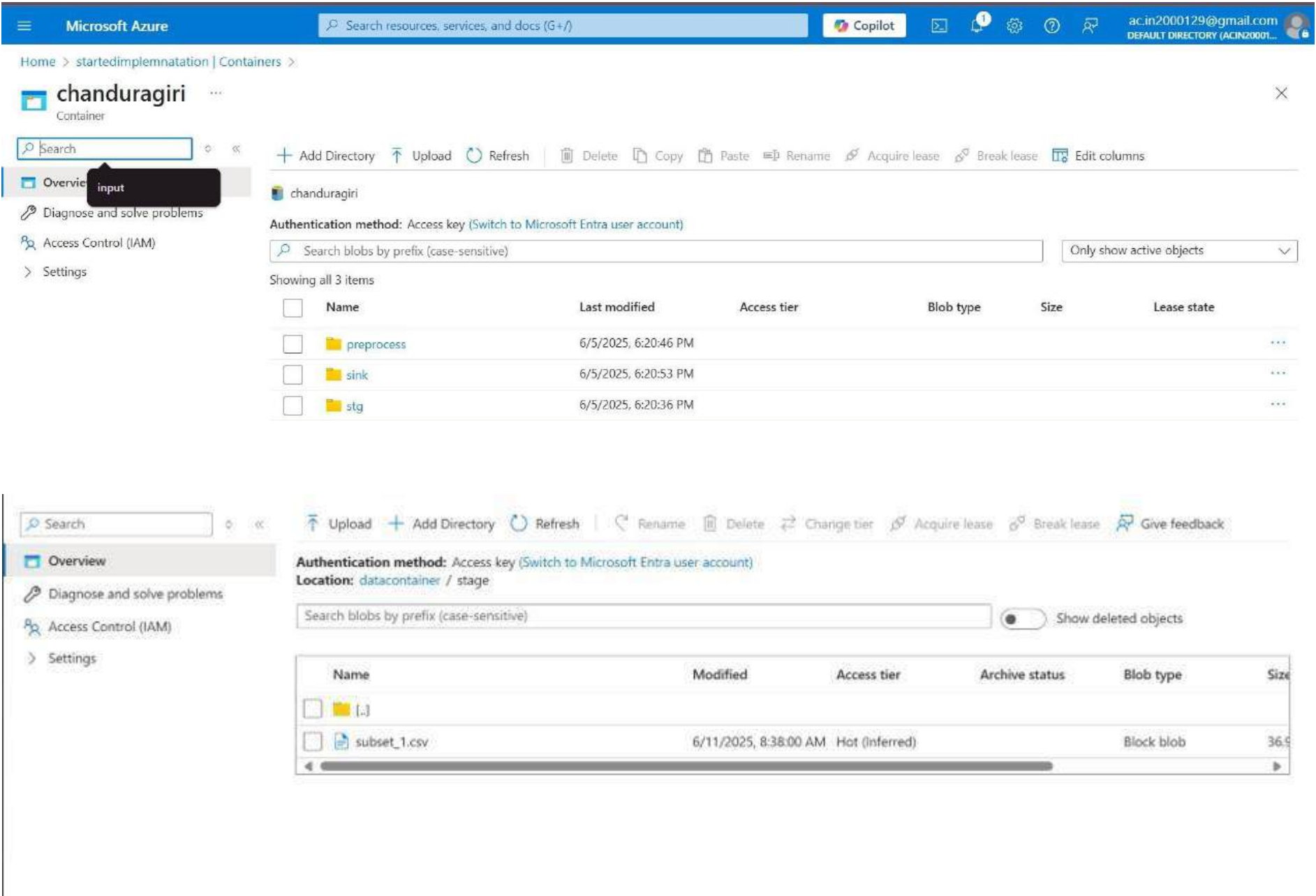


Fig 3: Data is uploaded to storage folder

Upload the 6 subsets, ie sample one subset uploaded of .csv file in the storage folder. After that created an ADF.

