
Business Requirement:

A company XYZ, operating in the **Cycling and Outdoor Sports Retail** sector, aims to gain deeper insights into customer demographics and their purchasing patterns. Specifically, stakeholders are interested in understanding the **gender distribution of customers** and its impact on sales across various product categories. The company's product portfolio includes a diverse range of cycling-related items, from **bikes and accessories** to **clothing and components**.

Key requirements include:

1. **KPI Dashboard:**
    - Insights into **sales by gender** and **product category**, highlighting:
        - Total products sold.
        - Total sales revenue.
        - Gender split of customers.
2. **Dynamic Filtering:**
    - Allow stakeholders to filter data by **product category**, **gender**, and **date range**.
3. **Real-time Access:**
    - Provide up-to-date data through an automated and scalable pipeline.

The company's existing data resides in an on-premises SQL database, requiring a robust and efficient system to process, analyze, and visualize the information.

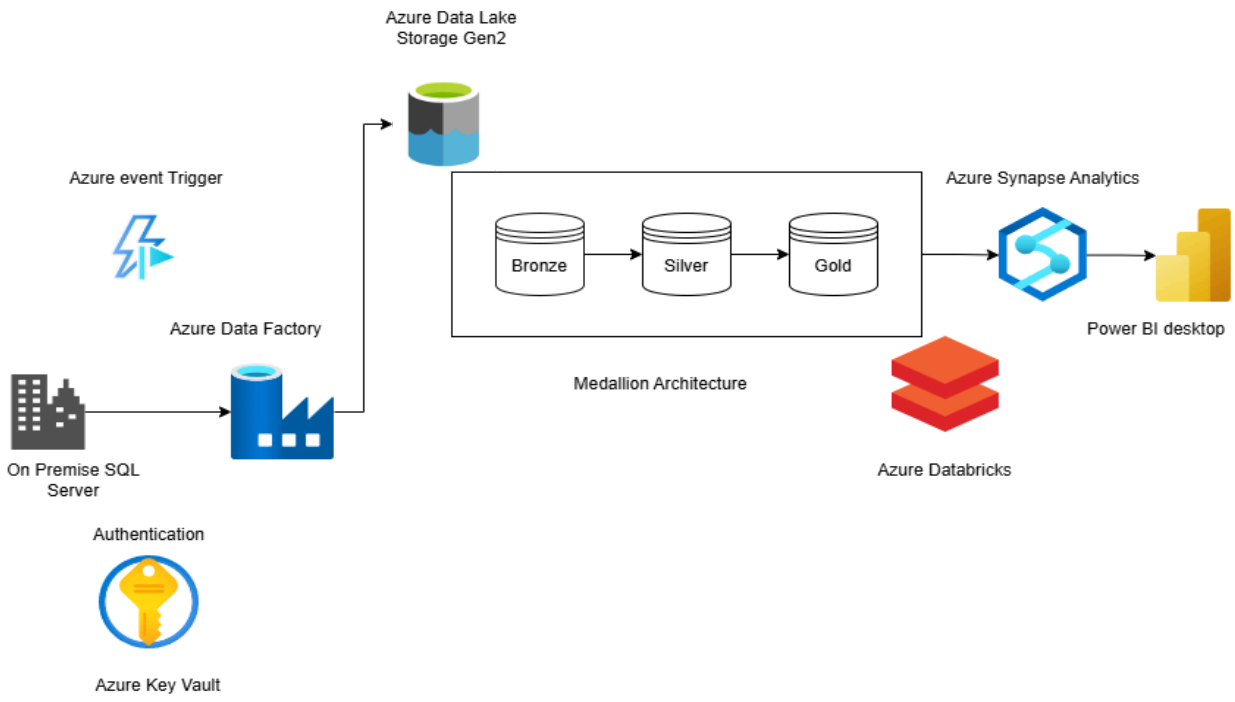
Solution:

To address these business requirements, I proposed the following solution:

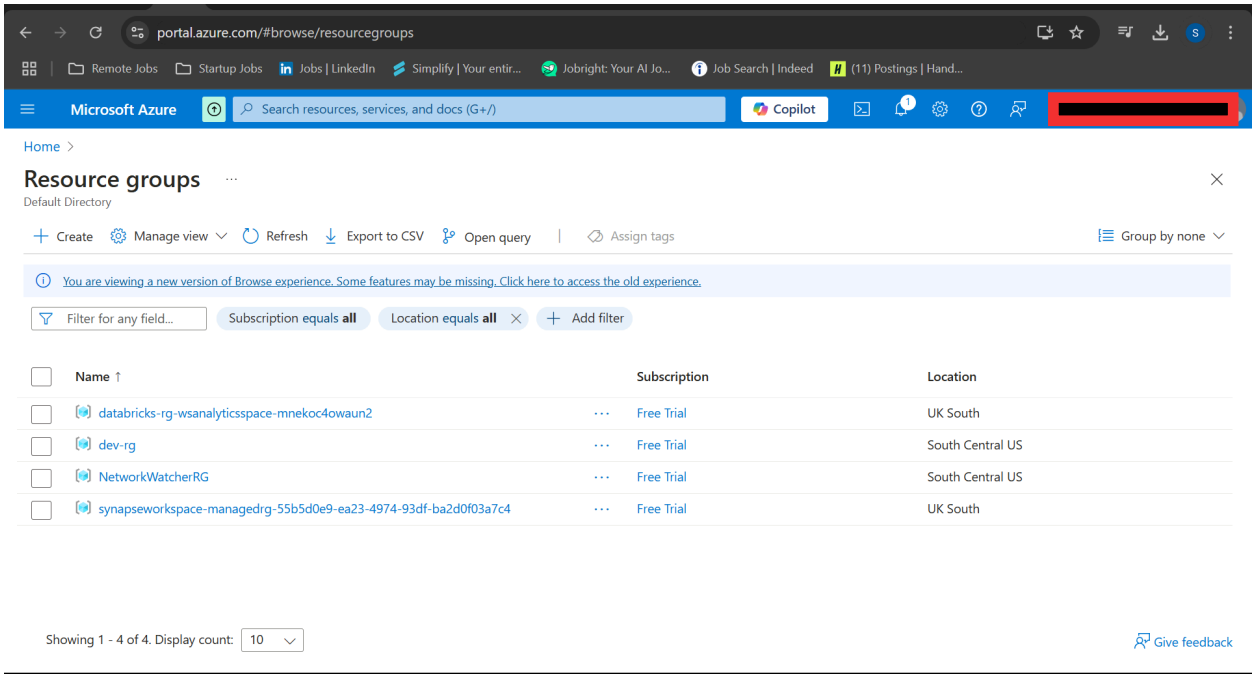
1. **Data Extraction and Ingestion:**
    - Extract sales and customer data from the on-premises SQL database using **Azure Data Factory (ADF)**.
    - Store the raw data securely in **Azure Data Lake Storage (ADLS)** for centralized processing.
2. **Data Transformation:**
    - Use **Azure Databricks** with **PySpark** to cleanse and transform the data.
    - Key transformations include:
        - Mapping product categories (e.g., **Bikes, Accessories, Clothing, Components**).
        - Aggregating KPIs such as:
            - Total products sold by gender and category.
            - Total revenue by gender and category.
            - Percentage breakdown of gender distribution.
3. **Data Modeling and Architecture:**
    - Implement the **Medallion Architecture**:

        - **Bronze Layer:** Raw data directly extracted from the SQL database.
        - **Silver Layer:** Cleansed and structured data with enriched product and customer details.
        - **Gold Layer:** Aggregated KPI data, ready for reporting and dashboarding.
4. **KPI Dashboard Development:**
    - Build a **Power BI dashboard** with:
        - Visualizations for total sales, products sold, and gender distribution.
        - Interactive slicers for filtering by **gender**, **product category**, and **date**.
        - Line charts and bar graphs showing trends over time.
        - Pie charts for gender distribution within product categories.
5. **Pipeline Automation:**
    - Schedule daily pipeline runs using **ADF triggers** to ensure real-time data updates.
6. **Governance and Security:**
    - Secure credentials using **Azure Key Vault**.
    - Manage access with **Azure Active Directory** to ensure data confidentiality and integrity.

