**Capstone Project Submission Report**

# E-Commerce Data Engineering Pipeline with Azure and Databricks

**Name**: **Jatin Midha**

**Employee ID**: **127287**

**Course/Program**: **Data Engineering**

**Instructor Name**: **Mr. Piyush Raj Katayan**

# Table of Contents

Problem Statement & Overview 3

Workflow Architecture 4-8

Data Collection, Exploratory Data Analysis, and Data Preprocessing 8

Data Storage and Optimization 9-10

Real-Time Processing and Streaming 11-13

Solution Design & Integration 14-15

Implementation and Results 16-12

Working Screenshots 13-34

Conclusion and Future Work 35

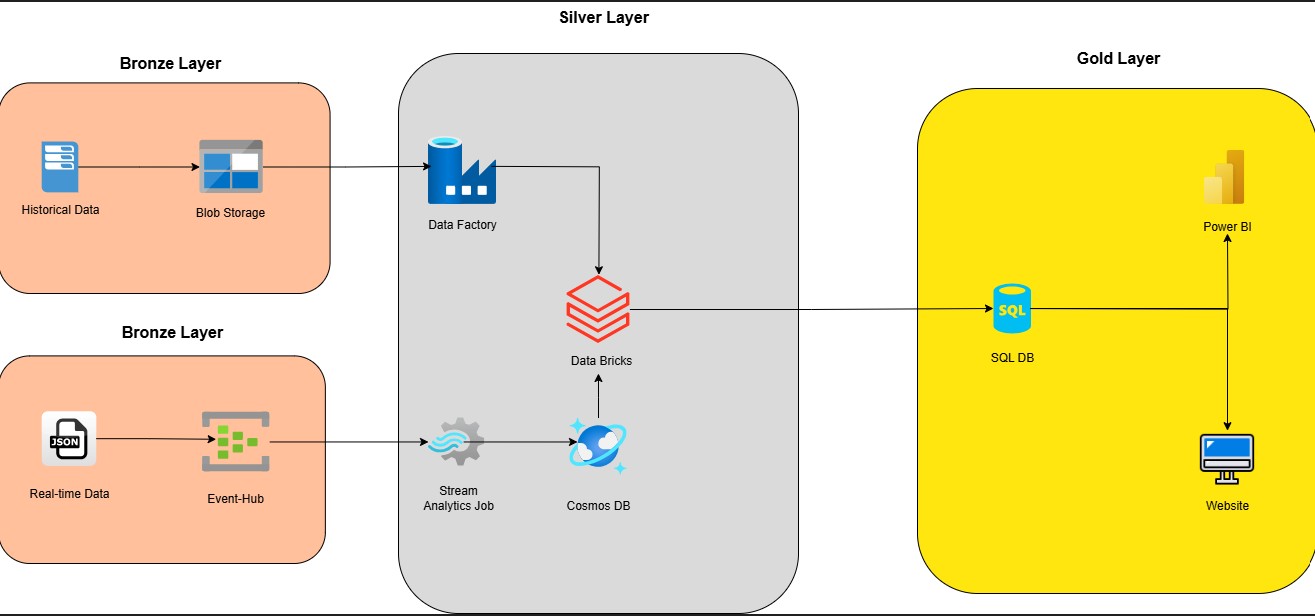
## Problem Statement

The task involves developing a comprehensive data processing and analytics solution for a fictional ecommerce company. This company processes millions of transactions daily, necessitating the analysis of this data to extract valuable business insights. The solution must be capable of ingesting raw transactional data from various sources, including APIs, structured databases, and event-driven streams. Subsequently, the data must be transformed and loaded into a database for reporting and analysis. The solution must be highly available, scalable, secure, and optimized for cost efficiency.

## Objective

Implement an ETL pipeline using Azure services and Databricks to process, analyze, and visualize large datasets, adhering to production-level requirements. We will design, deploy, and present an end-to-end solution that integrates various Azure services, focusing on real-world data engineering challenges.

# Workflow Architecture



## Bronze Layer

* **Data Sources**: Historical data is stored in Blob Storage, and real-time data is ingested through Event Hub.
* **Data Processing**: Stream Analytics Job is used to process real-time data as it arrives.
* **Purpose**: This layer captures raw data from various sources and performs initial processing on real-time data. It serves as the landing zone for all incoming data.

## Silver Layer

* **Data Processing**: Data Factory is used to further process and transform the data.
* **Data Storage**: The processed data is stored in SQL DB and Cosmos DB.
* **Purpose**: This layer cleanses, validates, and enriches the data. It involves deduplication, transformation, and integration of data from different sources.

## Gold Layer

* **Data Consumption**: The refined data is made available for business intelligence and analytics through Power BI and a website.
* **Purpose**: This layer contains highly curated, business-ready data optimized for reporting, dashboards, and advanced analytics.

## Data Collection, Exploratory Data Analysis(EDA), and Data Preprocessing

**Data Collection:**

The provided datasets are:

### Historical Data

1. **customer\_data.csv**: Contains customer information such as names, contact details, and demographic data.

### Table: Data Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Customer\_id | Integer | Id of Customers | 0% |
| First\_name | String | Customer’s first name | 0% |
| Last\_name | String | Customer’s last name | 0% |
| Signup\_date | Date | Signup date of cust. | 0% |
| Address | String | Customers address | 0% |
| Email | String | Customers email | 0% |

2. **inventory\_data.csv**: Includes details about the products in stock, such as item names, quantities, and prices.

### Table: Data Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Product\_id | Integer | Id of product | 0% |
| Current\_stock | Integer | Stock Available | 0% |
| Reorder\_level | Integer | Items used | 0% |

3. **Product\_data.csv**: Similar to the previous file, contains data of product inventory with price, categories.

### Table: Data Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Product\_id | Integer | Id of product | 0% |
| Product\_name | String | Name of Product | 0% |
| Stock\_Quantity | Integer | Items available | 0% |
| Category | String | Categories of product | 0% |
| Price | Integer | Price of Product | 0% |

1. **Reviews\_data.csv**: Holds customer reviews and ratings for various products or services. **Table: Data Description**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Review\_id | String | Review ID | 0% |
| Customer\_id | String | Customer ID | 0% |
| Product\_id | String | Product ID | 0% |
| Rating | Integer | Product Rating | 0% |
| Review\_Text | String | Customer Feedback | 0% |
| Review\_date | Date | Date of review | 0% |

1. **Transaction\_data.csv**: Records transaction details, including purchase dates, amounts, and customer IDs.

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Transaction\_id | String | Review ID | 0% |
| Customer\_id | String | Customer ID | 0% |
| Transaction\_date | date | Transaction date | 0% |
| Product\_id | String | Product ID | 0% |
| Quantity | Integer | Product Quantity | 0% |
| Payment\_Type | String | Mode of Payment | 0% |
| Transaction\_amount | Float | Transaction amount | 0% |

### Real-Time Data

1. **Realtime transactions.json:** Contains real-time transaction data including transaction IDs, timestamps, amounts, and customer details.

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** | **Missing Values (%)** |
| Transaction\_id | String | Review ID | 0% |
| Customer\_id | String | Customer ID | 0% |
| Transaction\_date | date | Transaction date | 0% |
| Product\_id | String | Product ID | 0% |
| Quantity | Integer | Product Quantity | 0% |
| Payment\_Type | String | Mode of Payment | 0% |
| Transaction\_amount | Float | Transaction amount | 0% |

### Data Preprocessing

The data preprocessing phase involved cleaning the raw transactional data using PySpark in Databricks. This process is crucial for ensuring the quality and reliability of the data before it is used for analysis and reporting. Here is a detailed explanation of the steps involved:

1. **Data Ingestion**:

The raw data, stored in CSV files in Azure Blob Storage, was ingested into Databricks using PySpark. Databricks provides a collaborative environment for data engineering and data science, making it an ideal platform for preprocessing large datasets.

1. **Data Cleaning**:

**Removing Duplicates**: Duplicate records were identified and removed to ensure that each transaction is unique. This was achieved using the dropDuplicates() function in PySpark.

**Handling Missing Values**: Missing values were handled by either filling them with appropriate default values or by removing the records with missing critical information. The dropna() functions in PySpark was used for this purpose.

#### Table: Data Preprocessing Summary

|  |  |  |
| --- | --- | --- |
| **Step** | **Technique** | **Result** |
| Checked Null Values | Counting each column null | Consistent Data |
| Checked Duplicates | Using filter and count together | Clean data without duplicates |

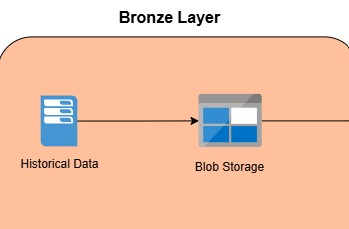
**Data Storage and Optimization**

## For Historical Data

**Azure Blob Storage:**

**Purpose:** Used for extracting and storing raw transactional data from CSV files.

Reason for Choice: Azure Blob Storage provides scalable and cost-effective storage for unstructured data. It supports a wide range of data formats and integrates seamlessly with other Azure services. Blob Storage offers high durability and availability, making it an ideal choice for storing large volumes of raw data.



## For Real Time Data Storage

**Azure Cosmos DB:**

**Purpose:** Used for storing real-time data from Event Hub.

Reason for Choice: Azure Cosmos DB is a globally distributed, multi-model database service that provides low-latency and high-throughput performance. It is designed to handle real-time data streams and supports automatic scaling to accommodate varying workloads. Cosmos DB offers comprehensive security features, including encryption and fine-grained access control, ensuring the protection of real-time data.

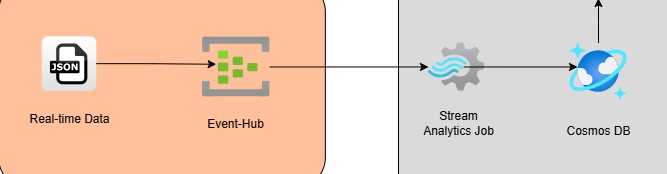


Fig. Real time data storage

**Azure SQL Database:**

**Purpose:** Used for storing transformed data for reporting and analysis. Reason for Choice: Azure SQL Database is a fully managed relational database service that offers high performance, scalability, and security. It supports automated backups, patching, and scaling, ensuring high availability and reliability. The service also provides advanced security features, including encryption and threat detection, making it suitable for storing sensitive business data.

**Data Storage Architecture:**

**Partitioning & Indexing Strategies**:

**Composite Indexing**:

* **Technique**: Composite Indexing
* **Description**: Composite indexes were created on multiple columns that are often used together in queries, such as customer ID and transaction date. This technique helps in optimizing complex queries that involve multiple columns.
* **Result**: Improved performance for multi-column queries, as the composite index allows for efficient data retrieval based on multiple criteria.

### 6. Real-Time Data Processing and Streaming

**Real-Time Data Processing**:

**Azure Event Hubs for Data Ingestion** – Real-time transaction data is streamed into Azure Event Hubs, acting as a highly scalable event ingestion service.

**Azure Stream Analytics for Data Processing** – Incoming transaction data is processed using Azure Stream Analytics (ASA) to perform transformations, aggregations, and anomaly detection in real time.

**Apache Spark on Azure Databricks for Advanced Processing** – Real-time fraud detection, risk assessment, and customer behavior analysis are performed using Spark Structured Streaming in Azure Databricks.

**Delta Lake for Streaming Data Storage** – Streamed data is written into Delta Lake tables in ADLS (Bronze Layer), ensuring ACID compliance and real-time updates.

**Power BI & Alerts for Instant Insights** – Real-time dashboards in Power BI display live transaction trends, and Azure Functions trigger alerts for suspicious activities like potential fraud.

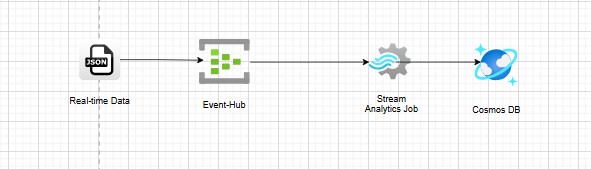


Fig. Real time data entering Cosmos DB

**Triggering Events** :

* **Fraud Detection Alerts** : If a transaction amount exceeds a predefined threshold or an unusual pattern is detected, an Azure Function triggers an alert.
* **Service Availability Monitoring** : If a bank service is down or slow, an Event Grid notification is sent to the IT support team for immediate action.
* **Real-Time Risk Assessment** : High-risk transactions (e.g., large withdrawals, rapid transactions in different locations) trigger real-time risk assessment models to flag suspicious activity.

### Solution Design & Integration

* **Data Extraction from Multiple Sources** : Historical data is pulled from blobstorage via Azure Data Factory (ADF), and real-time data is ingested using Azure Event Hubs from banking transactions.
* **Transformation & Cleansing in Databricks :** Raw data is processed in Azure Databricks using Apache Spark, including data validation, missing value handling, and feature engineering.

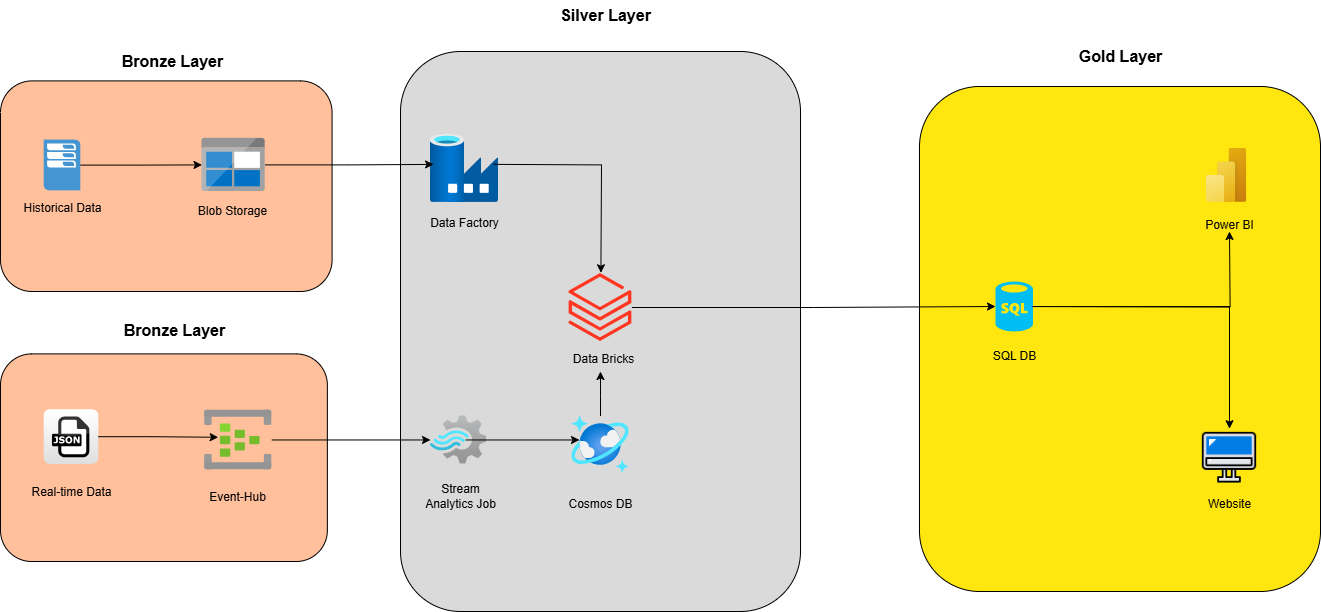
* **Loading into Medallion Architecture :** Processed data is stored in Azure Data Lake Storage (ADLS) following the Bronze (raw), Silver (cleaned), and Gold (aggregated) layers.