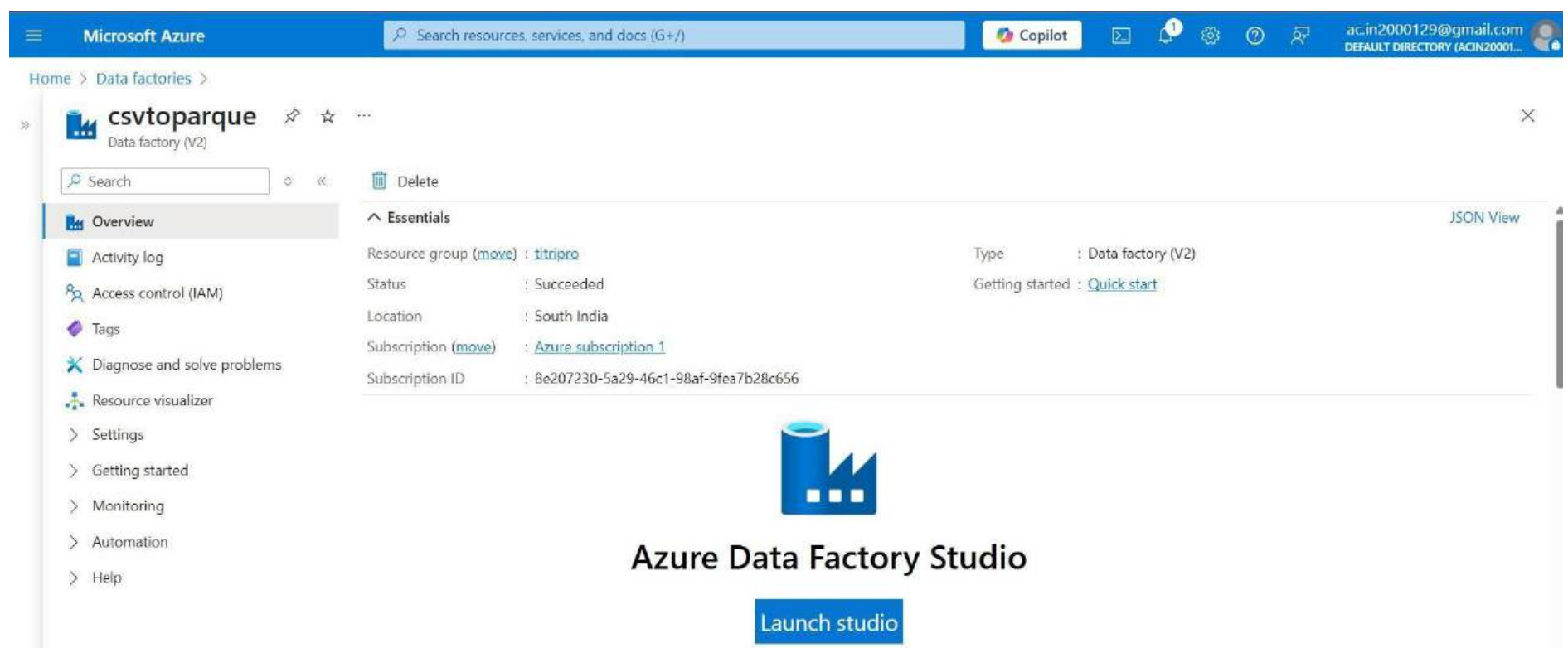


Fig 4: Azure Data Factory is Created, and Ready to Launch

Azure Data Factory (ADF) is a **cloud-based data integration service** that helps orchestrate and automate data movement and transformation at scale. It enables users to build **ETL (Extract, Transform, Load)** and **ELT (Extract, Load, Transform) workflows** to process data from various sources, including on-premises databases and cloud storage.

## FUNCTIONAL REQUIREMENT 3

### Converting CSV File to Parquet File

#### Source:

The Copy Activity in Azure Data Factory (ADF) is a core component of ETL and ELT pipelines, allowing seamless data movement and transformation across various storage solutions. In this implementation, it was utilized to efficiently convert .csv files into .parquet format, optimizing data storage and processing. Parquet, a columnar storage format, offers better performance and compression compared to traditional CSV, making it ideal for large-scale analytics. By leveraging ADF's built-in transformation capabilities, this process ensures structured, high-speed data processing while maintaining integrity and accessibility within the Azure ecosystem.

## Pipeline 1 – CSV → Parquet

- **Trigger:** Azure Blob Storage **event trigger** on .csv arrival in stg/.
- **Implementation:**
  - **Copy Activity (“Copy data1”)** reads from CSV.
  - Converts to Parquet with schema drift handling.
  - Writes Parquet file into preprocess/ with mirrored structure.
  - Enabled logging for traceability.
- **Impact:** Achieved performant storage format suitable for large-scale downstream processing.