Architecture Diagram:



I provisioned the resources in Azure as per the above architecture in a resource group. Once the resources are provisioned, I can see the below image in resource groups in azure



I created a user in the on-prem database and the user credentials will be saved in the key vault. The ADF will use the credentials in the key vault to access the on prem database.

Before creating a secret, I need to create a role that has the ability to create a secret. Then I assigned this role to myself.

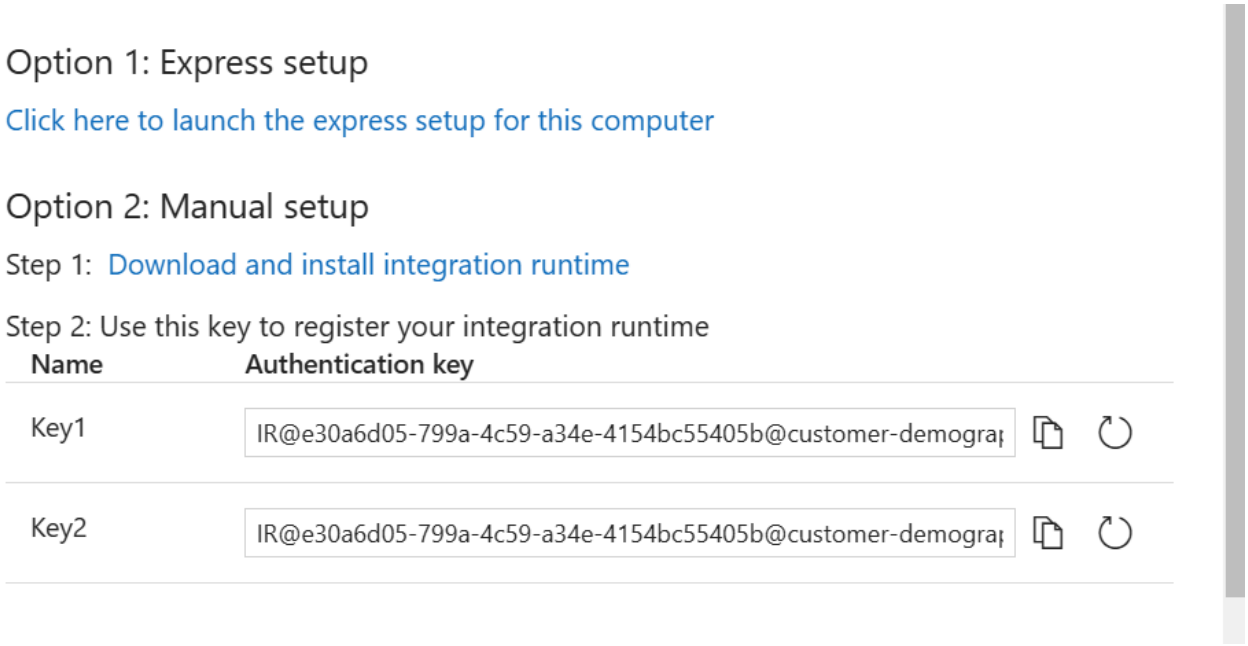
Go to IAM, click ADD, add role assignment, click on :"Azure Key Vault Administrator", click "Next". Select "User, group or service principal". Select "Members". Then I selected my username and assigned it to myself (Review and assign)

Now, go to secret, now I have the ability to create a secret.click create and then give the details of username and password and click create

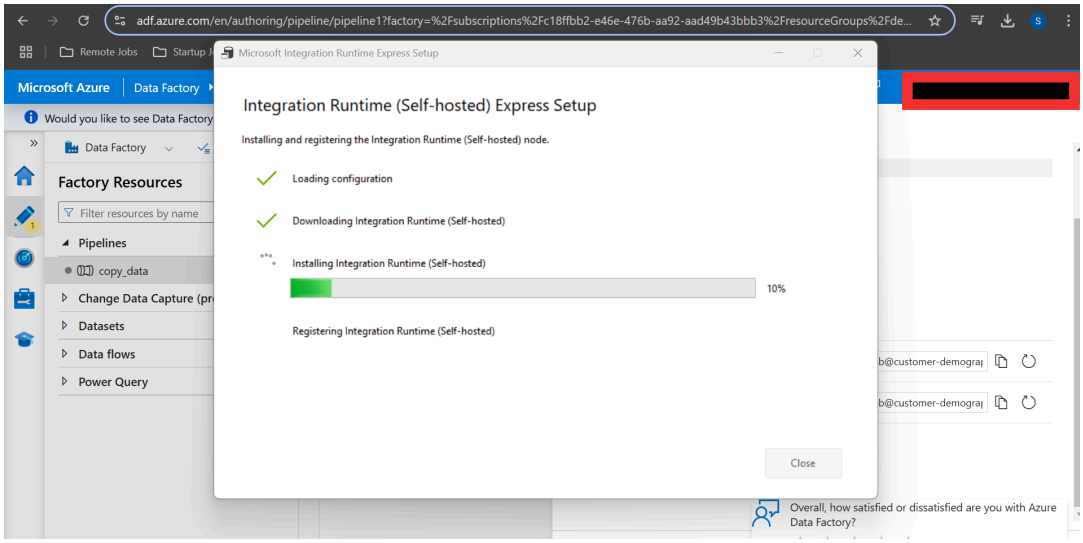
Go to Azure Data Factory, select the new pipeline, name the pipeline.
Select the copy data activity in ADF

In this case, our source is an on-prem SQL database. Go to the source section and click on "New". I was prompted to select a new data store. Select "SQL Server" from the list, Now, I was asked to select a linked service. Select "New". name the linked service

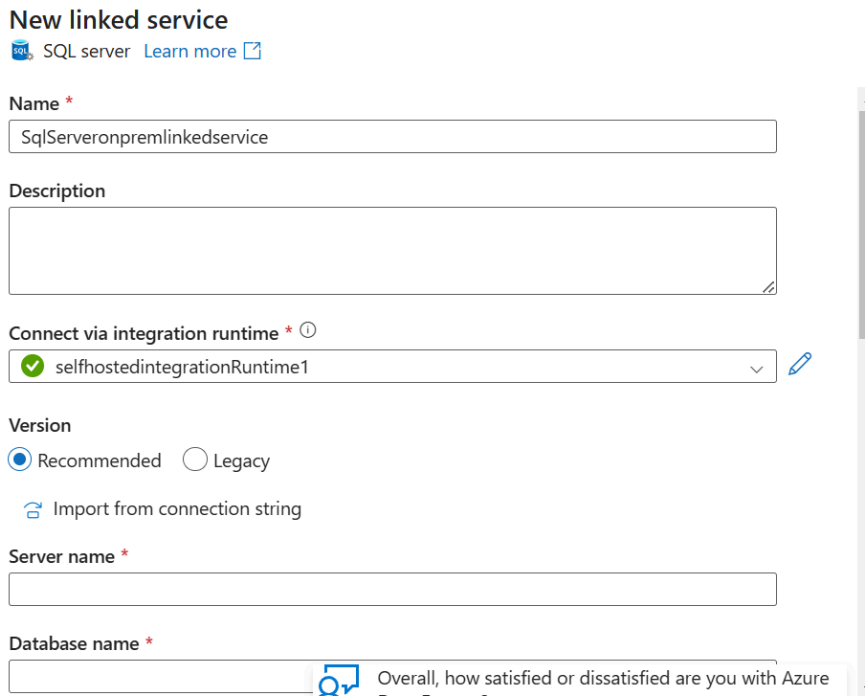
In the integration runtime section, select "New", select "Self Hosted Integration Runtime". Name the integration runtime, click "Create"