* **Batch & Real-Time Processing Integration :** Historical transaction data is processed in batch mode while real-time streaming is handled with Stream Analytics.

* **Orchestration Using Azure Data Factory :** ADF Pipelines automate data movement between layers, trigger transformations, and integrate batch

+ real-time processing for seamless operations.

### System Architecture



**Automated Workflows**:

* **ADF Pipeline for ETL Automation** : ADF triggers data extraction, transformation, and storage using scheduled pipelines, reducing manual intervention.
* **Event-Driven Processing with Azure Functions :** Fraud detection triggers real-time alerts based on transaction anomalies, ensuring quick action.
* **CI/CD Deployment with Azure DevOps :** Automated deployments using GitHub Actions and Azure DevOps Pipelines, ensuring seamless updates to ETL workflows.

#### Implementation and Results

**Architecture:**

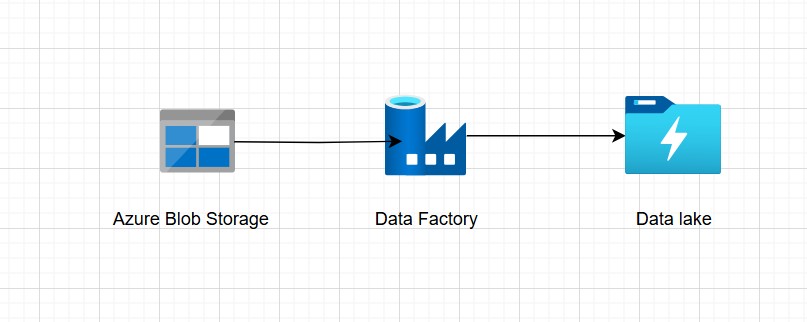


Fig. Historical data flow

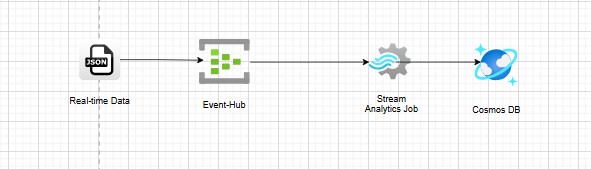


Fig. Real time data flow

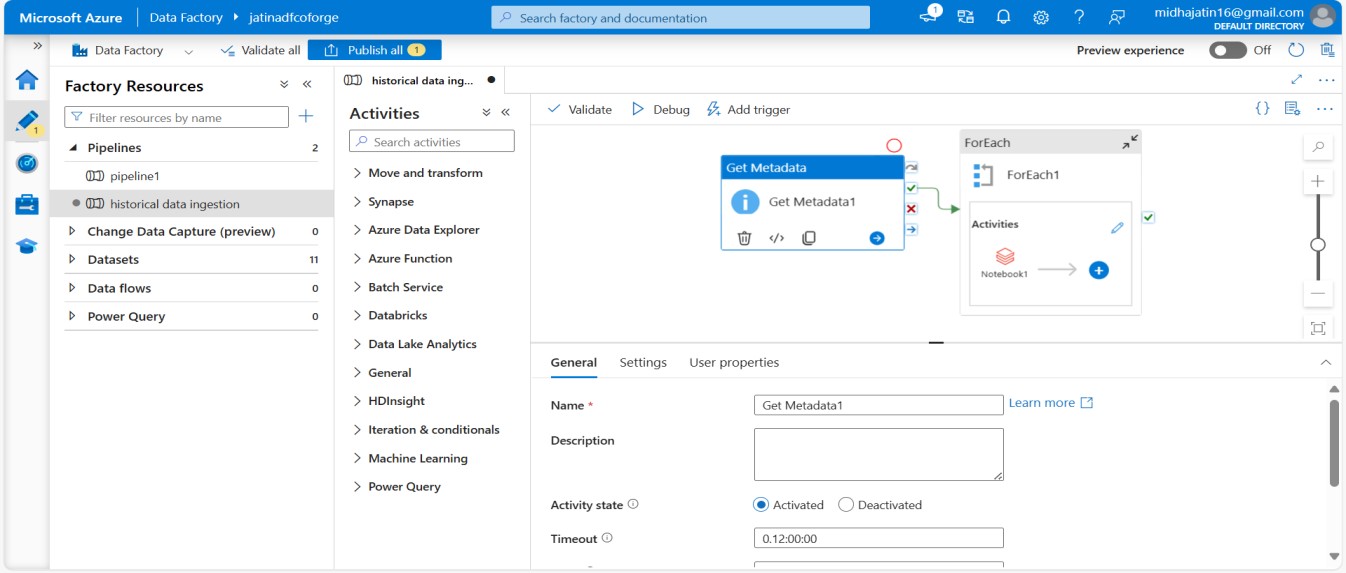


Fig. Ingesting historical data from blob to data bricks towards delta lake

# Connection details as variables

cosmos\_endpoint = ["https://jatincosmosdb.documents.azure.com:443/"](https://jatincosmosdb.documents.azure.com/)

cosmos\_key =

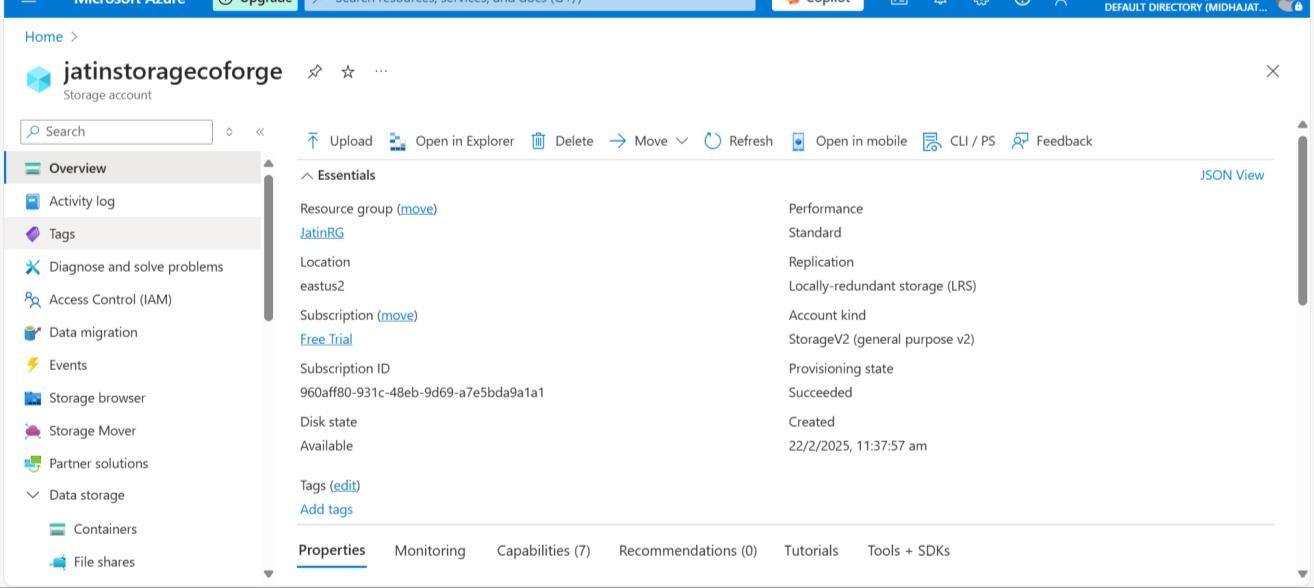
|  |
| --- |
| "IOvGwpuVgubfdwdSgHknWaUfPFU8BJYKoX9TMOkac5i7Wbv24G5MSvOy8iVt9sE2TXl10rGiI5sBACDbb69zrw==" database\_name = "jatincosmosdb" container\_name = "livedatastorage"    # Read from Cosmos DB cosmos\_df = spark.read \  .format("cosmos.oltp") \  .option("spark.cosmos.accountEndpoint", cosmos\_endpoint) \  .option("spark.cosmos.accountKey", cosmos\_key) \  .option("spark.cosmos.database", database\_name) \  .option("spark.cosmos.container", container\_name) \  .load()    # Select only required columns (remove Cosmos DB metadata) cosmos\_realtime\_df = cosmos\_df.select( col("transaction\_id"), col("customer\_id"), col("transaction\_date"), col("product\_id"), col("quantity"), col("payment\_type"), col("transaction\_amount") ) |