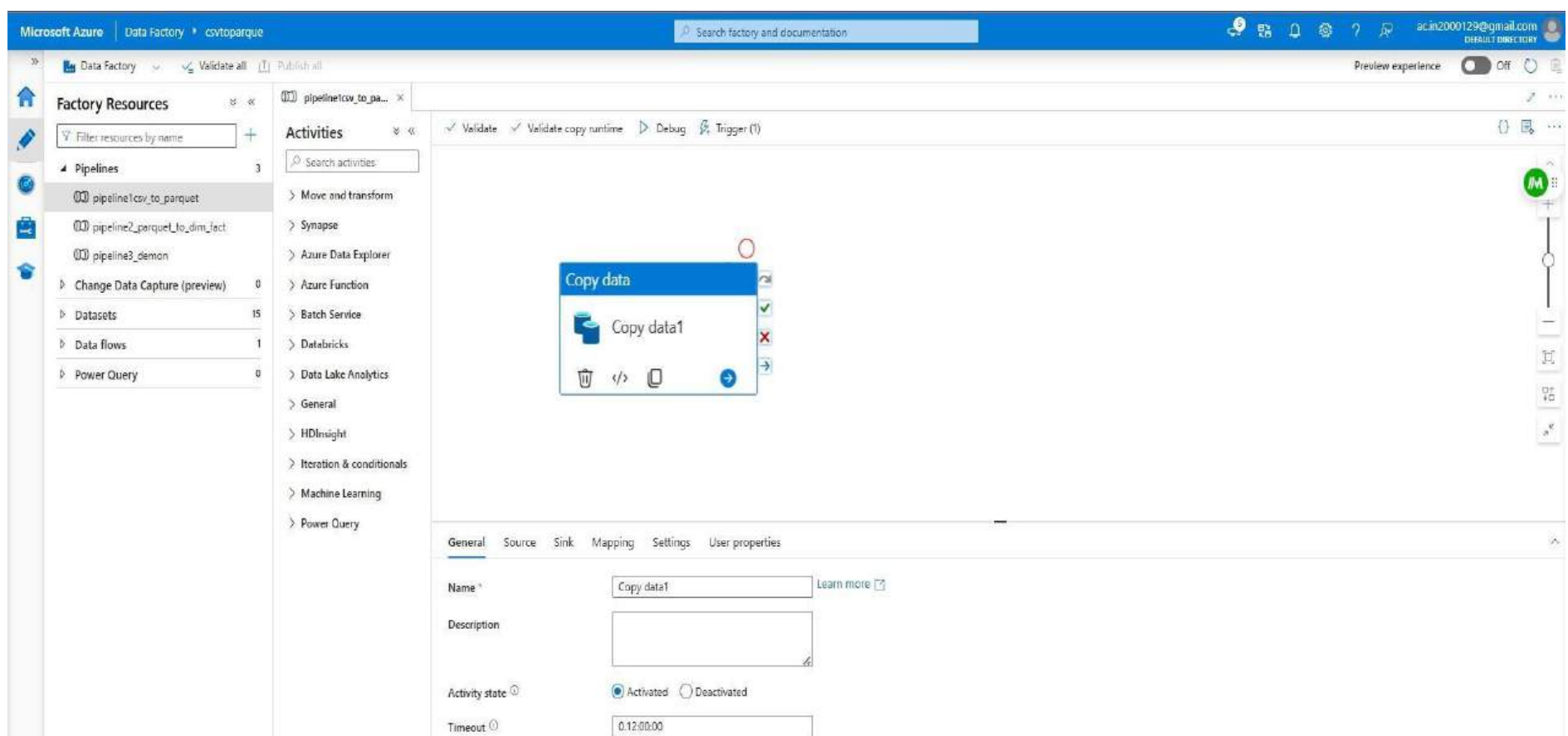


Fig 5: Copy Activity in Azure


TARGET


The parquet file that has been converted from csv file using copy activity

Name	Modified	Access tier	Archive status	Blob type	Size
 [-]					
 subset_1.parquet	6/11/2025, 8:38:40 AM	Hot (Inferred)		Block blob	21.3


Fig 6: After changing the format, the files are stored in pre-process container


Below is the screenshot of trigger and pipeline runs:

 Trigger runs







 Change Data Capture (previ...

Runtimes & sessions

 Integration runtimes

 Data flow debug

Showing 1 - 31 items

Trigger na... ↑↓	Trigger type	Trigger time ↑↓	Status ↑↓	Pipelines	Run	Message	Properties	Run ID
demontrigger	Schedule trigger	6/11/2025, 12:41:1	 Succeeded	1	Original			085845198101...
trigger_preproc...	Storage events ...	6/11/2025, 12:32:4	 Succeeded	1	Original			4ed17cbe-9a29...
csv_to_parquet	Storage events ...	6/11/2025, 12:31:3	 Succeeded	1	Original			d40b663c-4caf...

Microsoft Azure | Data Factory | csvtoparquet

Search factory and documentation

ic.in2000129@gmail.com  
DEFAULT DIRECTORY

Dashboard  
Runs  
Pipeline runs  
Trigger runs  
Change Data Capture (previ...  
Runtimes & sessions  
Integration runtimes  
Data flow debug  
Notifications  
Alerts & metrics

Pipeline runs

Triggered Debug Run Cancel options Refresh Edit columns List Gantt

Filter by run ID or name Chennai, Kolkata, Mu... Last 7 days Pipeline name: All Status: All Runs: Latest runs Triggered by: All Add filter Copy filters Export to CSV