From the above options, click “Express Setup”. A program will be downloaded. Run and execute the program

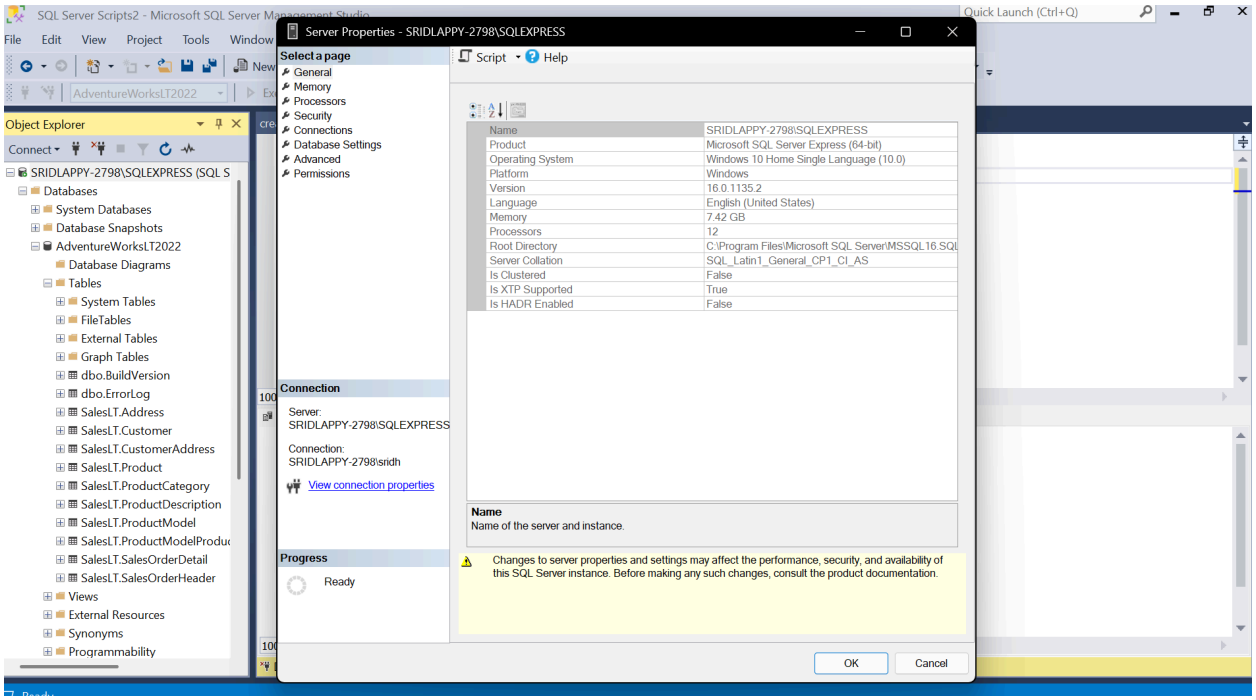


Once it is completed,I found the below screen

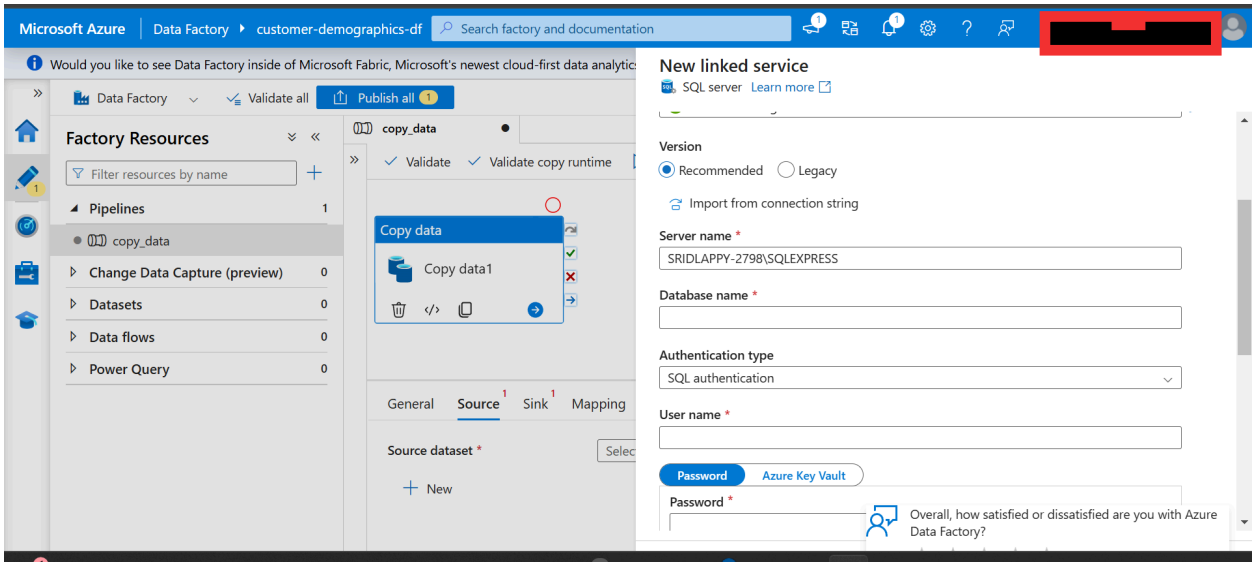


Now, I need to populate the server name and database credentials of the on premise SQL server. I can find the details from the SQL server management studio

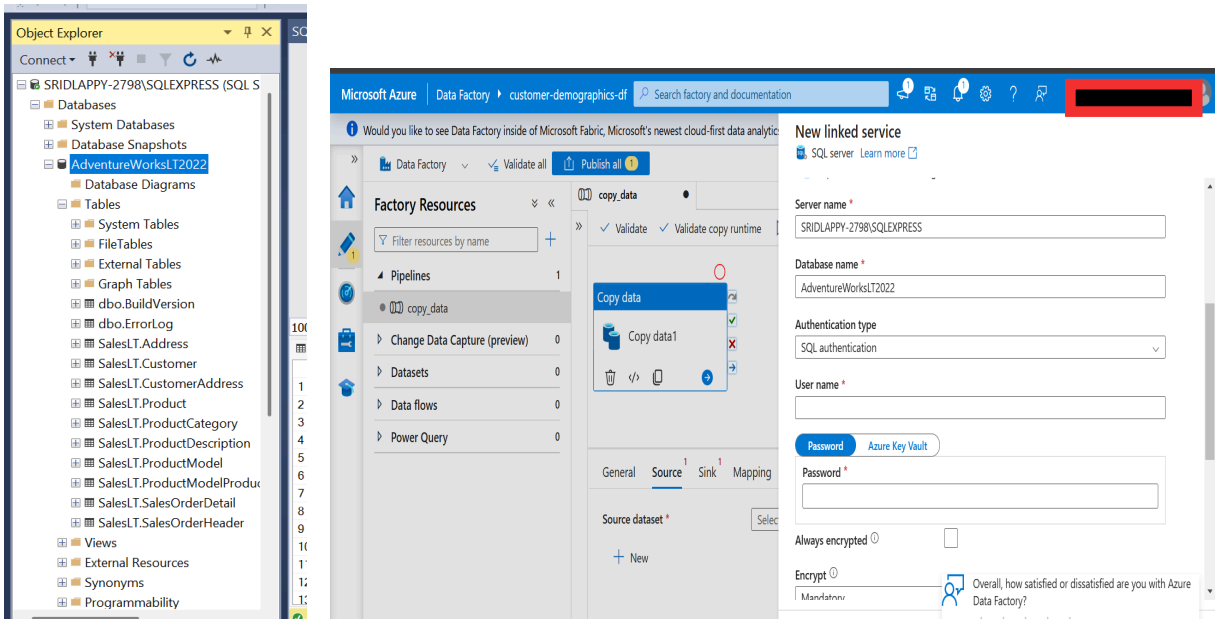
If I click on the “properties” of the database in SQL server management studio, I can find the below screen