Connecting Cosmos to databricks for transformations

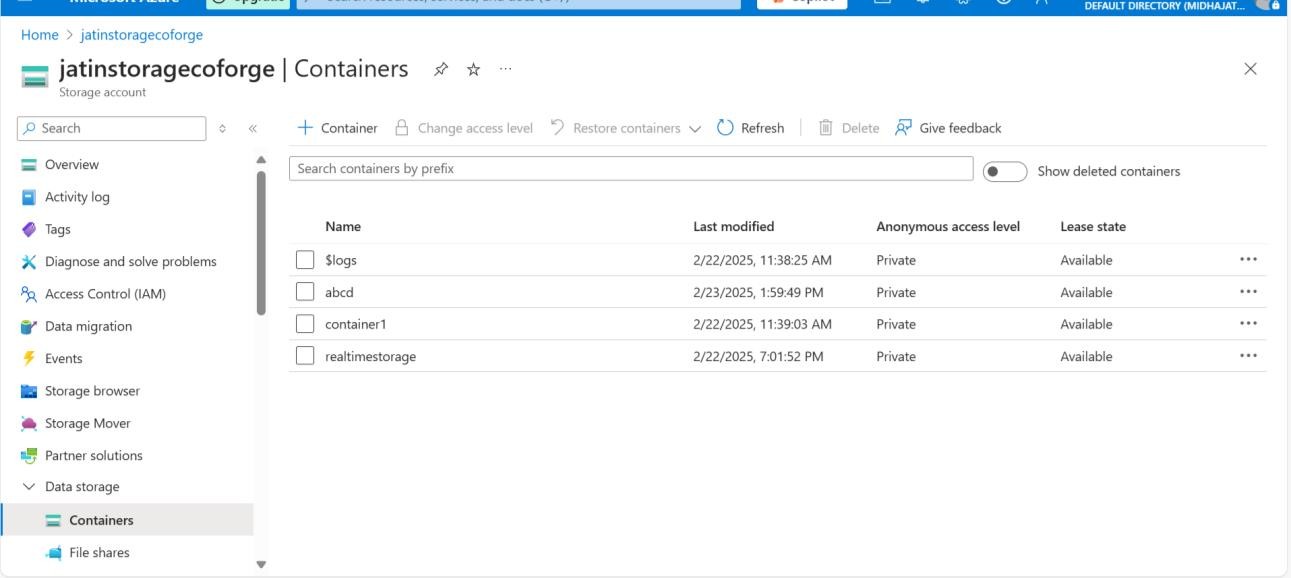
## WORKING SCREENSHOTS

**For historical data:**

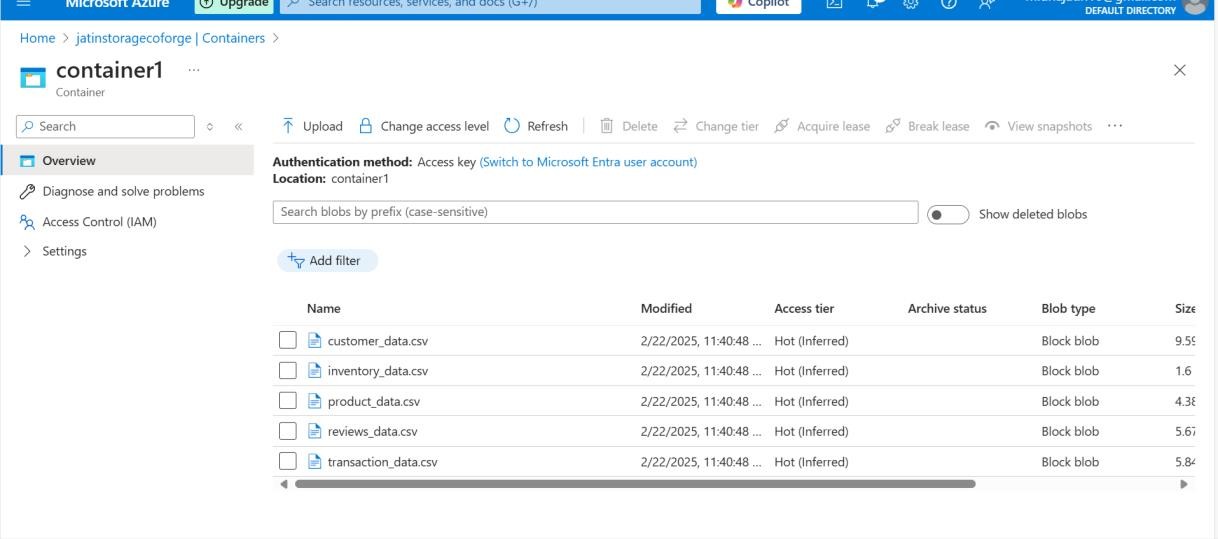
### Creating Storage Account



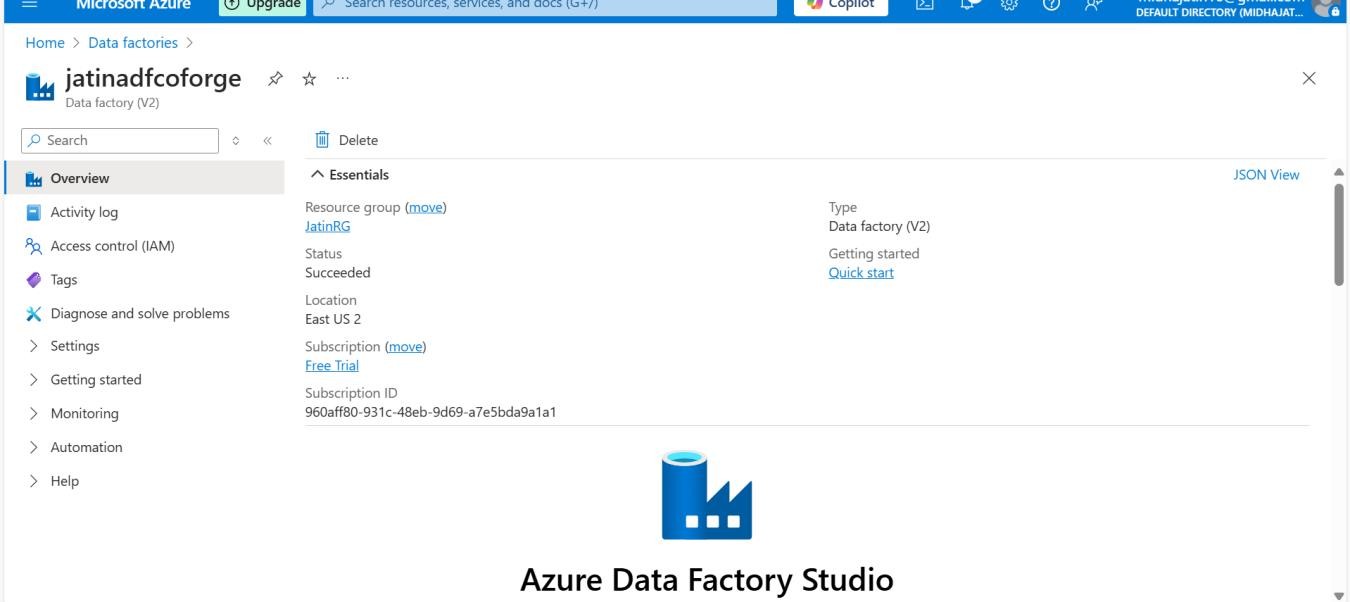
### Creating Containers in the storage account



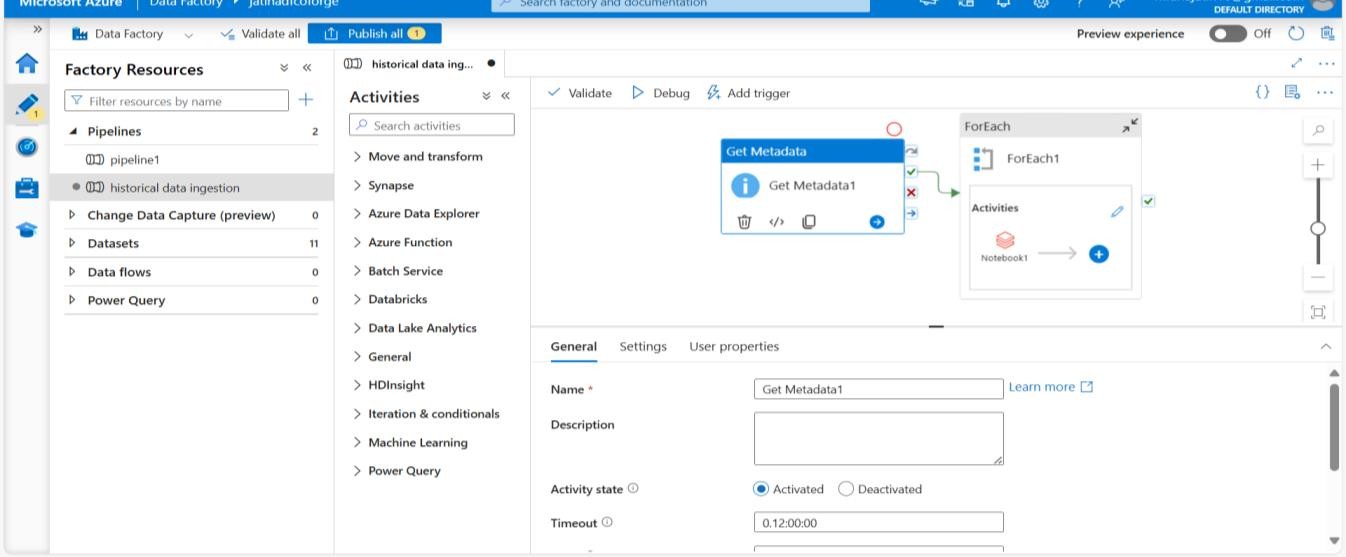
**Blob storage where csv’s were stored**



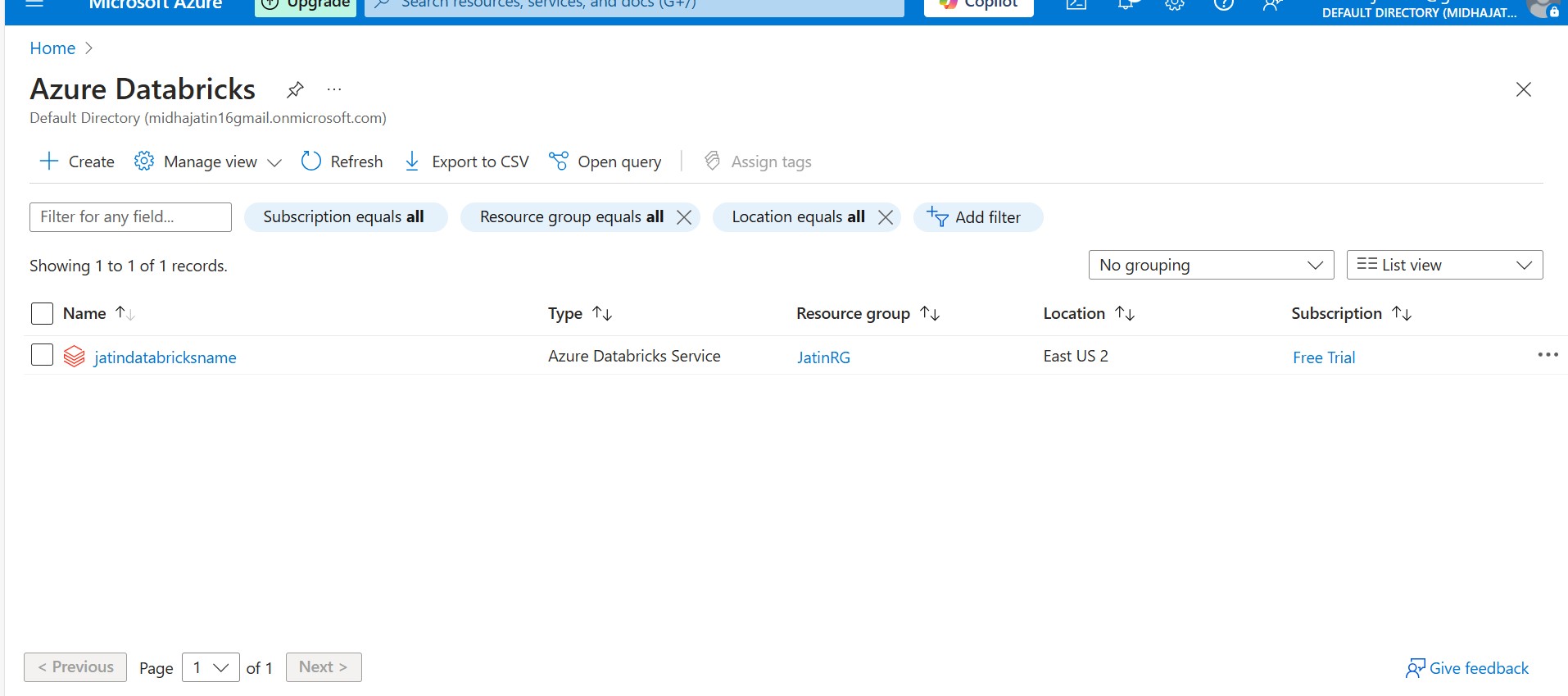
### Initializing Data Factory



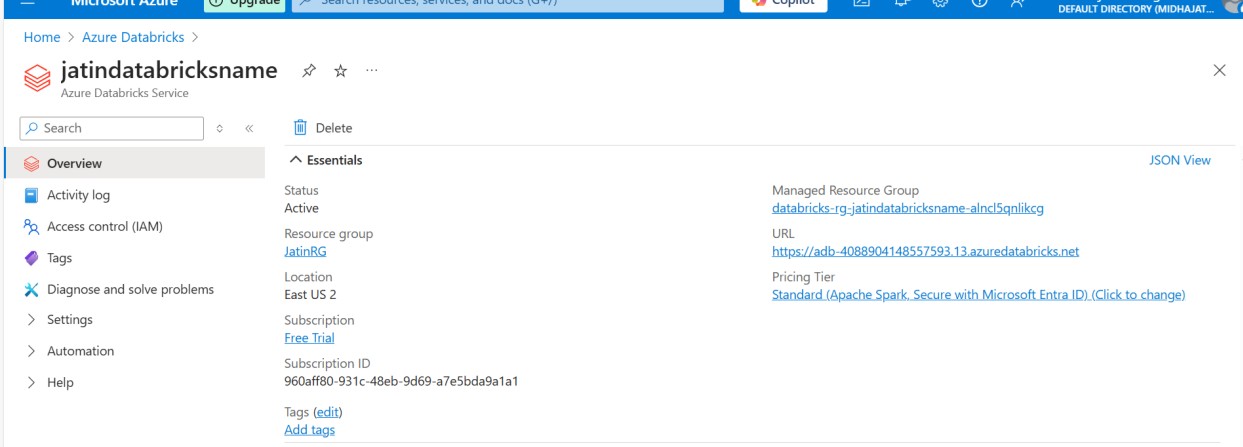
### Ingesting data from Blob Storage through Data Factory