Showing 1 - 36 items

Pipeline name	Run start	Run end	Duration	Triggered by	Status	Run	Parameters	Annotations	Run ID
pipeline3_demo	6/11/2025, 12:41:12 PM	6/11/2025, 12:41:22 PM	10s	demontrigger	Succeeded	Original			7a2aec1b-4504-44bf-9539-ce65b26db44a
pipeline2_parquet_to_dim_fact	6/11/2025, 12:32:45 PM	6/11/2025, 12:35:52 PM	3m 7s	trigger_preprocess_to...	Failed	Original			a0c77ca4-a85f-4089-b0f5-3337452492f1
pipeline1csv_to_parquet	6/11/2025, 12:31:39 PM	6/11/2025, 12:31:57 PM	19s	csv_to_parquet	Succeeded	Original			a051947f-0a91-4f28-bcd1-b832126dd51c
pipeline3_demo	6/11/2025, 12:31:13 PM	6/11/2025, 12:32:51 PM	1m 38s	demontrigger	Succeeded	Original			d3c91fde-463c-4344-bdac-84b241b24bc7
pipeline2_parquet_to_dim_fact	6/11/2025, 12:21:18 PM	6/11/2025, 12:25:01 PM	3m 44s	trigger_preprocess_to...	Succeeded	Original			3c7c8ddc-416f-4385-81b5-761cd97b2b16
pipeline3_demo	6/11/2025, 12:21:12 PM	6/11/2025, 12:21:22 PM	11s	demontrigger	Succeeded	Original			c28e8cb2-78ba-43f9-b5ee-07e077a092ac
pipeline1csv_to_parquet	6/11/2025, 12:20:56 PM	6/11/2025, 12:21:16 PM	20s	csv_to_parquet	Succeeded	Original			17490dcf-1f3a-4e97-ad8a-e5b7732279c2
pipeline2_parquet_to_dim_fact	6/11/2025, 12:17:24 PM	6/11/2025, 12:20:08 PM	2m 44s	trigger_preprocess_to...	Failed	Original			63fed2d6-7004-4858-98d1-4e5d9af60011
pipeline1csv_to_parquet	6/11/2025, 12:17:23 PM	6/11/2025, 12:17:41 PM	19s	csv_to_parquet	Succeeded	Original			1cb0e47b-a6fa-41de-936a-4f5c1ddc6234
pipeline3_demo	6/11/2025, 12:16:59 PM	6/11/2025, 12:18:13 PM	1m 15s	Manual trigger	Succeeded	Original			1da3e139-b4d3-4670-a01c-8f086f3ac7f0
pipeline2_parquet_to_dim_fact	6/11/2025, 11:06:22 AM	6/11/2025, 11:10:39 AM	4m 17s	trigger_preprocess_to...	Succeeded	Original			8f175249-109f-4d5c-a972-9f1839ada57a
pipeline1csv_to_parquet	6/11/2025, 11:06:02 AM	6/11/2025, 11:06:21 AM	19s	csv_to_parquet	Succeeded	Original			64056774-a4f2-4b63-92c3-110a51c2d17d

Last refreshed 0 minutes ago

## FUNCTION REQUIREMENT 4

Creating a dataflow to split the data into the Fact and Dimensions Table

### Source

The primary requirement of this process was to convert .csv files into .parquet format using Copy Activity in Azure Data Factory (ADF). This transformation enhances data storage efficiency by leveraging Parquet’s columnar format, which optimizes compression and query performance. The use of ADF’s Copy Activity ensures seamless data movement and transformation between various sources and destinations, facilitating high-speed, scalable data processing across Azure cloud services.

### Data Transformation and Table Creation

#### Trigger Condition:

The transformation pipeline initiates once a Parquet file is detected in the /preprocess/ folder. This structured approach enables automated and



incremental data processing, ensuring continuous ingestion and refinement of incoming datasets.

#### Processing Steps:

- Retrieve the latest Parquet subset from /preprocess/ for transformation.
- Apply a dataflow to refine the dataset by selecting and aggregating relevant columns for fact and dimension tables.
- Dynamically generate surrogate keys to maintain a structured sequence, ensuring smooth data integrity across updates.
- Append transformed data to the respective fact and dimension tables, reinforcing data accuracy and analytical consistency.

#### Azure Data Flows Transformations

Azure Data Flows in ADF enable users to perform scalable, visual data transformations without coding. These transformations execute on ADF-managed Apache Spark clusters, ensuring efficient big data processing while maintaining schema integrity.

- **Select Transformation:** Allows users to refine datasets by renaming, removing, and reordering columns, ensuring schema consistency and supporting dynamic expressions for precise data modifications.
- **Aggregate Transformation:** Works similarly to SQL's GROUP BY, facilitating calculations such as SUM, COUNT, AVG, MIN, MAX, and more. This helps summarize and structure data for advanced reporting and analytical insights.