Take the server name from the details and paste it in the “New Linked Service” as below

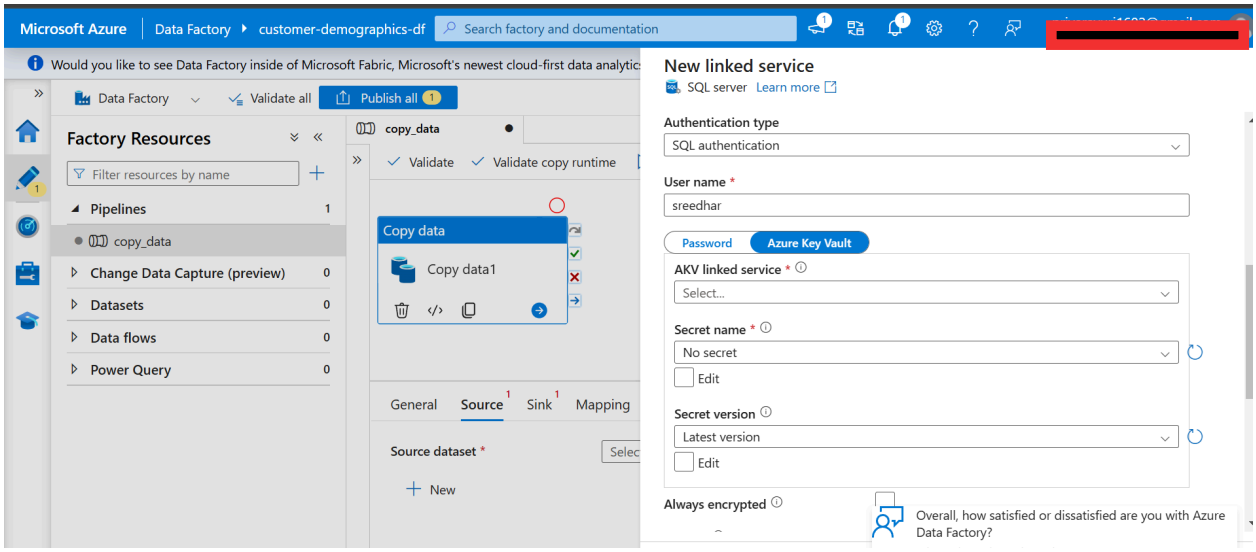


For the database name, take the name from sidebar and paste it in linked service section as below

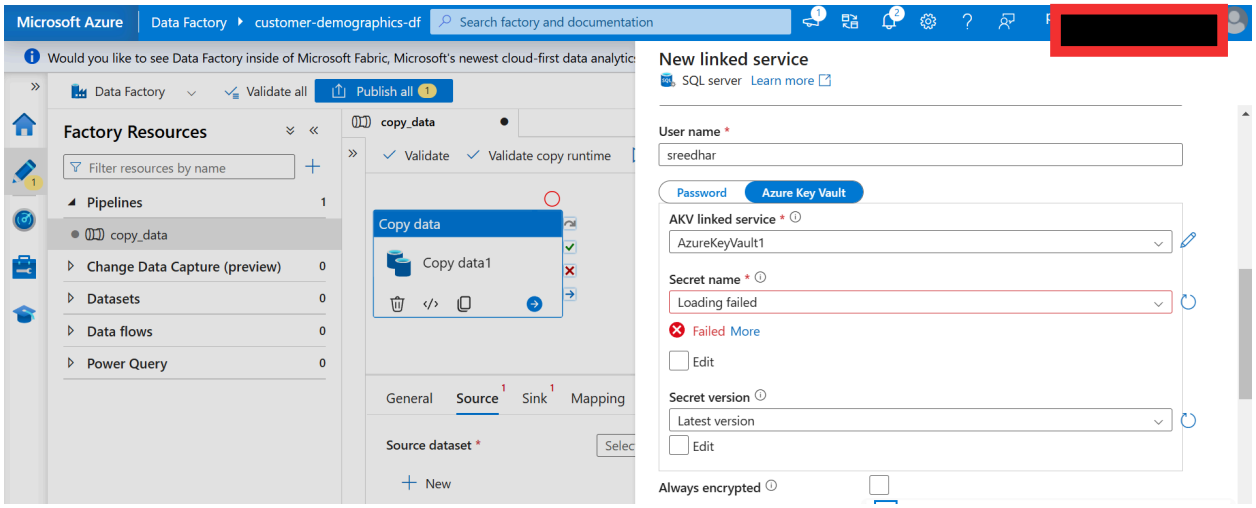


For the username , I have to give the details of the username of the user which I have created earlier in SQL Server Studio.

For the password, I can leverage the Azure Key Vault in which I have created secrets

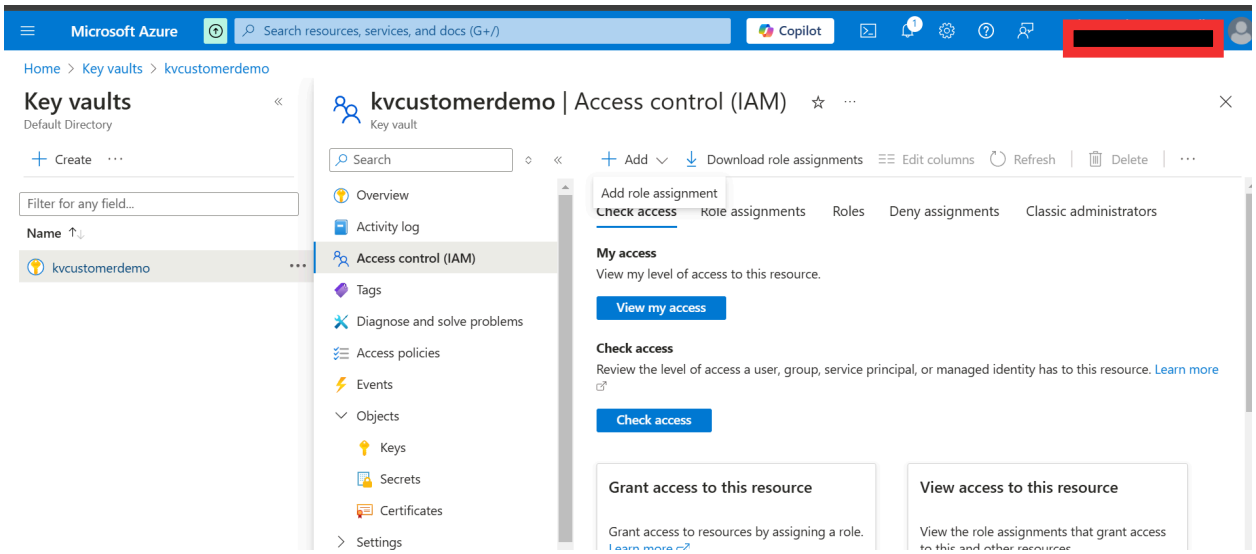


Clock “New” for the AKV linked service. Name the linked service “Azurekeyvault1”. Choose “From Azure subscription”. Select free trail, select the azure key vault I have created earlier while I was setting up the infrastructure. Choose “system-assigned-managed-identity” in the Authentication Method section.