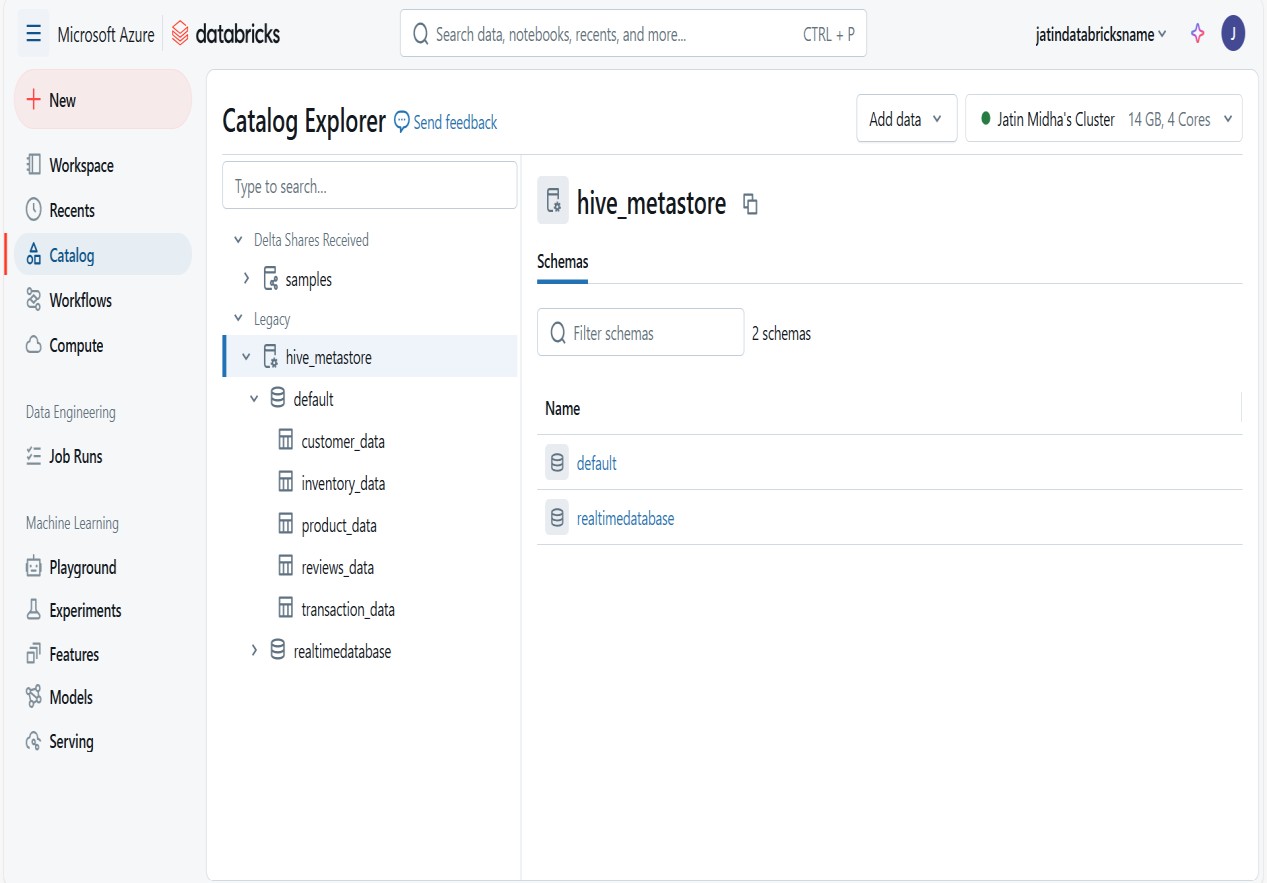


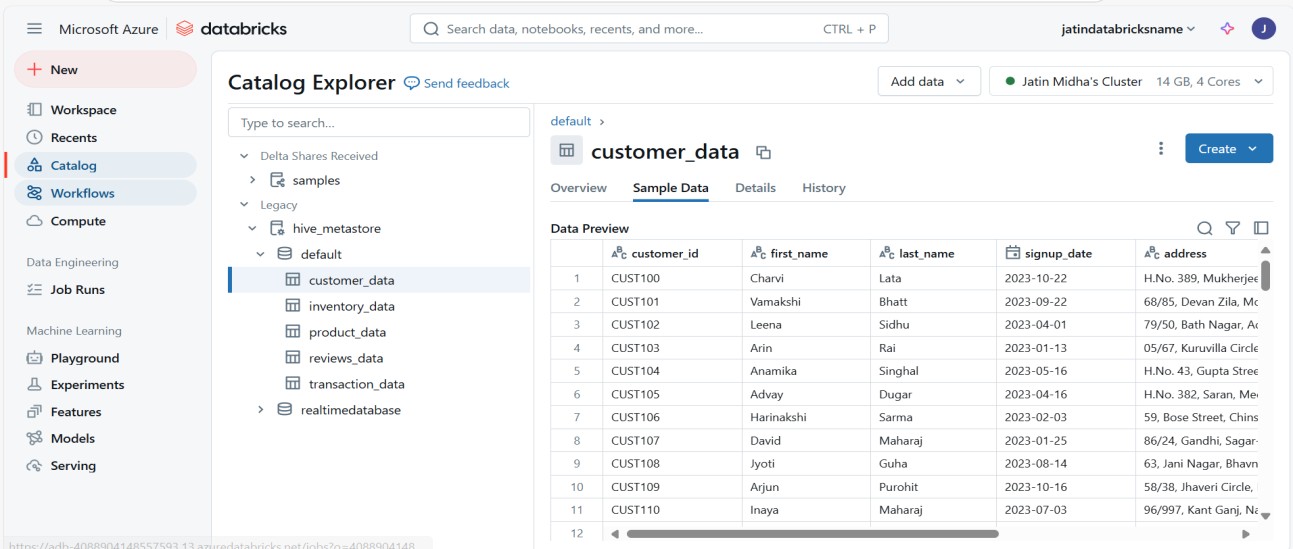
### Opening databricks

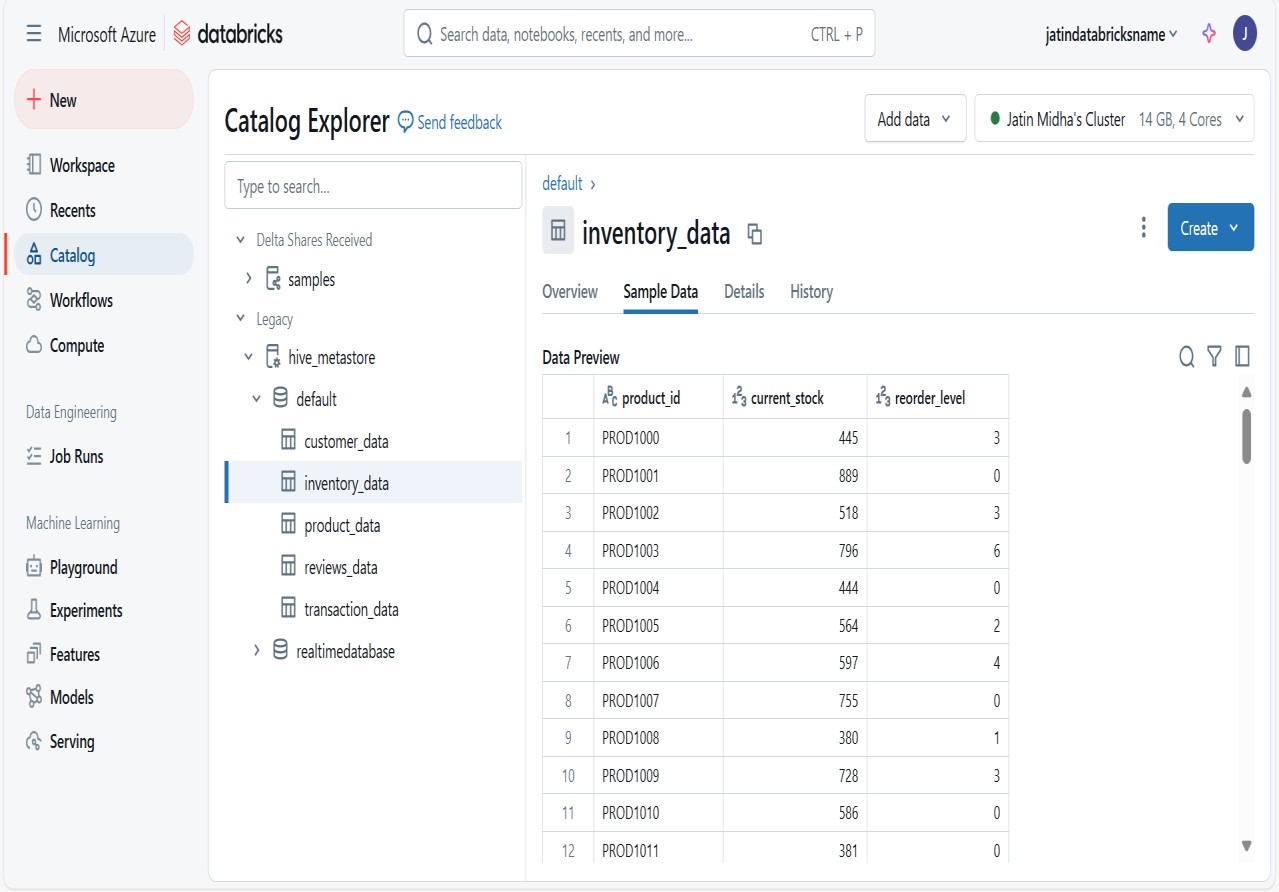


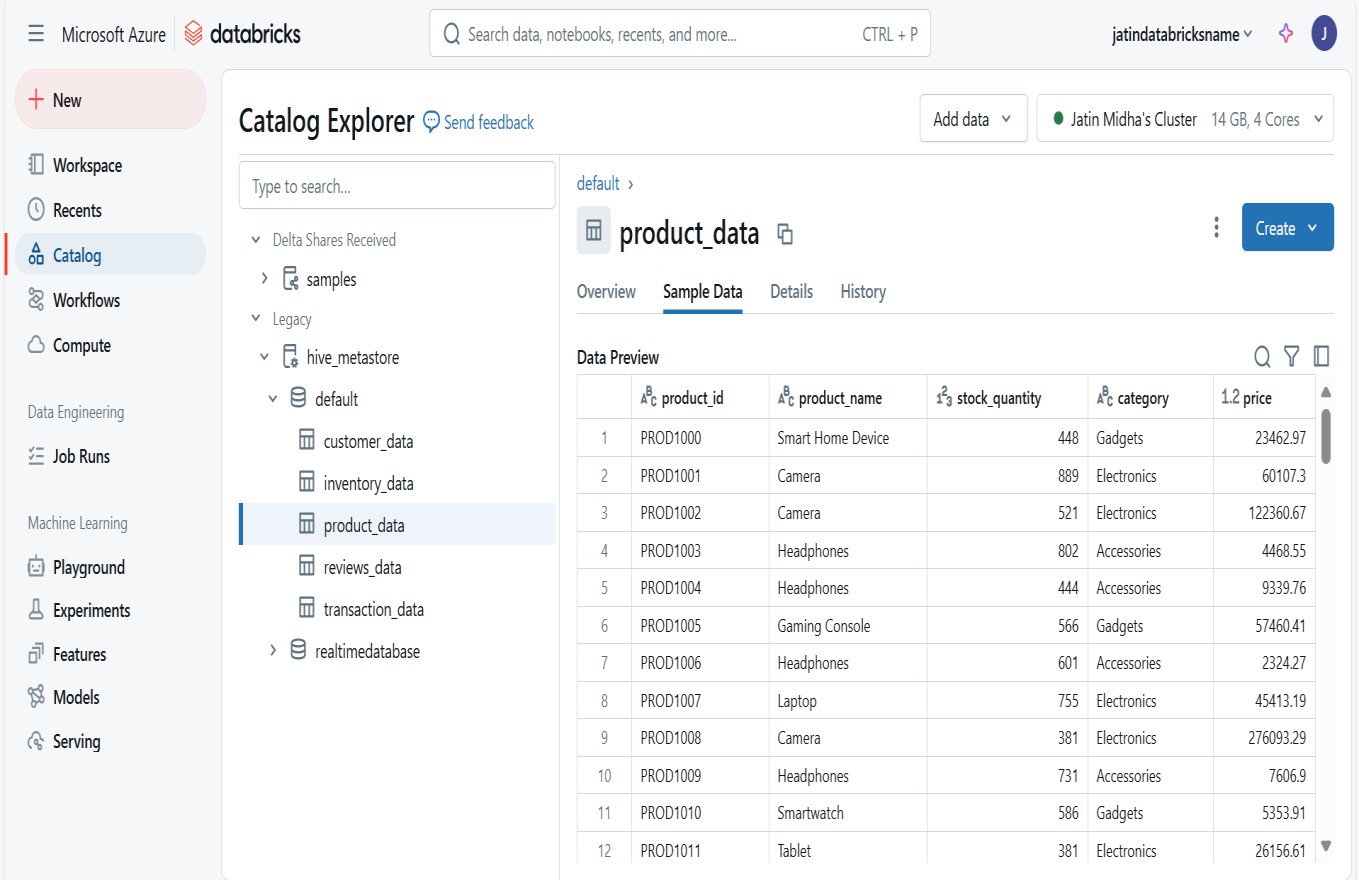
### Databricks Details

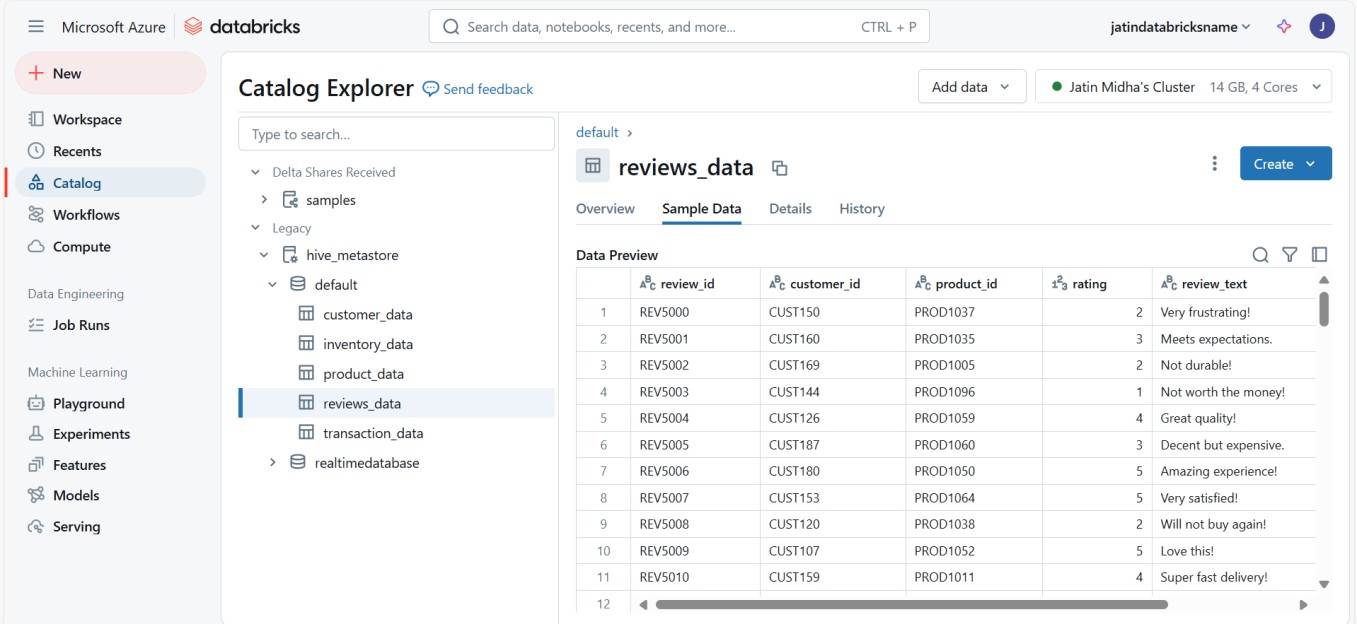


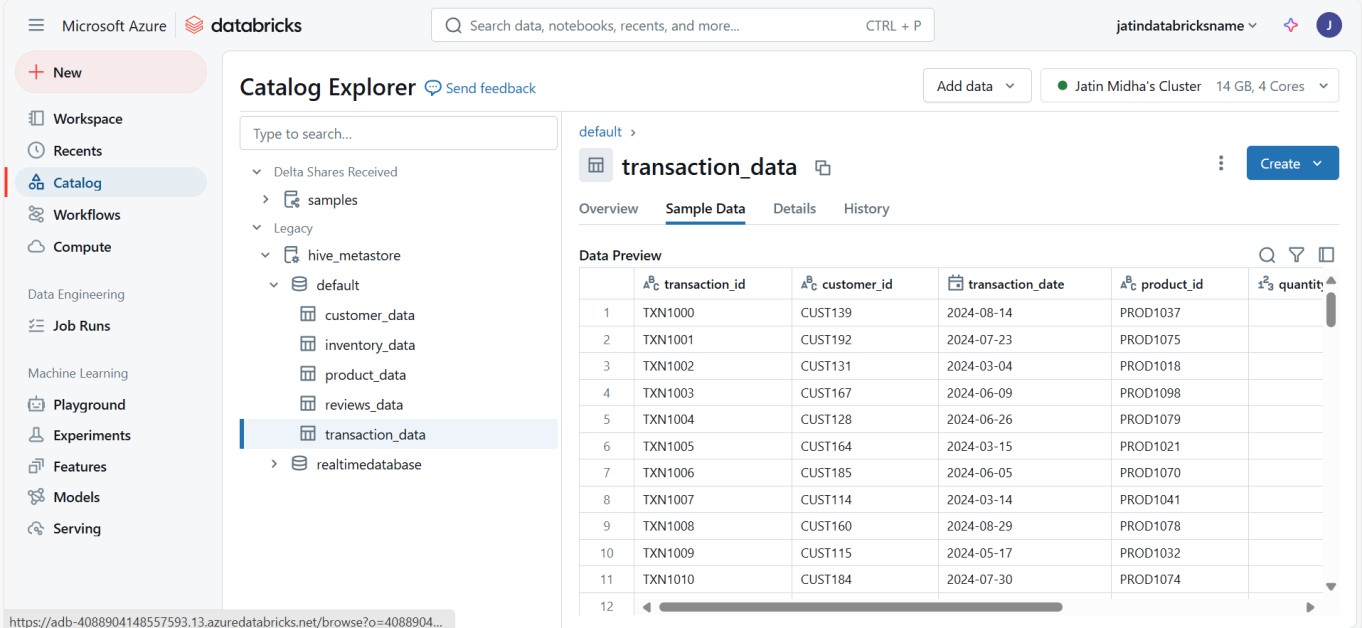






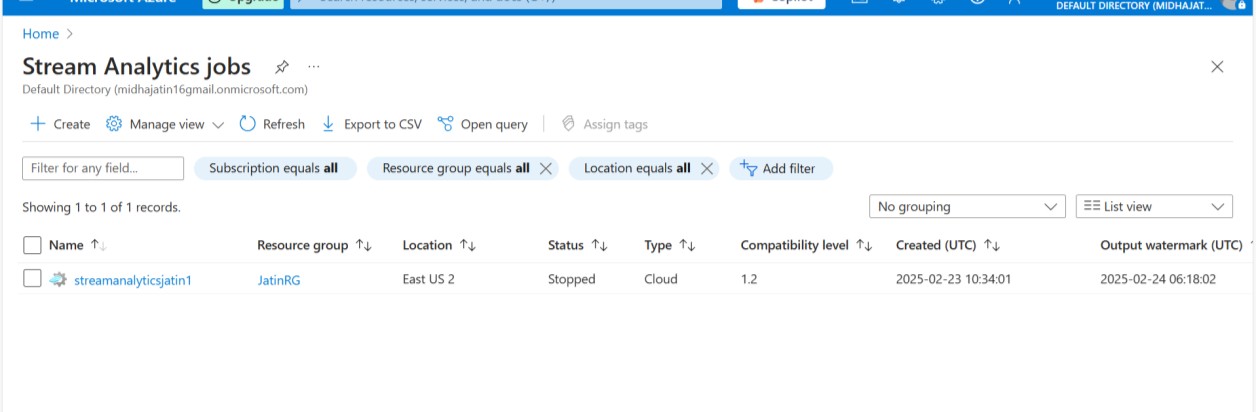


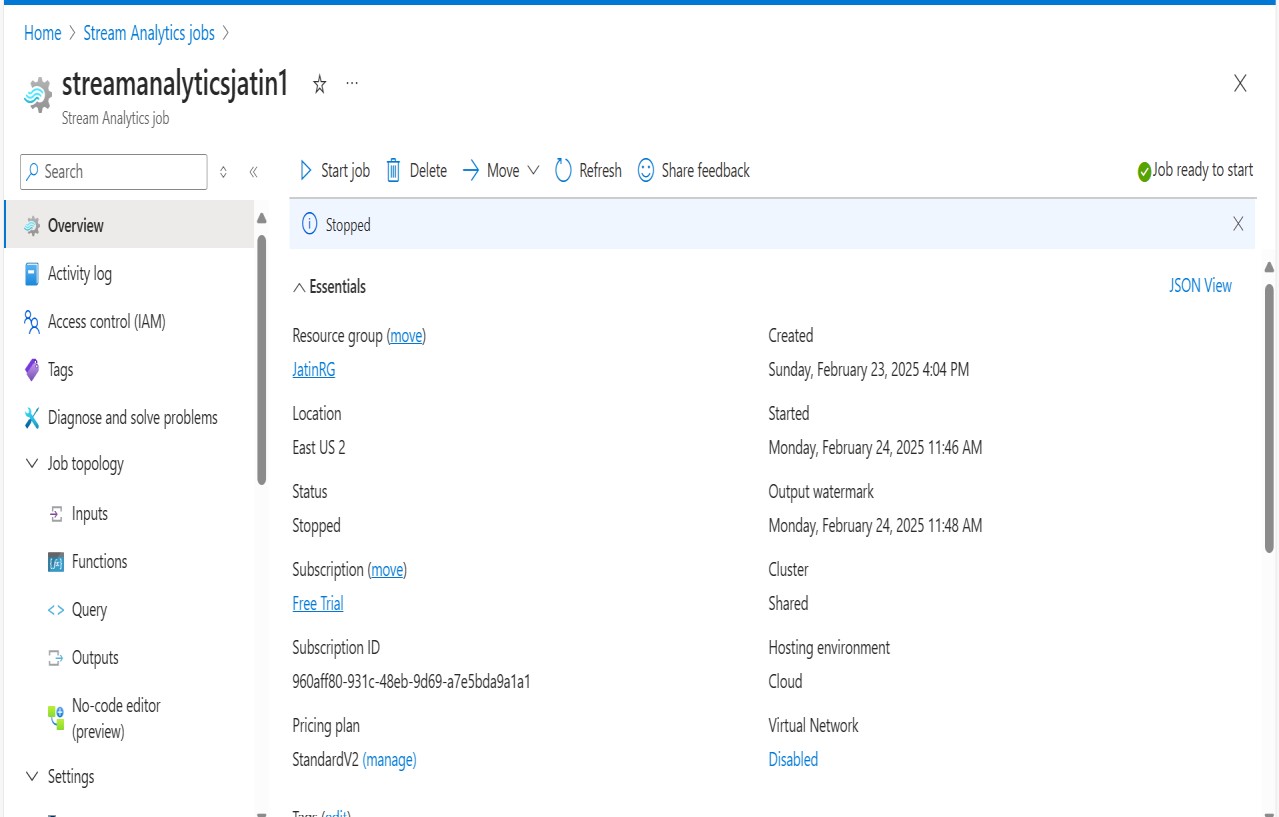




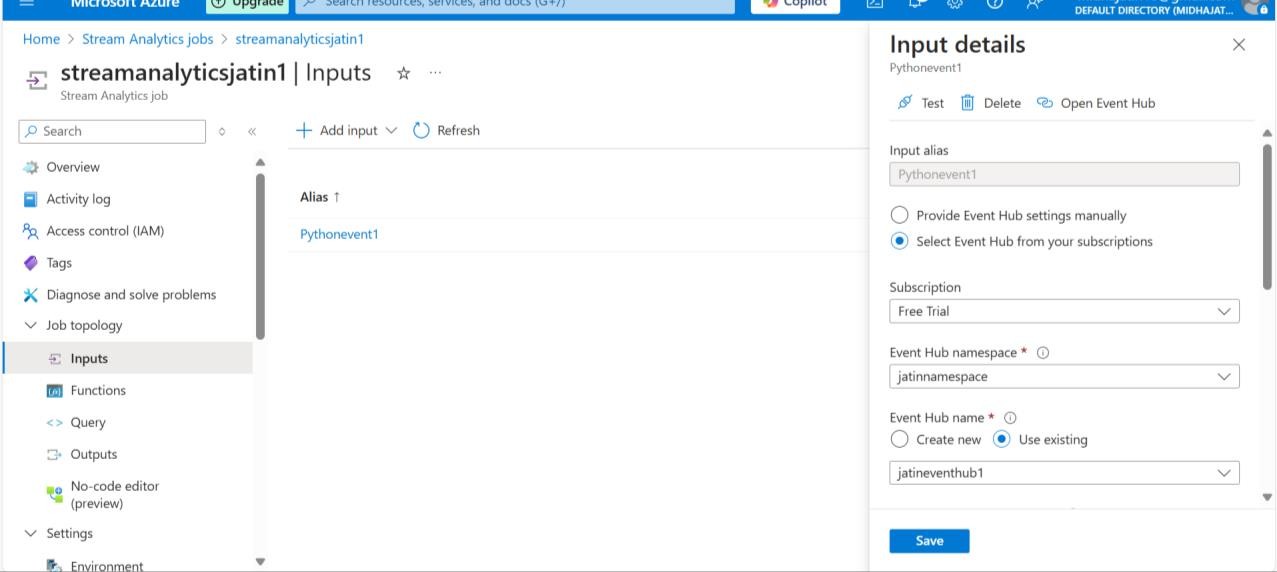
### For Real-Time Data

Initializing Stream Analytics Job

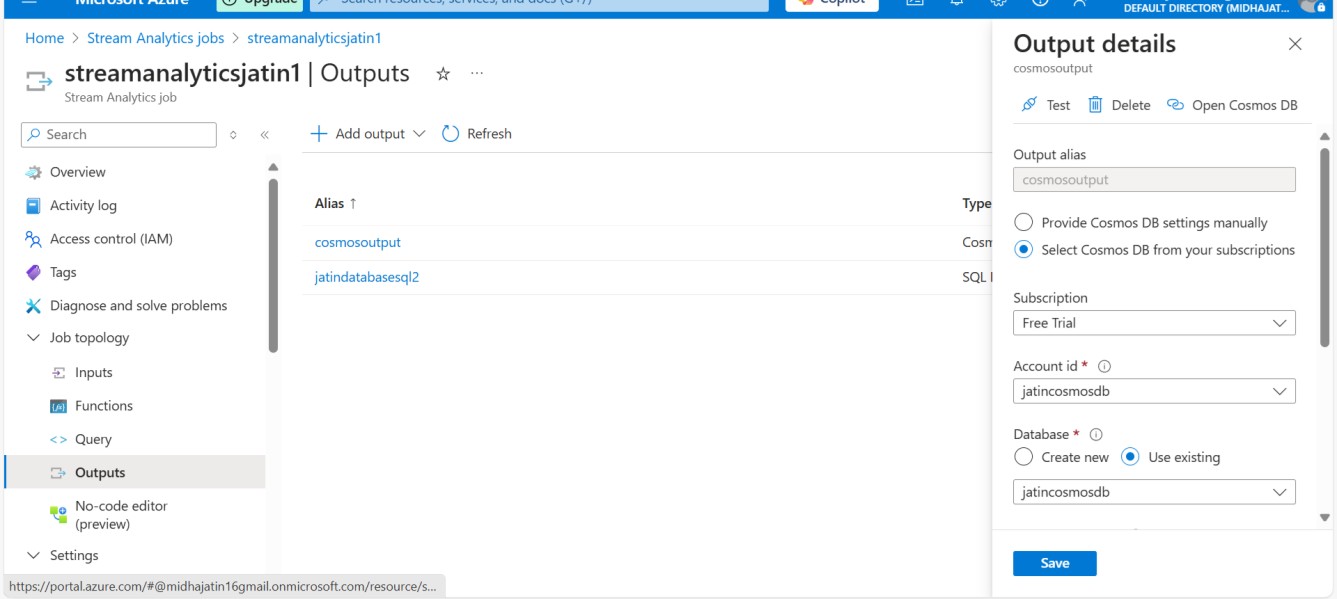




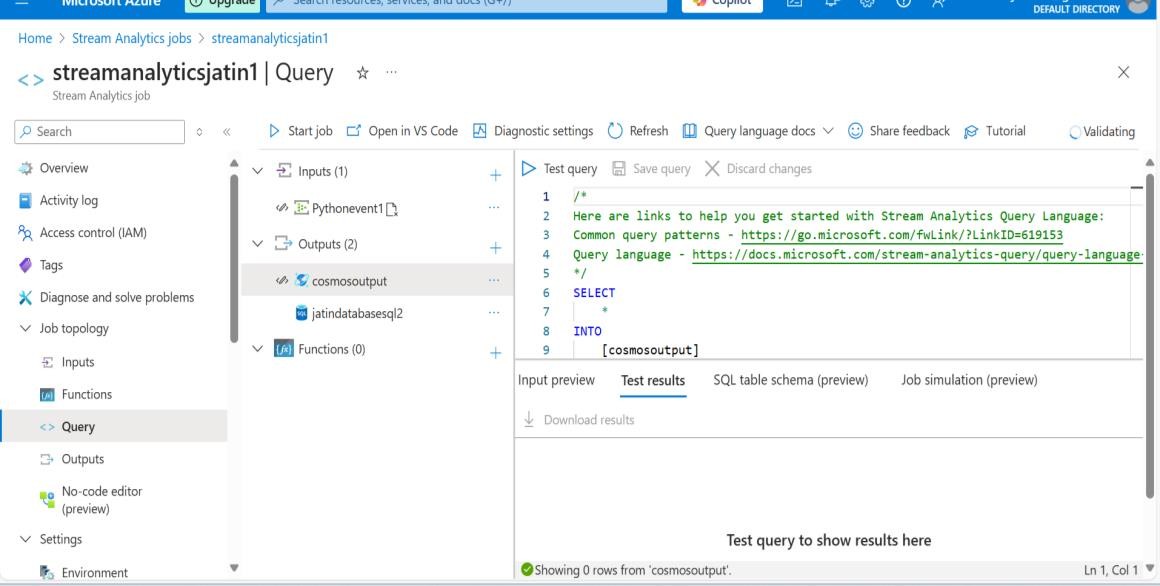
#### Setting up Input



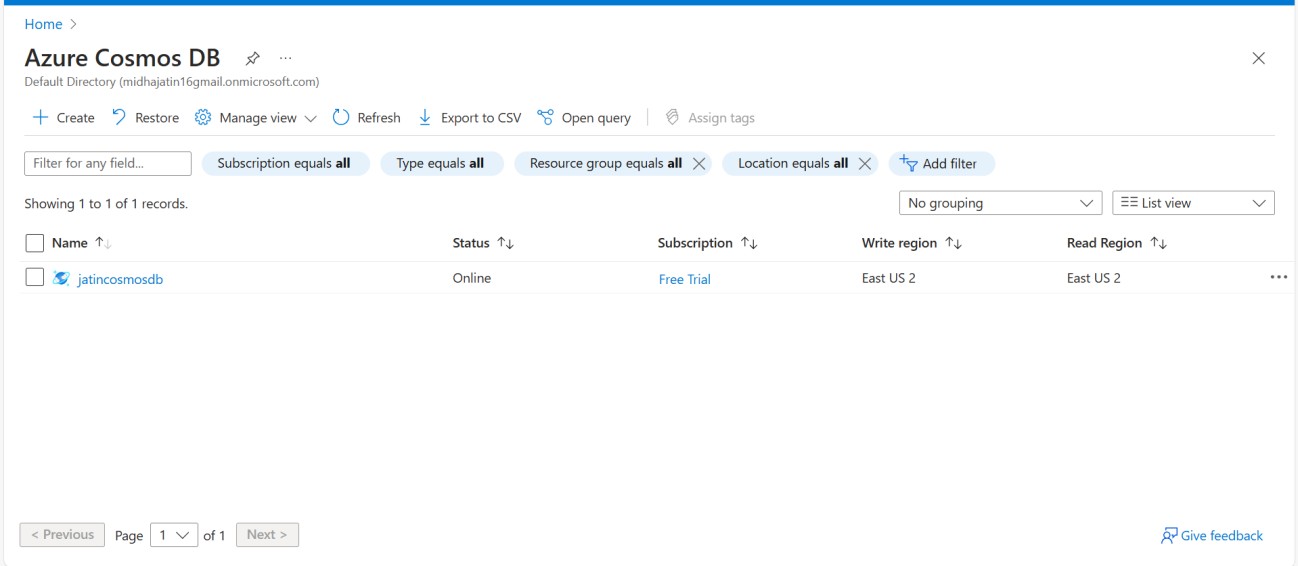
#### Setting up Output



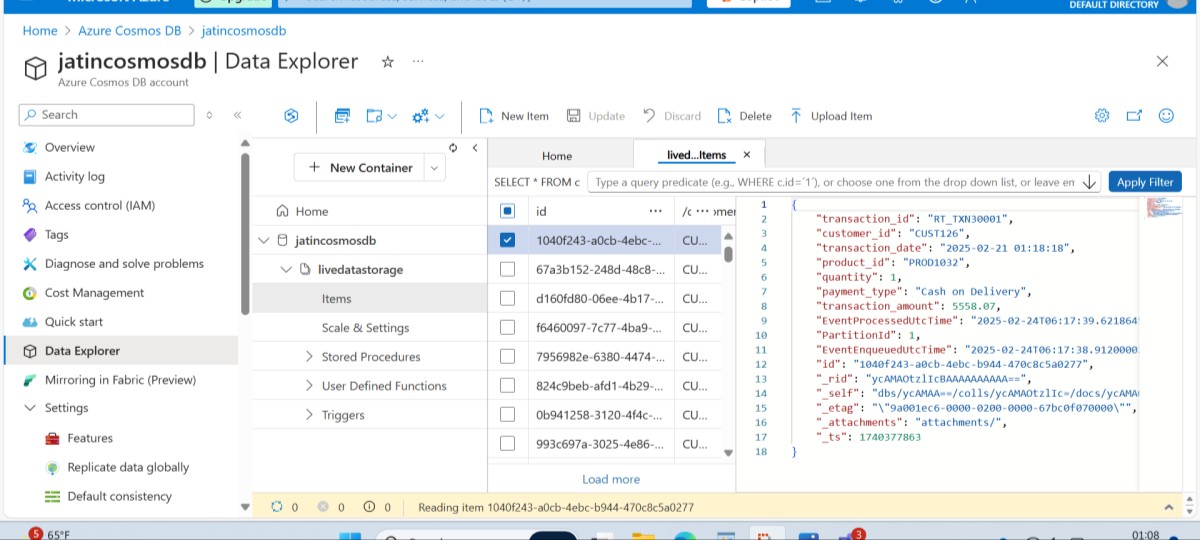
#### Using Stream Analytics Job to ingest real time data



#### Setting up Cosmos DB



#### Data is sent to Cosmos DB



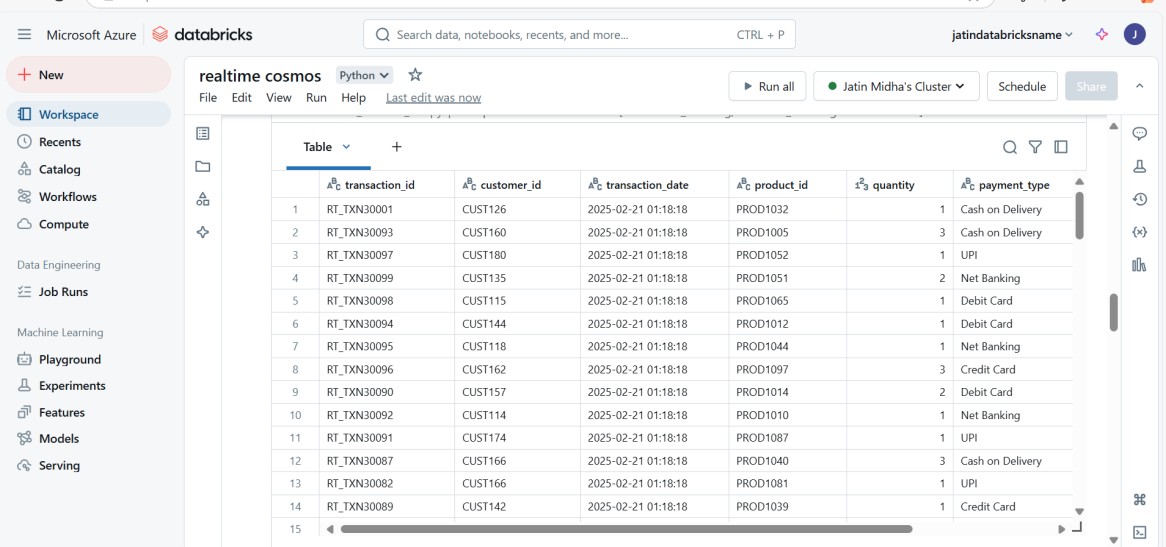