#### Surrogate Keys C Data Integrity

A surrogate key is a unique identifier assigned to each dataset row, ensuring consistent, business-independent indexing—a key feature of dimension tables within a star schema. These keys provide structural consistency, improving data retrieval performance and maintaining referential integrity in analytical processes.

#### Sink Container: Fact and Dimension Tables

The sink container efficiently stores transformed data in Fact C Dimension Tables, ensuring optimized data structuring for analytical processing.

#### Fact Tables



Fact tables serve as the quantitative backbone of business analytics. They store measurable data such as sales amount, revenue, and transaction volumes, alongside foreign keys linking dimension tables. Fact tables are designed for aggregation, enabling powerful queries and trend analysis for business intelligence.

Dimension Tables

Dimension tables provide contextual details for fact tables, encompassing attributes like product names, customer profiles, or time periods. These descriptive elements allow analysts to categorize, filter, and segment data, making it easier to extract actionable insights for decision-making and performance evaluation.

Target

The structured processing pipeline ensures efficient splitting of the CSV file into Fact C Dimension Tables using ADF’s Data Flow functionality. This systematic approach guarantees optimal data storage, organization, and analytics, enabling precision-driven insights within the healthcare claim dataset.

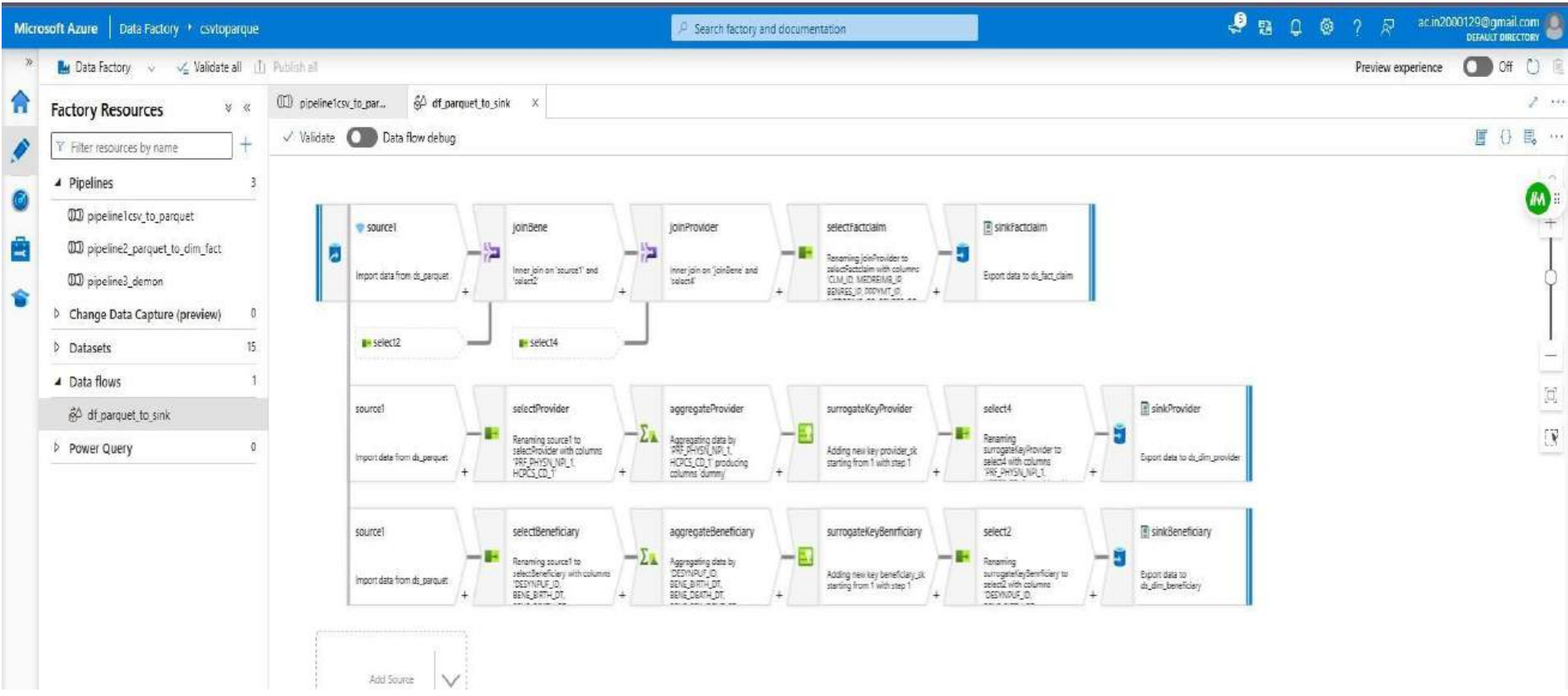


Fig 7: Data Flow to split the Parquet file into Facts and Dimensions Tables

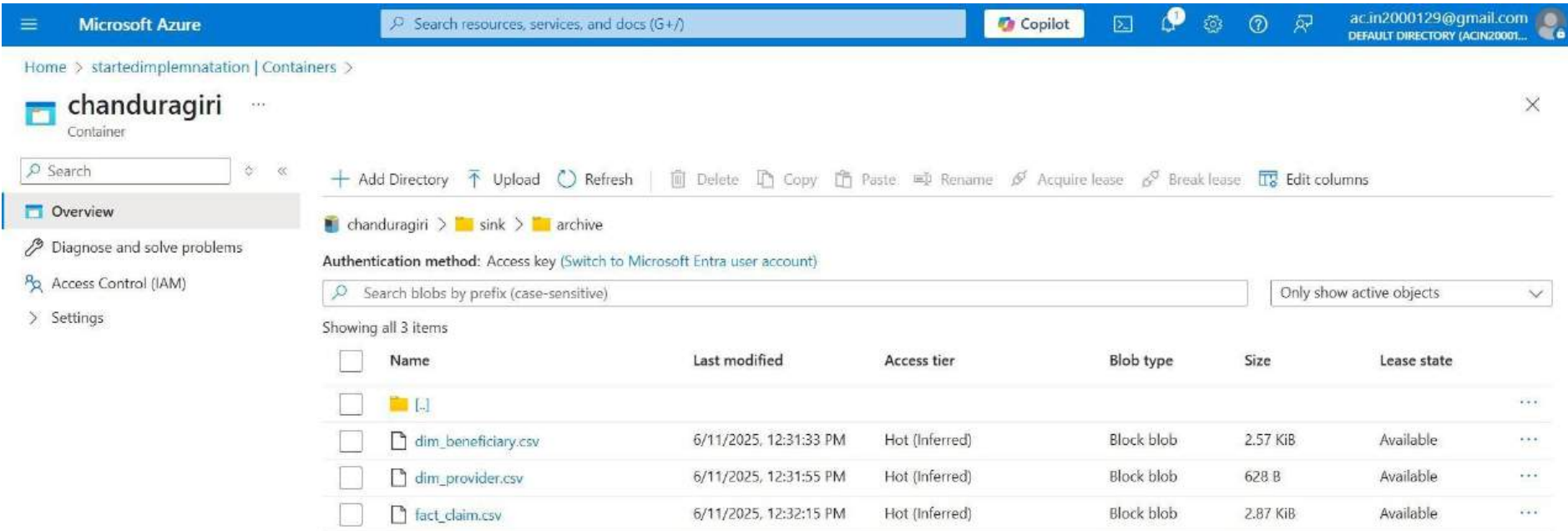


Fig 8: The sink container having the Fact and Dimension Table

## FUNCTION REQUIREMNET 5

Archive and Purging is done at this stage

### Archiving and purge Strategy

To maintain a **clean and efficient data architecture**, an **archiving and purging mechanism** is implemented to manage processed files systematically. The **archiving strategy** ensures historical data remains accessible while preventing clutter in active processing directories. This is achieved by **moving processed files to the /archive/ folder after one day**, allowing users to retrieve past data without interfering with ongoing operations.

To further **optimize storage and improve efficiency**, a **purging strategy** is employed. **Archived files are automatically deleted after two days**, ensuring that outdated data does not accumulate unnecessarily. The process is executed using **Copy Activity in Azure Data Factory (ADF)** to structure the archiving sequence, while the **Delete Activity** is responsible for removing files after the defined retention period. The workflow is designed so that any file **not modified for more than 24 hours is moved to the archive folder**, and files **not modified for over 48 hours are permanently deleted**.

This approach maintains **storage efficiency**, **enhances system performance**, and ensures that critical data remains accessible while obsolete records are systematically removed.