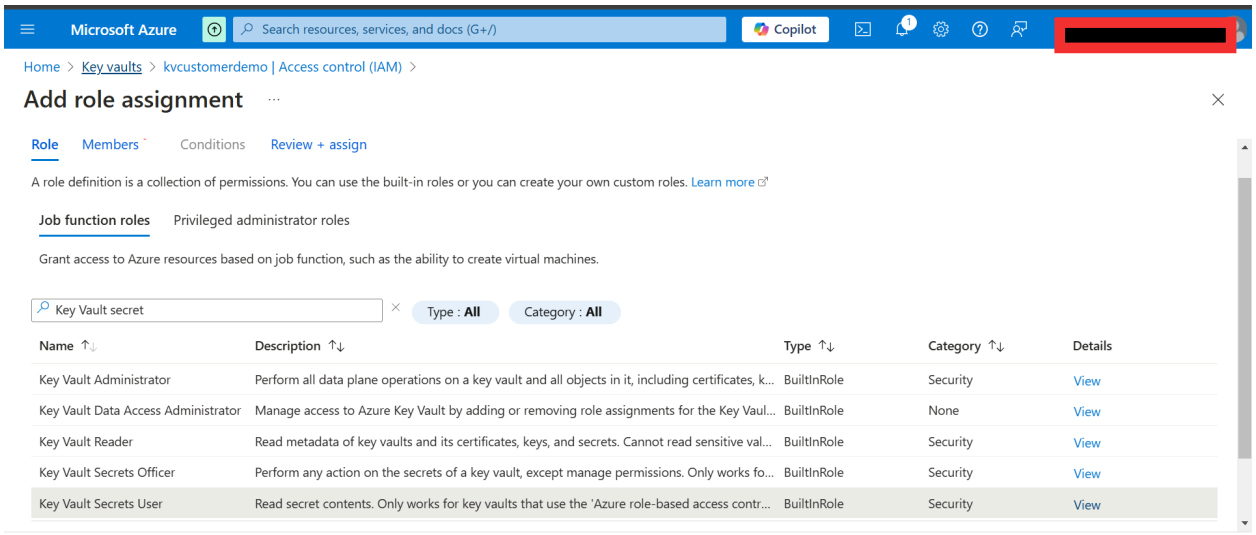


Here I am not able to see any secret name even though I created a secret in the azure key vault earlier.

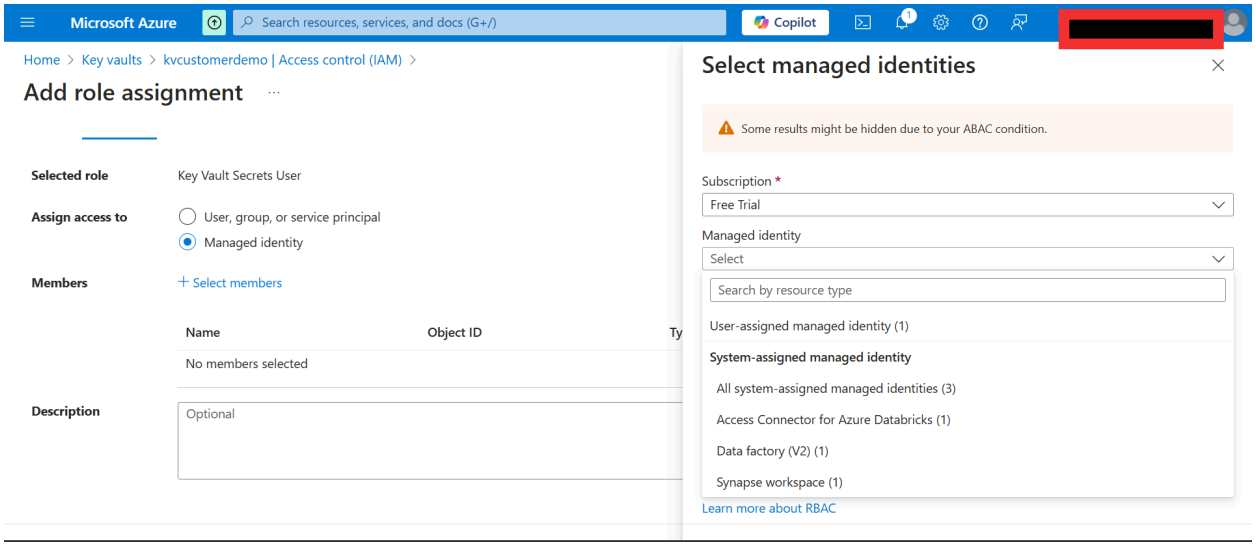
To resolve this issue,
Go to the key vaults, select the “kv customer demo” key vault I have created earlier.



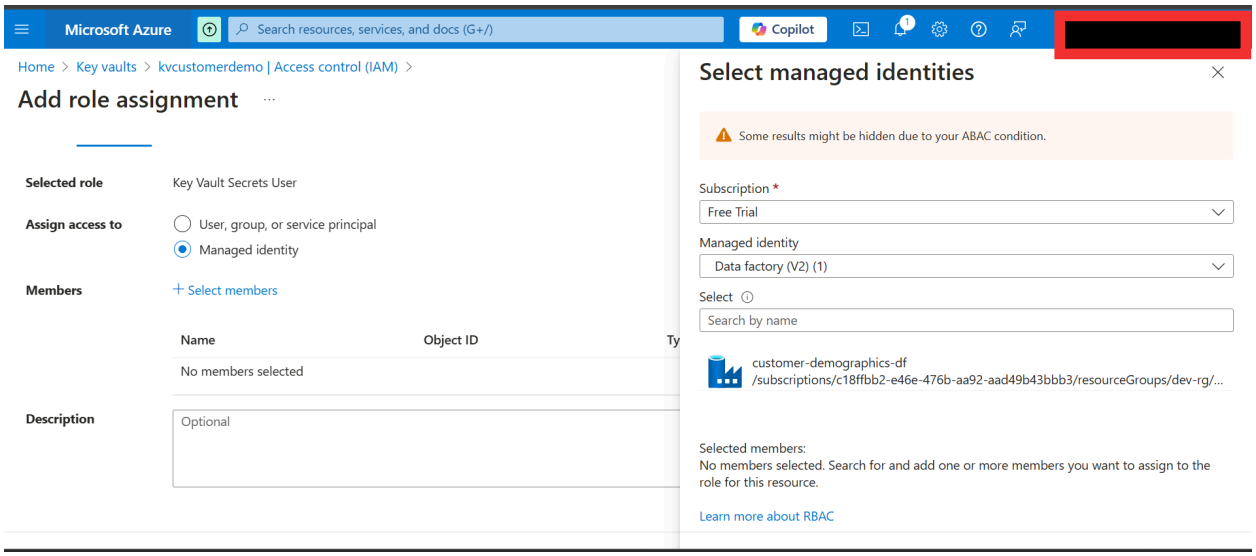
Click the “Add role assignment” and select “Azure Key Vault Secret User”