|  |  |
| --- | --- |
| from pyspark.sql.functions import col  # Connection details as variables  cosmos\_endpoint = ["https://jatincosmosdb.documents.azure.com:443/"](https://jatincosmosdb.documents.azure.com/) cosmos\_key = "IOvGwpuVgubfdwdSgHknWaUfPFU8BJYKoX9TMOkac5i7Wbv24G5MSvOy8iVt9sE2TXl10rGiI5sBACDbb69zrw==" database\_name = "jatincosmosdb" container\_name = "livedatastorage"    # Read from Cosmos DB cosmos\_df = spark.read \  .format("cosmos.oltp") \  .option("spark.cosmos.accountEndpoint", cosmos\_endpoint) \  .option("spark.cosmos.accountKey", cosmos\_key) \  .option("spark.cosmos.database", database\_name) \  .option("spark.cosmos.container", container\_name) \  .load()    # Select only required columns (remove Cosmos DB metadata) cosmos\_realtime\_df = cosmos\_df.select( col("transaction\_id"), col("customer\_id"), col("transaction\_date"), col("product\_id"), col("quantity"), col("payment\_type"), col("transaction\_amount")  )    # Show data  display(cosmos\_realtime\_df) | |
|  |

%sql

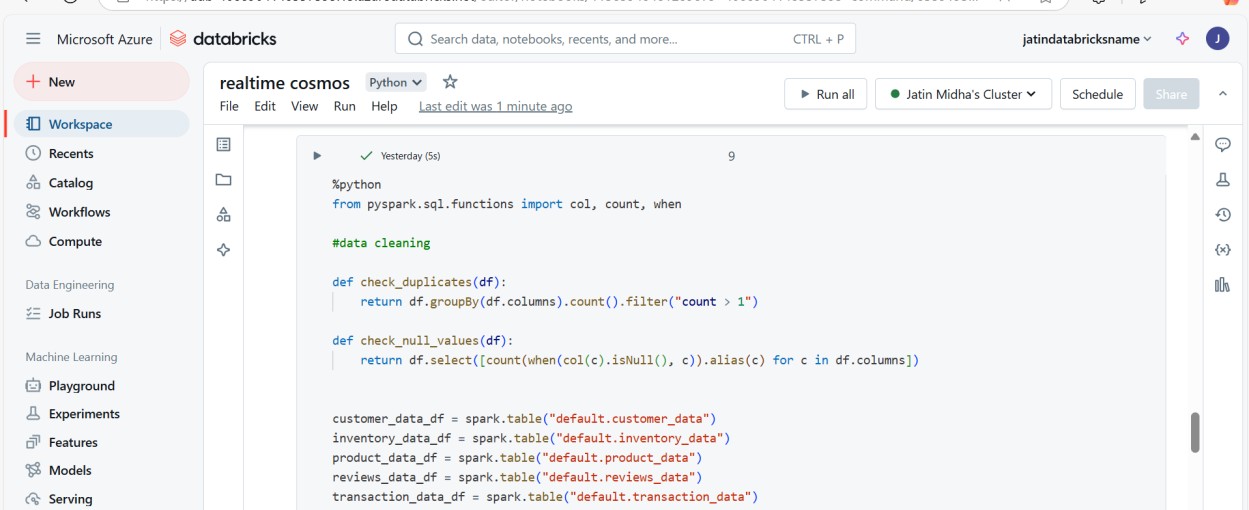
create database if not exists realtimedatabase;

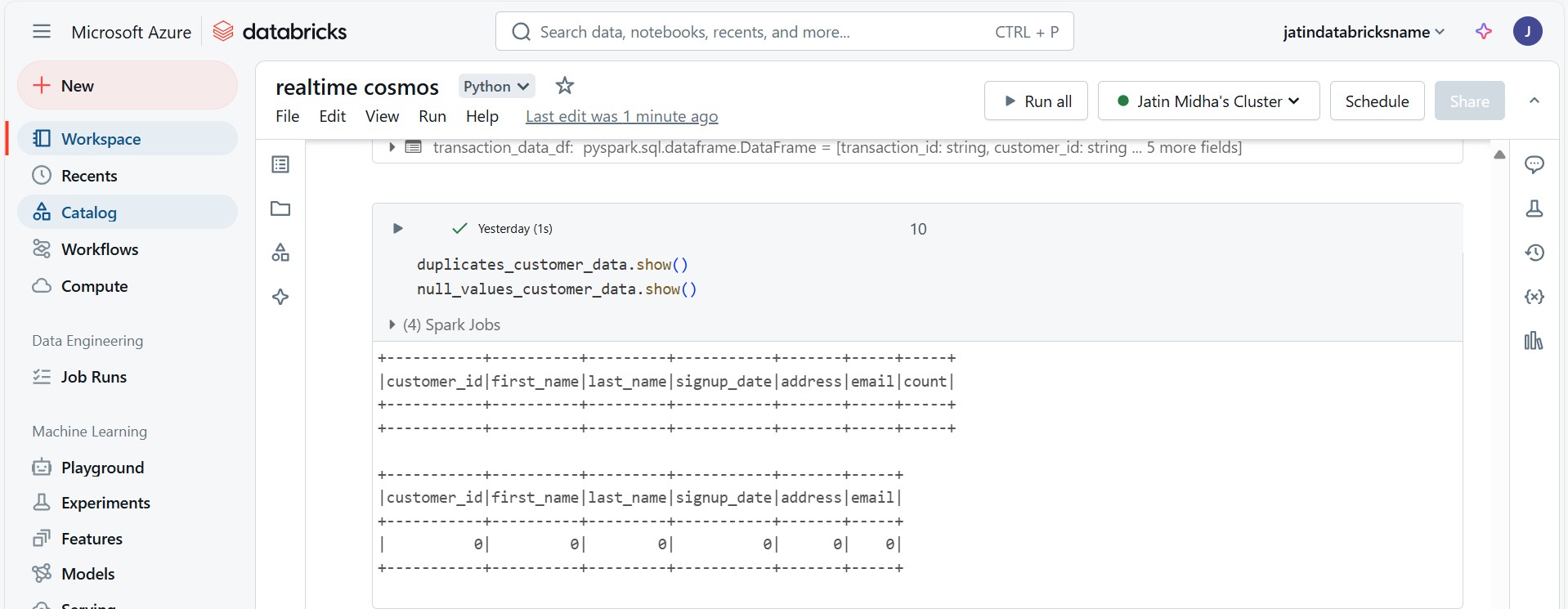
|  |
| --- |
| # Save as a Delta table in Hive Metastore database\_name = "realtimedatabase" table\_name = "real\_time\_data\_table" path = f'dbfs:/user/hive/warehouse/realtimedatabase.db/{table\_name}' cosmos\_realtime\_df.write.format("delta").mode("overwrite").option("path", path).saveAsTable(f'{database\_name}.{table\_name}')  # Print confirmation message print(f" Data successfully saved as a Delta table: {table\_name}") |