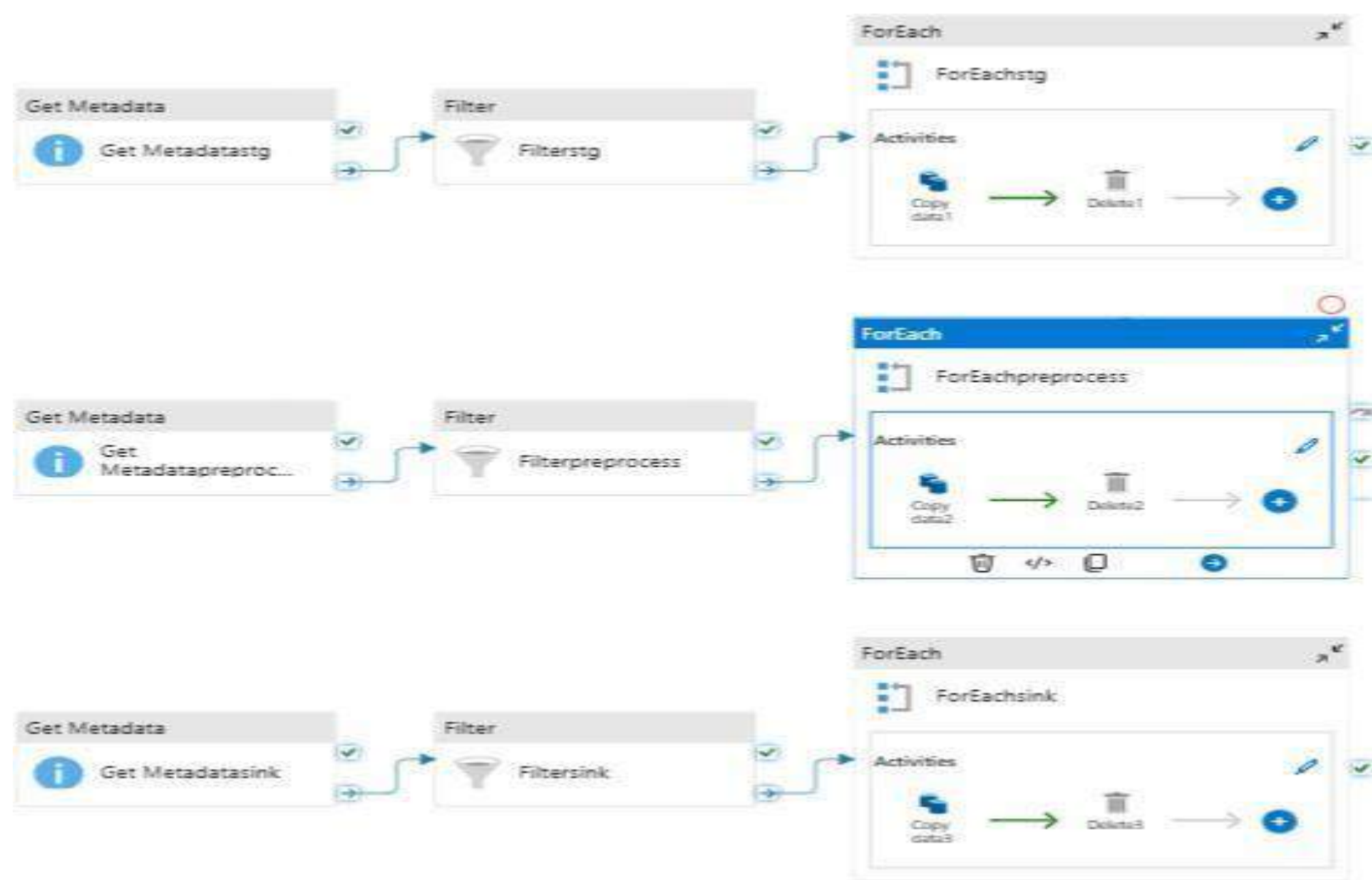


Fig 9: Copy and Delete Activity for each container  
Linked services that I created for this project:

Name	Type	Related	Annotations
AzureDataLakeStorage1	Azure Data Lake Storage Gen2	0	
ls_csv	Azure Data Lake Storage Gen2	3	
ls_perquet	Azure Data Lake Storage Gen2	2	
ls_sink_adls	Azure Data Lake Storage Gen2	5	
rpc	Azure Data Lake Storage Gen2	6	

Fig 10: Linked services utilized

Triggers that I have created and utilized for this project:

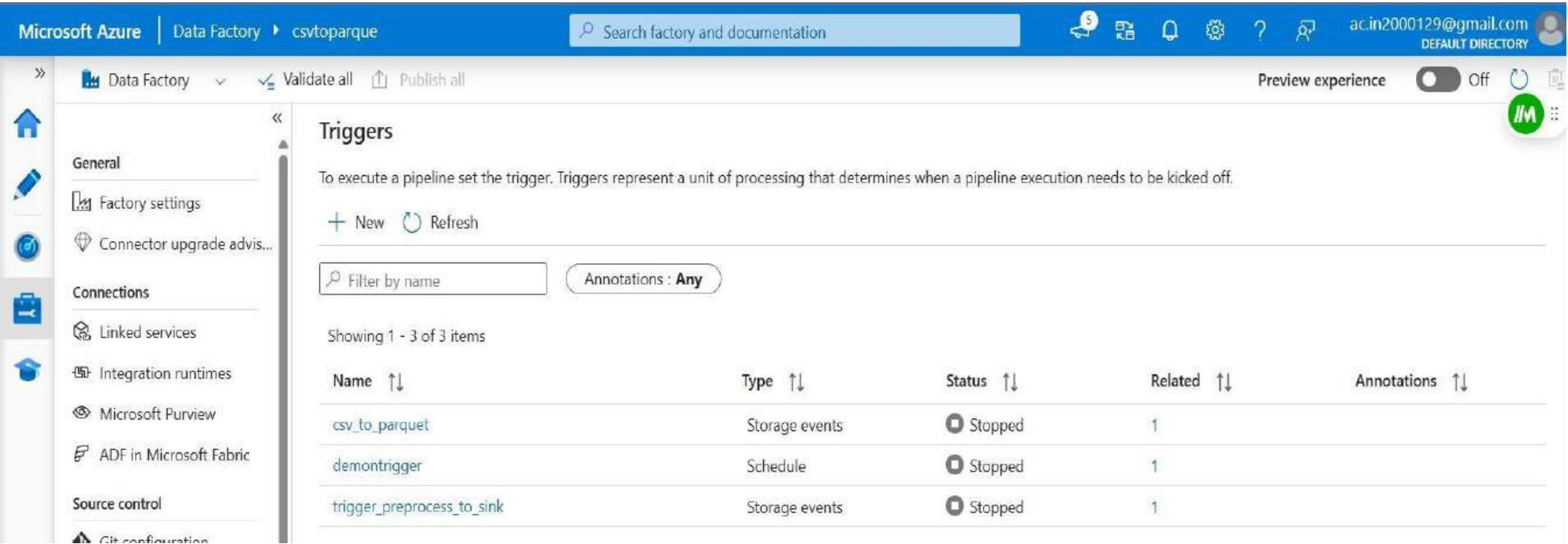


Fig 11: Triggers utilized

Once processed files were moved to the **archive folder**, they remained available for **retrieval and reference**. However, to prevent unnecessary accumulation of outdated data, an automated **purging mechanism** was introduced. This ensured that archived files were **deleted after two days**, optimizing storage and system performance.

To achieve this, you leveraged **Azure Storage Lifecycle Management**, setting policies that tracked **file modification timestamps**. Files **older than 24 hours** were moved to the archive, while those **older than 48 hours** were permanently deleted. This automation ensured a **seamless data lifecycle**, preventing manual intervention and enhancing operational efficiency.

By integrating **Azure’s lifecycle policies**, your pipeline maintained a **clean, structured, and scalable** storage framework. This approach minimized **redundancy**, reduced **storage costs**, and kept the processing environment optimized for **continuous data ingestion**.



Microsoft Azure

Search resources, services, and docs (G+)

Home > Storage accounts > startedimplemnnatation | Lifecycle management >

Update a rule ...

Details

Base blobs

Filter set

Lifecycle management uses your rules to automatically move blobs to cooler tiers or to delete them. If you create multiple rules, the associated actions must be implemented in tier order (from hot to cool storage, then archive, then deletion).

If

Base blobs haven't been modified in 2 days

Then

Delete the blob

+ Add conditions

Fig12: added the rules of the life cycle management to delete the Files

Microsoft Azure

Search resources, services, and docs (G+)

Copilot

ac.in2000129@gmail.com

DEFAULT DIRECTORY (ACIN20001...

Home > Storage accounts > startedimplemnnatation | Lifecycle management >

Update a rule ...

Details

Base blobs

Filter set

Blob prefix

Filter blobs by name or first letters. To find items in a specific container, enter the name of the container followed by a forward slash, then the blob name or first letters. For example, to show all blobs starting with "a", type: "mycontainer/a".

Blob prefix

chanduragiri/stg/archive/

chanduragiri/preprocess/archive/

chanduragiri/sink/archive/

Enter a prefix or file path such as "mycontainer/prefix"

Fig 13: Filters the blob that is to be deleted

## Conclusion

This **Azure-based data processing pipeline** offers a **structured, automated, and scalable** solution for **efficient medical claims data management**, ensuring seamless integration across multiple Azure services. By incorporating **PySpark**, the system enables **optimized dataset manipulation**, enhancing **processing speed, data accuracy, and analytical reliability**.

The **Azure Data Lake Storage (ADLS)** framework efficiently organizes data into **incoming, processed, and archived containers**, ensuring **smooth transitions and structured data storage**. Meanwhile, **Azure Data Factory (ADF)** automates **incremental data ingestion, format conversions, and table creation**, facilitating a **streamlined ETL workflow**. Through **event-driven triggers**, the pipeline continuously processes **new claim datasets every hour**, while a **robust archiving and purging mechanism** maintains historical records without overloading storage.

The **dynamic surrogate key generation** mechanism guarantees **data integrity and consistency**, ensuring fact and dimension tables remain structured for **advanced analytical insights**. Additionally, automated **scheduling and dependency management** enable the pipeline to **adapt dynamically** to incoming claim data, guaranteeing **real-time updates** and **optimized storage utilization**.

This **end-to-end solution** enhances data processing efficiency, ensures **structured analytical outputs**, and supports incremental load scenarios, making it an **ideal framework for large-scale medical claims analysis and reporting**. The combination of **cloud-native architecture, automation, and scalability** solidifies its role as a **reliable, future-ready system for data-driven healthcare insights**.