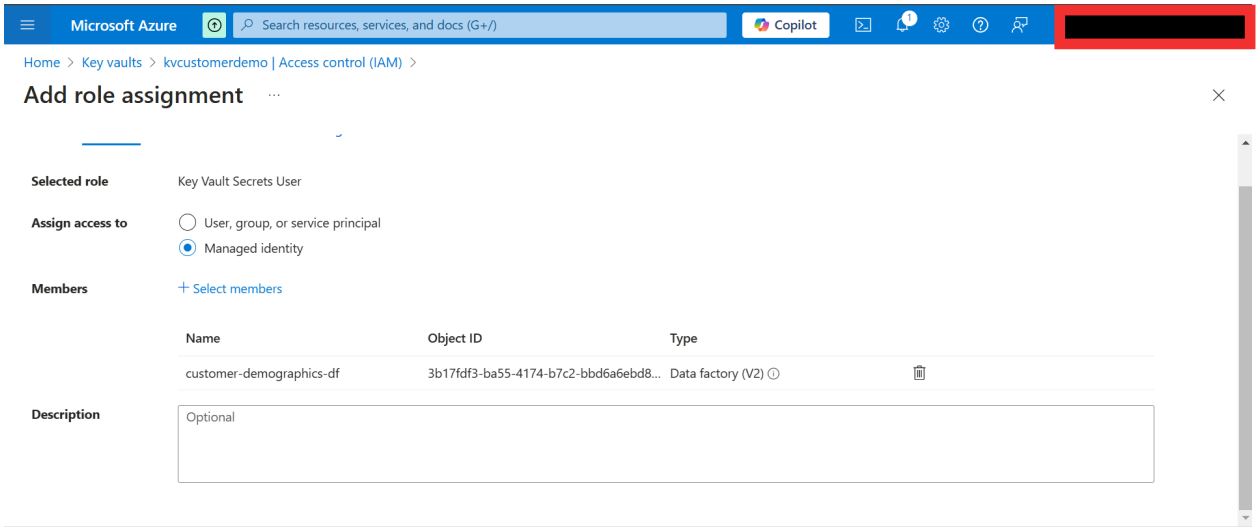


Select the “Managed Identity” section

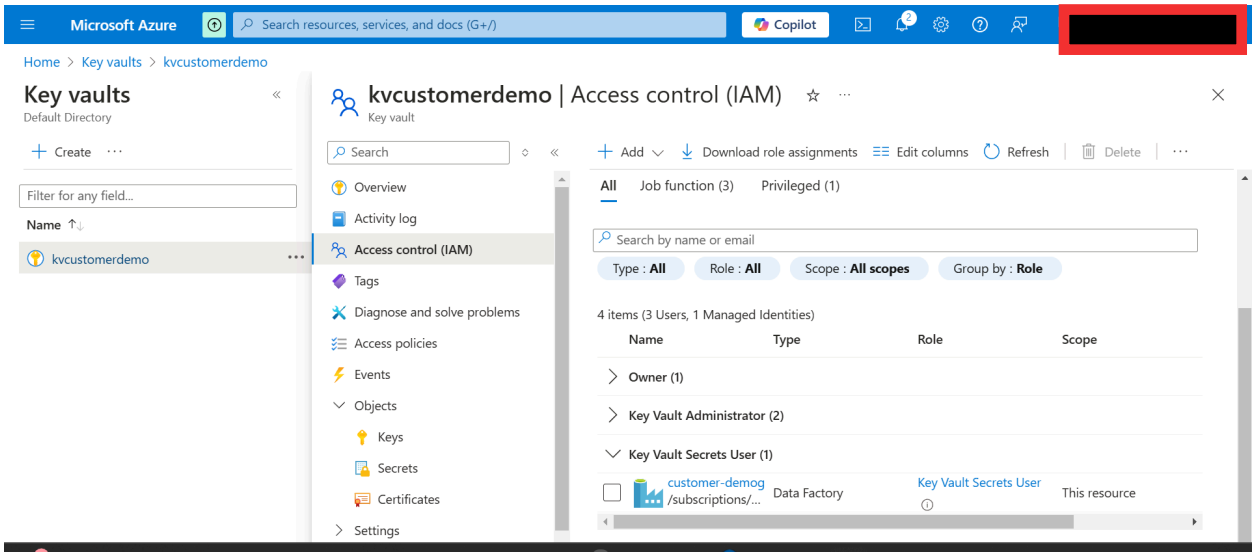


Now, choose “Data Factory”