#### Data loaded from Cosmos to Databricks



#### Data checked for cleaning





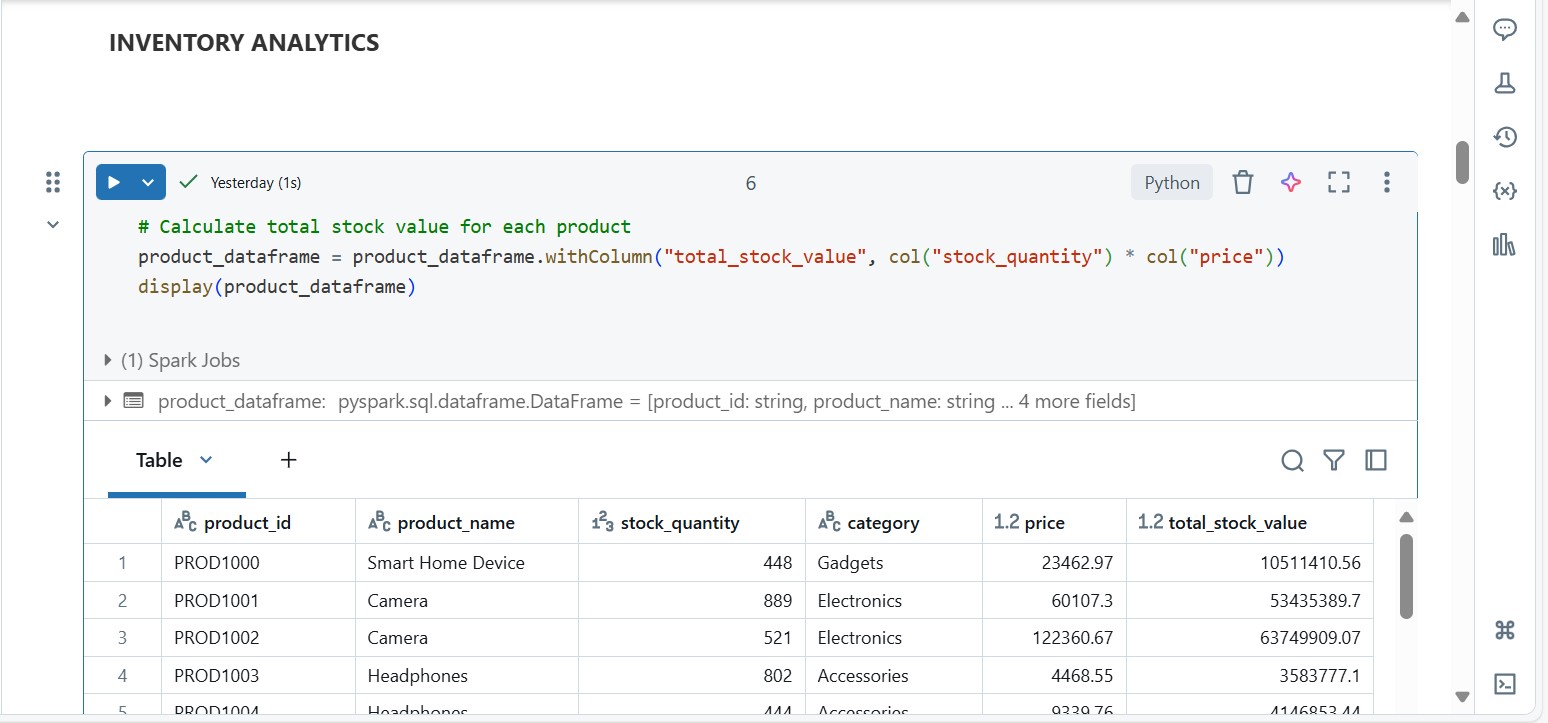
### Pyspark Transformations

**For Historical Data:**

from pyspark.sql import SparkSession

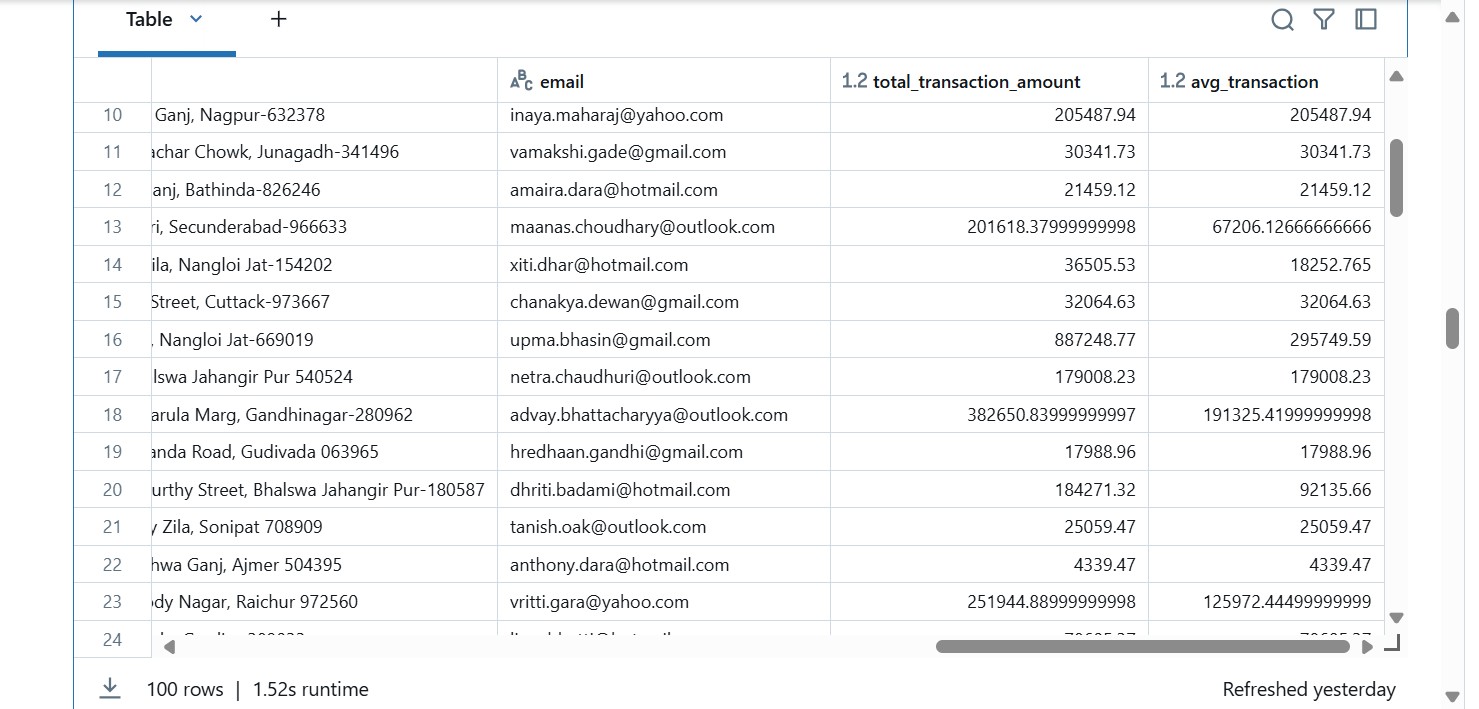
spark=SparkSession.builder.appName('Historical Data Transformations').getOrCreate() product\_dataframe=spark.sql("Select \* from default.product\_data")

|  |  |
| --- | --- |
| #transformations    # Calculate average price per category |  |
| from pyspark.sql.functions import col, sum, avg  avg\_price\_per\_category = product\_dataframe.groupBy("category").agg(avg("price").alias("average\_price"))  # Calculate total stock value for each product  product\_dataframe = product\_dataframe.withColumn("total\_stock\_value", col("stock\_quantity") \* col("price"))  display(product\_dataframe) | |

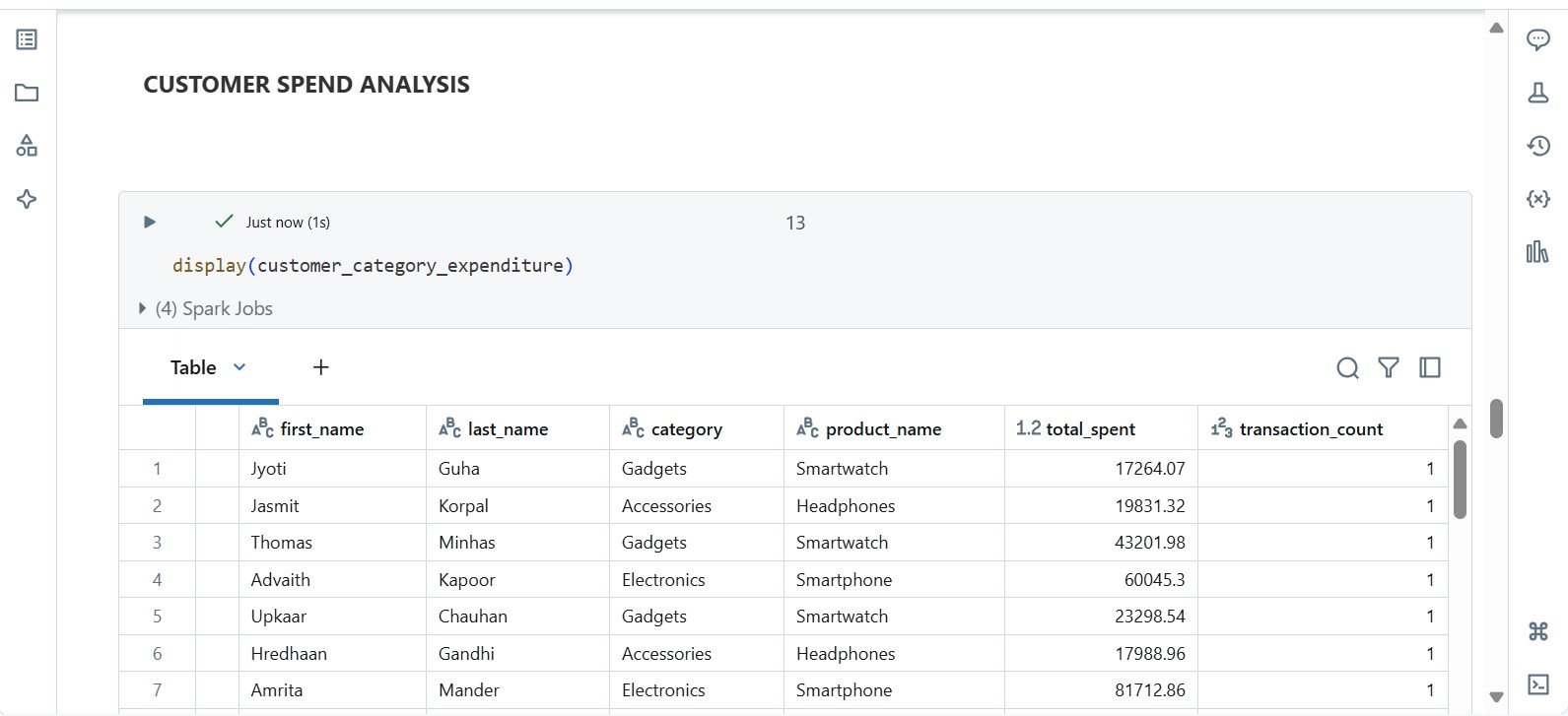


|  |
| --- |
| from pyspark.sql.functions import sum as spark\_sum    # Join the DataFrames on customer\_id joined\_df = transactions\_dataframe.join(customer\_dataframe, on="customer\_id", how="inner")    # Calculate total transaction amount for each customer customer\_total\_amount =  joined\_df.groupBy("customer\_id").agg(spark\_sum("transaction\_amount").alias("total\_transaction\_amount"))    customer\_spending\_df = joined\_df.groupBy("customer\_id") \  .agg(avg("transaction\_amount").alias("avg\_transaction")  )    joined\_df = joined\_df.join(customer\_total\_amount, on="customer\_id", how="inner")    joined\_df= joined\_df.join(customer\_spending\_df, on="customer\_id", how="inner")    # Drop the second 'transaction\_amount' column  #joined\_df = joined\_df.dropDuplicates() |

display(joined\_df)



|  |
| --- |
| from pyspark.sql import SparkSession  from pyspark.sql.functions import sum, count, avg  # Join transaction and product data joined\_df2 = transactions\_dataframe.join(product\_dataframe, on="product\_id")  # Join result with customer data joined\_df2 = joined\_df2.join(customer\_dataframe, on="customer\_id")  customer\_category\_expenditure = joined\_df2.groupBy("customer\_id", "first\_name", "last\_name", "category", "product\_name") \  .agg( sum("transaction\_amount").alias("total\_spent"), count("transaction\_id").alias("transaction\_count") ) |



|  |
| --- |
| *# Install the necessary library*  *%pip install textblob*    *from textblob import TextBlob from pyspark.sql.functions import udf from pyspark.sql.types import StringType*  *reviews\_dataframe = spark.sql("SELECT \* FROM default.reviews\_data")*    *# Define a function to analyze sentiment def analyze\_sentiment(review): analysis = TextBlob(review) if analysis.sentiment.polarity > 0:*  *return 'Positive' elif analysis.sentiment.polarity == 0:*  *return 'Neutral' else:*  *return 'Negative'*    *# Register the UDF (User Defined Function) sentiment\_udf = udf(analyze\_sentiment, StringType())*    *# Apply the sentiment analysis function to the review text reviews\_df2 = reviews\_dataframe.withColumn("sentiment", sentiment\_udf(reviews\_dataframe["review\_text"]))*    *# Display the results*  display(reviews\_df2) |

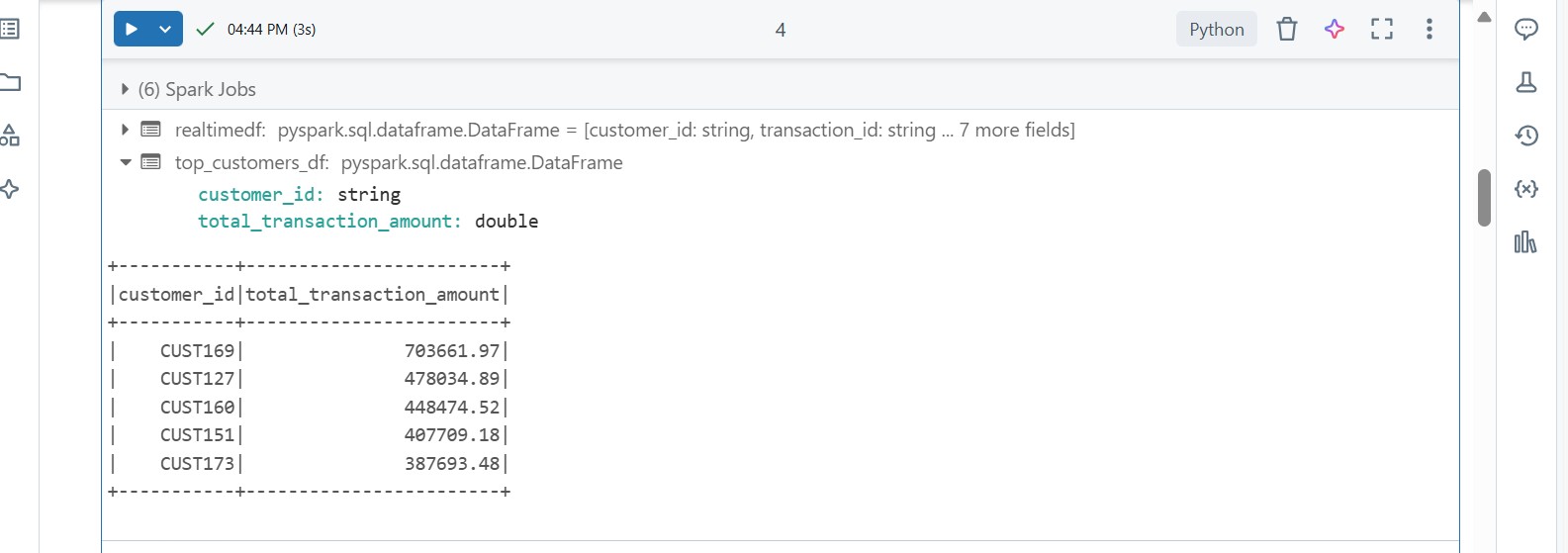
|  |
| --- |
| jdbc\_url = "jdbc:sqlserver://jatinsql1.database.windows.net:1433;databaseName=jatindatabasesql2 " db\_properties = {  "user": "jatin1",  "password": "Qwertyuiop12",  "driver": "com.microsoft.sqlserver.jdbc.SQLServerDriver"  }  joined\_df.write \  .format("jdbc") \  .option("url", jdbc\_url) \  .option("dbtable", "dbo.total\_average\_transactions") \  .option("user", db\_properties["user"]) \  .option("password", db\_properties["password"]) \  .option("driver", db\_properties["driver"]) \  .mode("overwrite") \  .save() |

**For Real Time Data**

#### Showing top customers by transaction amount

from pyspark.sql import SparkSession

|  |  |
| --- | --- |
| spark=SparkSession.builder.appName("Realtime\_trans").getOrCreate() realtimedf= spark.sql('select \* from realtimedatabase.real\_time\_data\_table') |  |
| from pyspark.sql.functions import sum total\_revenue\_df = realtimedf.agg(sum("transaction\_amount").alias("total\_revenue")) total\_revenue\_df.show()  #showing top customers by transactions amount top\_customers\_df = realtimedf.groupBy("customer\_id").agg(sum("transaction\_amount").alias("total\_transaction\_amount")) top\_customers\_df = top\_customers\_df.orderBy("total\_transaction\_amount", ascending=False).limit(5) top\_customers\_df.show()  realtimedf=realtimedf.join(top\_customers\_df, "customer\_id", "left")    jdbc\_url = "jdbc:sqlserver://jatinsql1.database.windows.net:1433;databaseName=jatindatabasesql2 " db\_properties = {  "user": "jatin1",  "password": "Qwertyuiop12", # Avoid storing passwords in code  "driver": "com.microsoft.sqlserver.jdbc.SQLServerDriver"  }  top\_customers\_df.write \  .format("jdbc") \  .option("url", jdbc\_url) \  .option("dbtable", "dbo.top\_customers\_shorttable\_realtime") \  .option("user", db\_properties["user"]) \  .option("password", db\_properties["password"]) \  .option("driver", db\_properties["driver"]) \  .mode("overwrite") \  .save()  #top 5 customers with high amount transaction in realtime data display(top\_customers\_df) | |



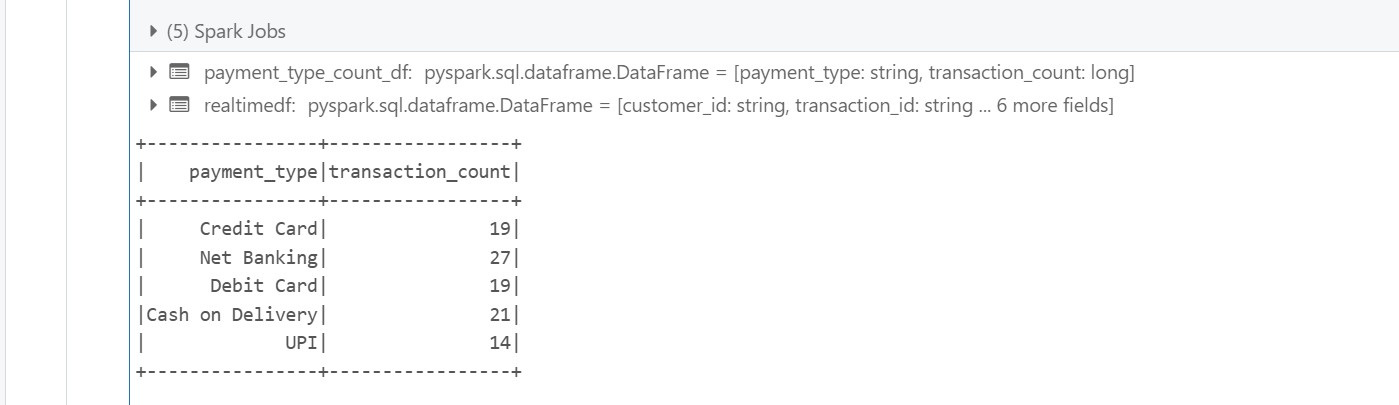
|  |
| --- |
| jdbc\_url = "jdbc:sqlserver://jatinsql1.database.windows.net:1433;databaseName=jatindatabasesql2 " db\_properties = {  "user": "jatin1",  "password": "Qwertyuiop12", # Avoid storing passwords in code  "driver": "com.microsoft.sqlserver.jdbc.SQLServerDriver"  }  realtimedf.write \  .format("jdbc") \  .option("url", jdbc\_url) \  .option("dbtable", "dbo.top\_customers\_realtime") \  .option("user", db\_properties["user"]) \  .option("password", db\_properties["password"]) \  .option("driver", db\_properties["driver"]) \  .mode("overwrite") \  .save() |

from pyspark.sql.functions import count payment\_type\_count\_df = realtimedf.groupBy("payment\_type").agg(count("transaction\_id").alias("transaction\_count")) payment\_type\_count\_df.show() # Drop the 'payment\_type' column

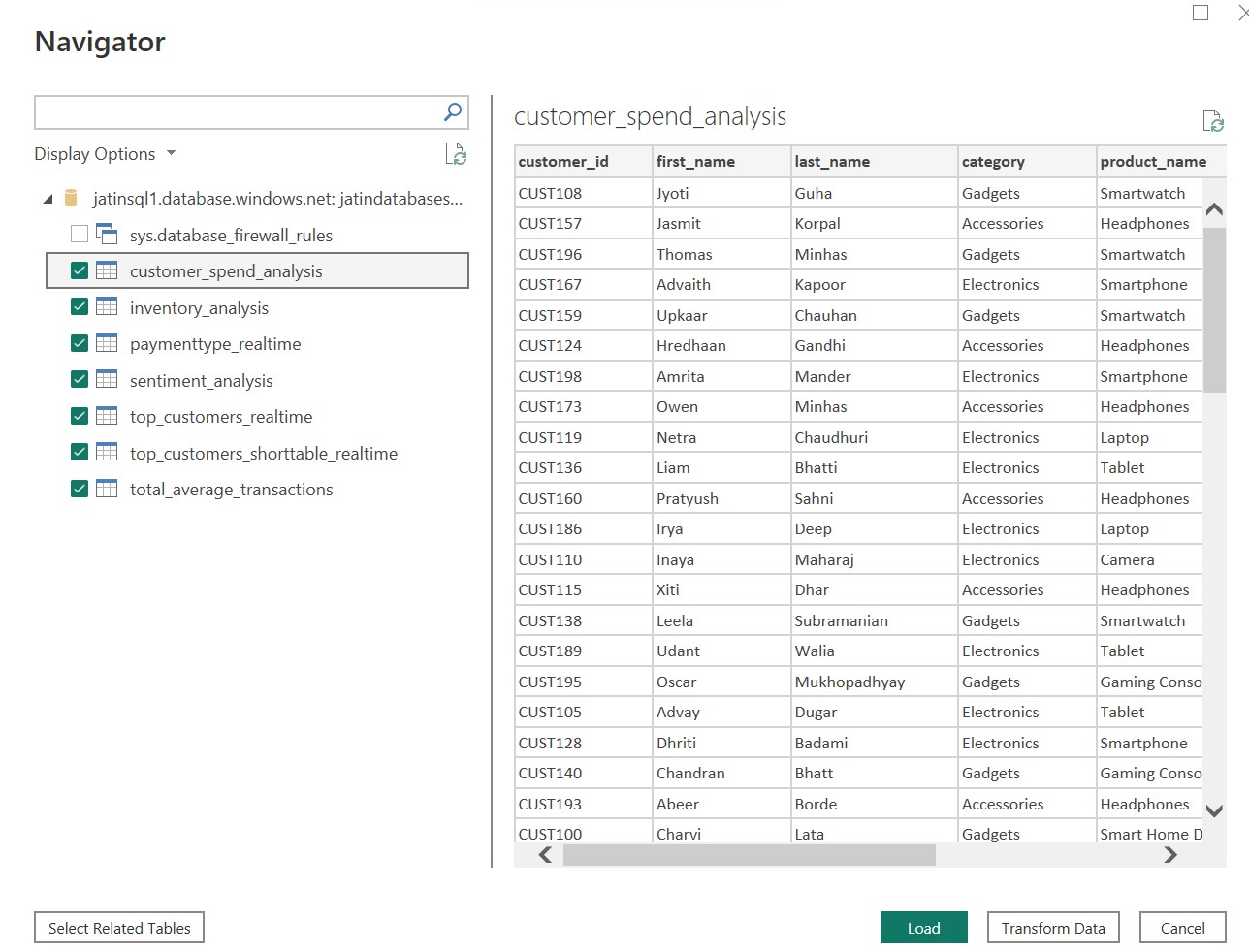
realtimedf = realtimedf.drop("total\_quantity\_sold")

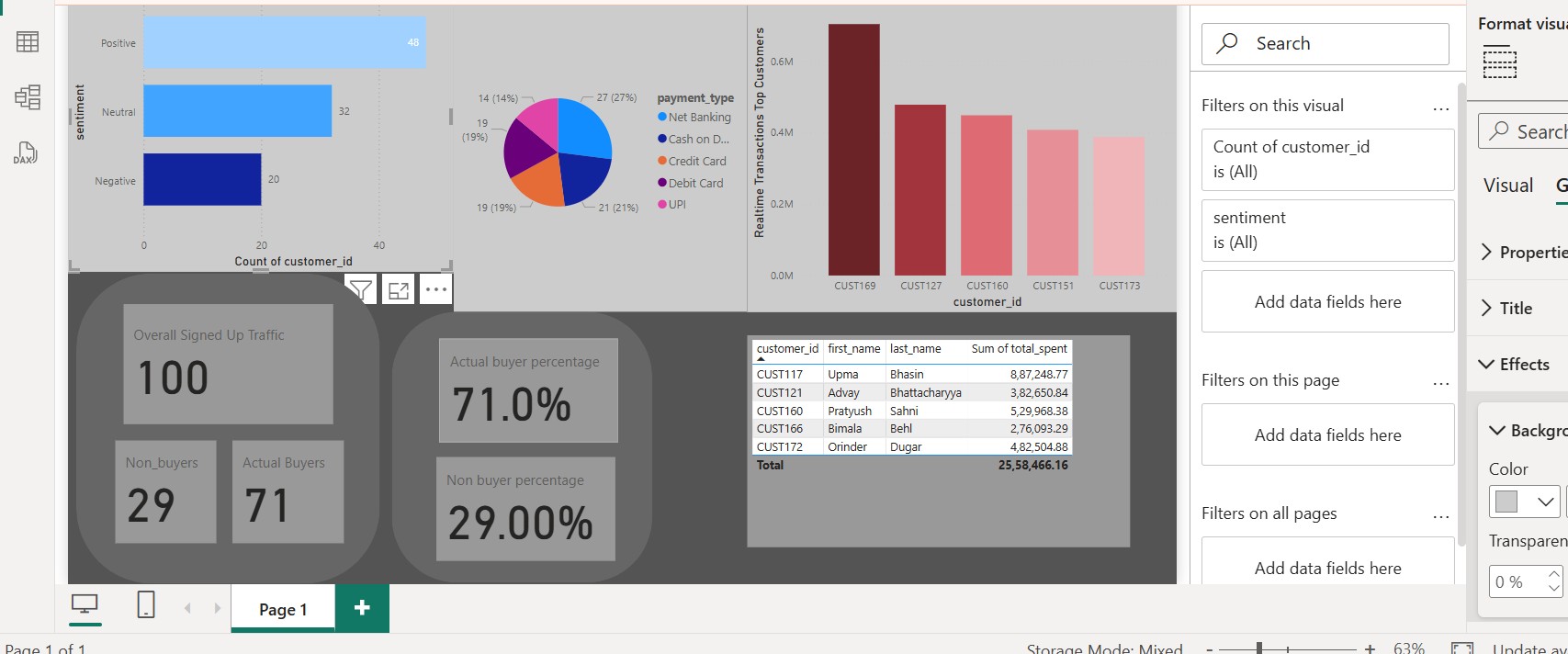
display(realtimedf)

### Transaction amount by payment type



### Connecting Power Bi to Azure Sql Database and importing tables





## Conclusion

* **Enhanced Data Management**: The project significantly improved the company's ability to manage and analyze large volumes of transactional data, ensuring high data quality and reliability.
* **Robust ETL Pipeline**: By leveraging Azure services and Databricks, a reliable ETL pipeline was established for both historical and real-time data, facilitating seamless data ingestion, transformation, and storage.
* **Valuable Business Insights**: The solution provided actionable insights, such as optimized inventory management, improved customer retention, and enhanced marketing ROI, driving strategic decisionmaking.
* **Real-Time Analytics**: The integration of real-time analytics enabled immediate detection of high-value transactions and potential fraud, enhancing operational efficiency and security.
* **Scalability**: The solution was designed to accommodate future growth and technological advancements, ensuring long-term viability and adaptability.

## Future Work

**Advanced Analytics and Machine Learning**: Develop and deploy more sophisticated machine learning models for predictive analytics, such as demand forecasting and customer behavior analysis, to enhance decision-making.

* **Global Data Distribution**: Expand the data infrastructure to support global operations, ensuring lowlatency access and high availability of data across multiple regions.
* **Microservices Architecture**: Transition to a microservices architecture to enable independent scaling of different components, improving system flexibility and maintainability.
* **Enhanced Security and Compliance**: Continuously update security protocols and conduct regular audits to ensure compliance with evolving regulations and protect sensitive customer data.
* **Integration with Emerging Technologies**: Explore and adopt new technologies, such as edge computing and AI-driven automation, to enhance system capabilities and maintain a competitive edge.