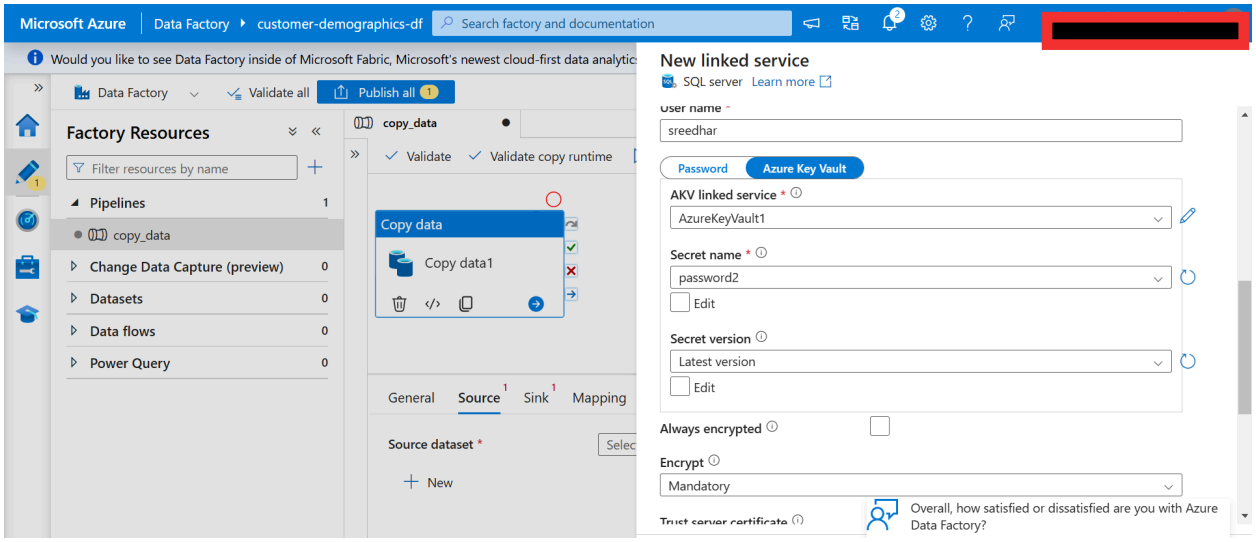




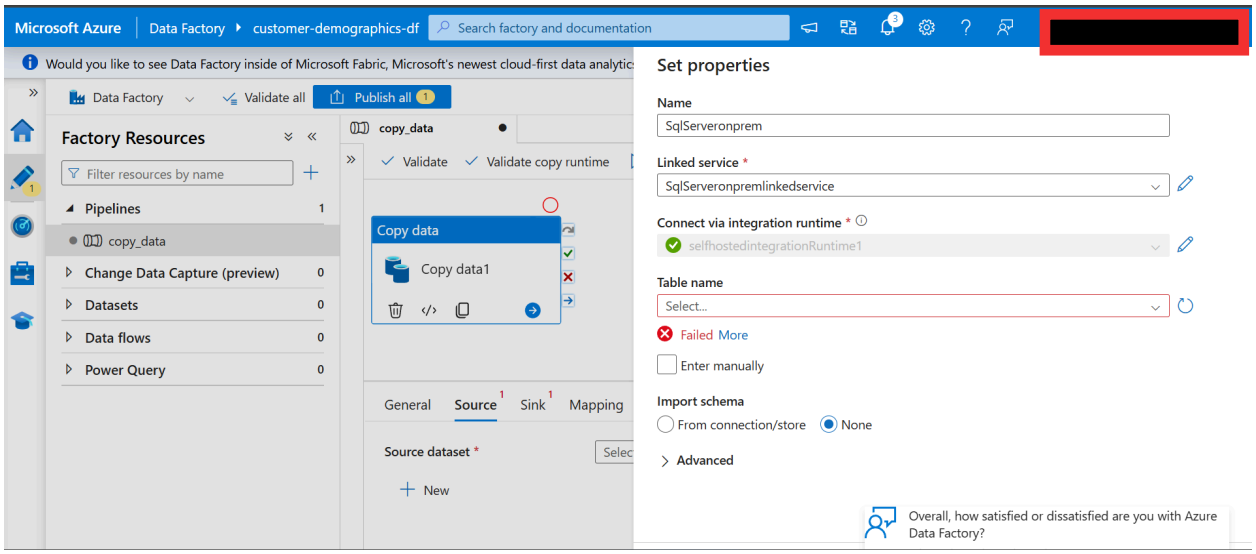
Once I click “Review and assign”, I can see the following page in which the data factory will be a key vault user



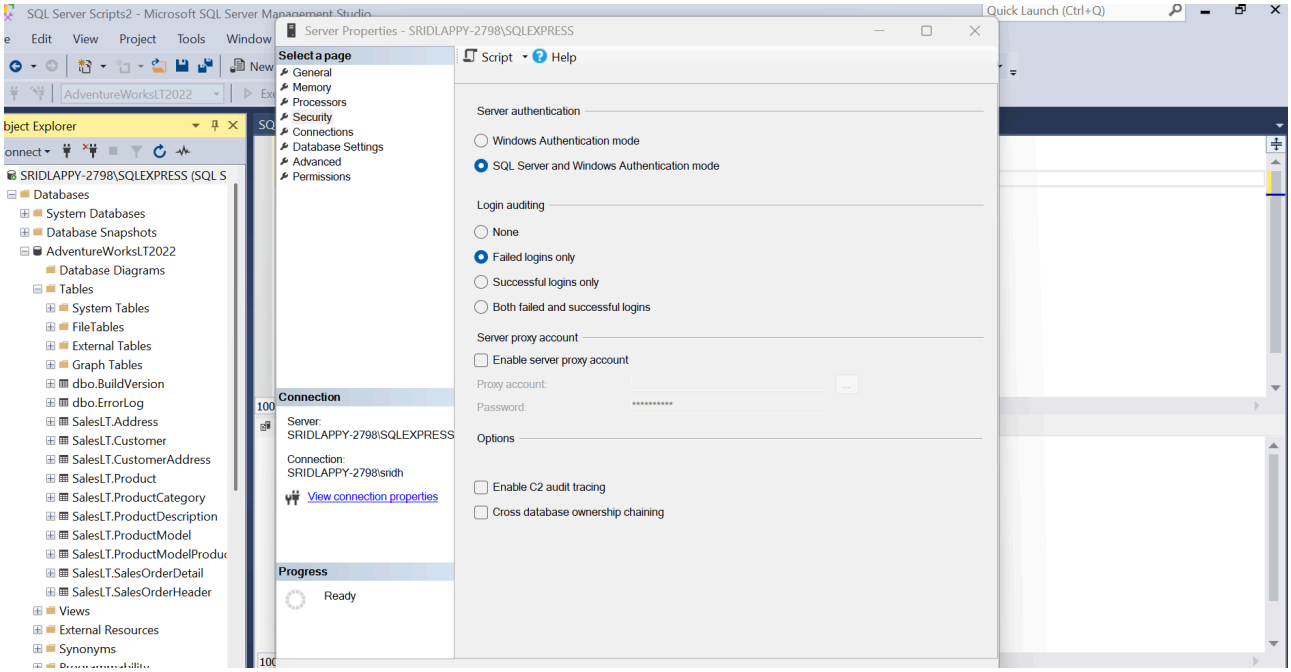
Now, if I go back to the azure data factory, I can see the below secret details that I have created earlier. I can see password2 in the drop down



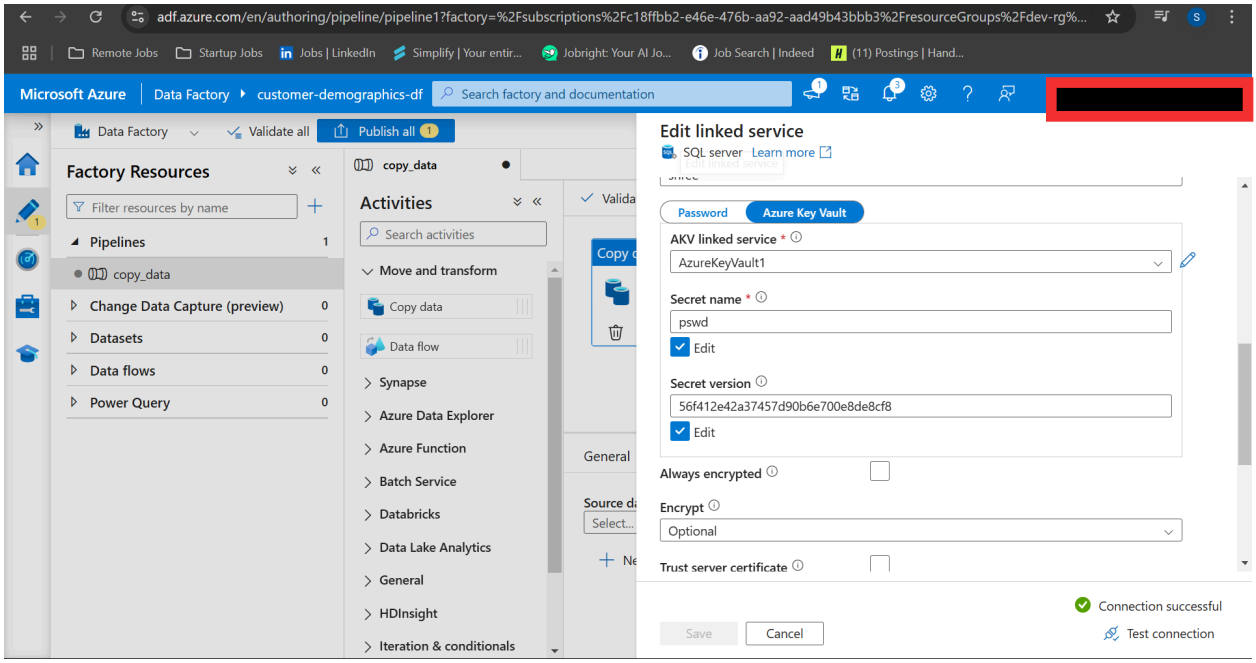
When I create the linked service error, the following error pops-up



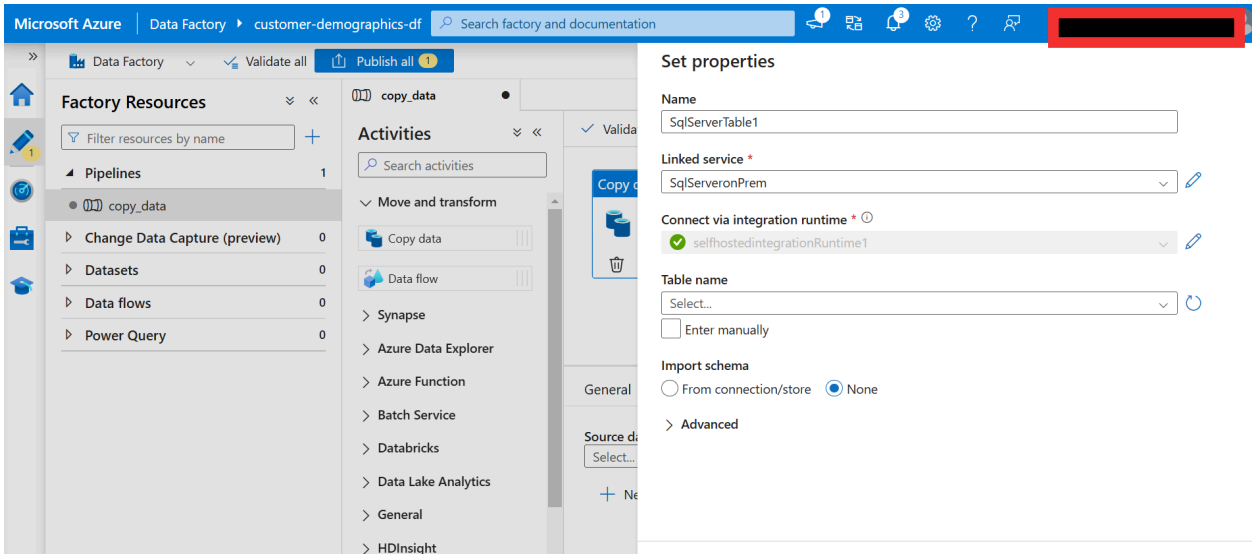
To resolve this issue, go to SQL server management studio, go to the “Properties” section of the database, click on “Security”, select “SQL Server and windows Authentication”



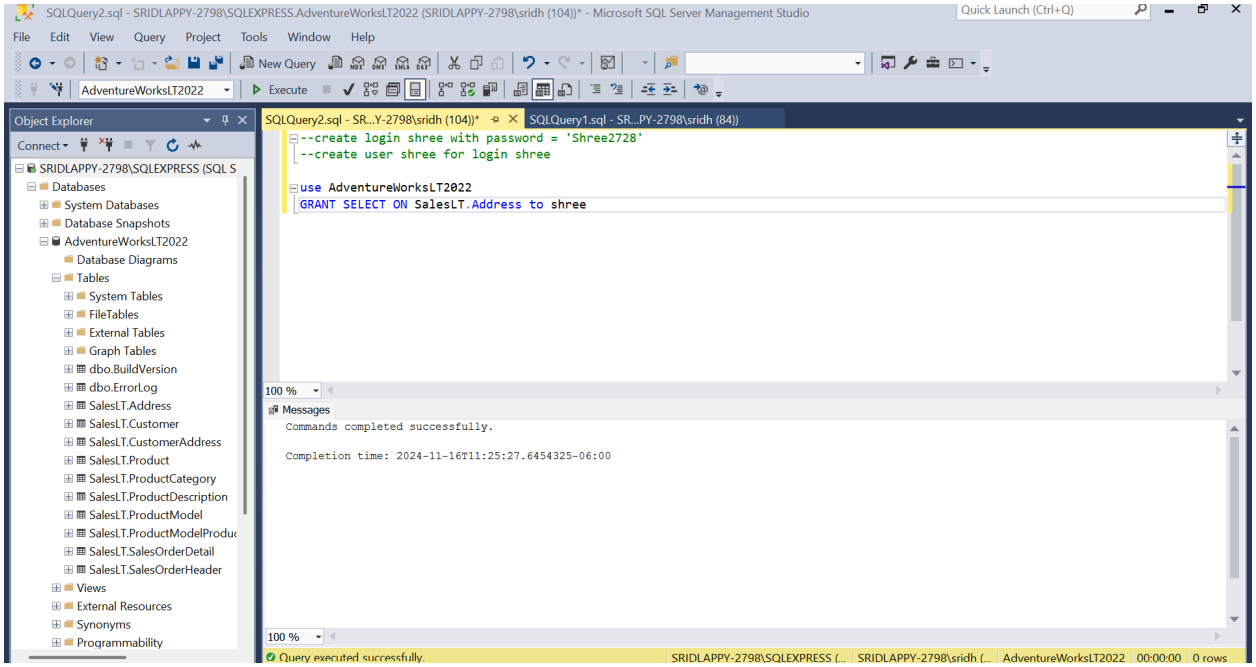
Also, edit the encryption settings in linked service to “Optional” and test the connection in the linked service section. Now I can see the connection is successful



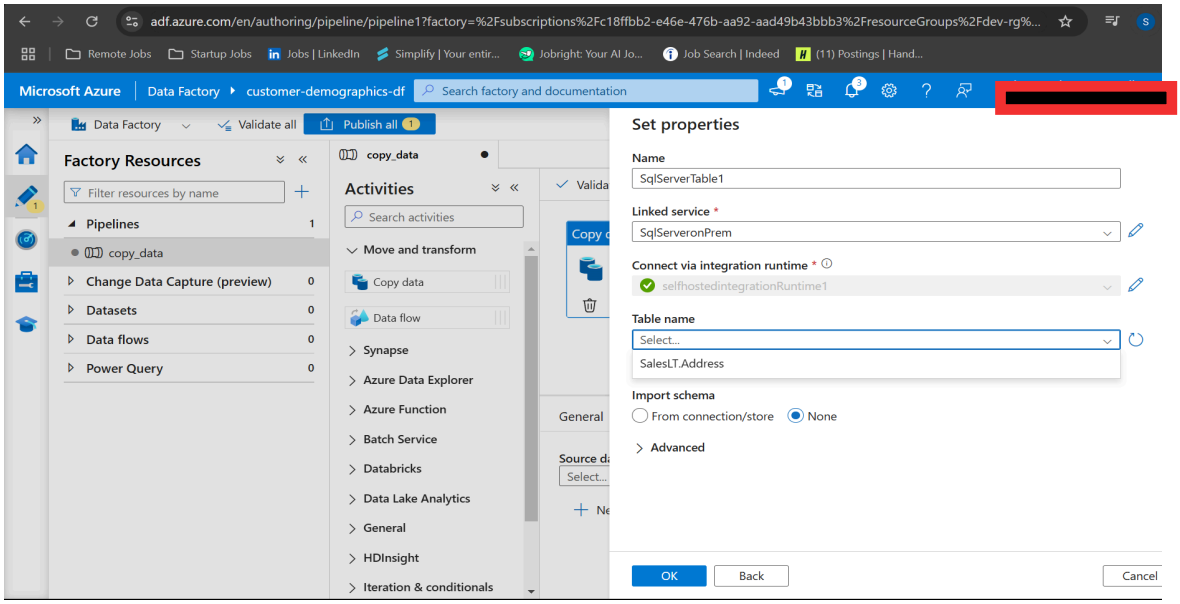
But, if I check the image below, I am not able to find any tables. The catch here is, to grant the user (which I have created in SSMS earlier) the permissions to access the tables.



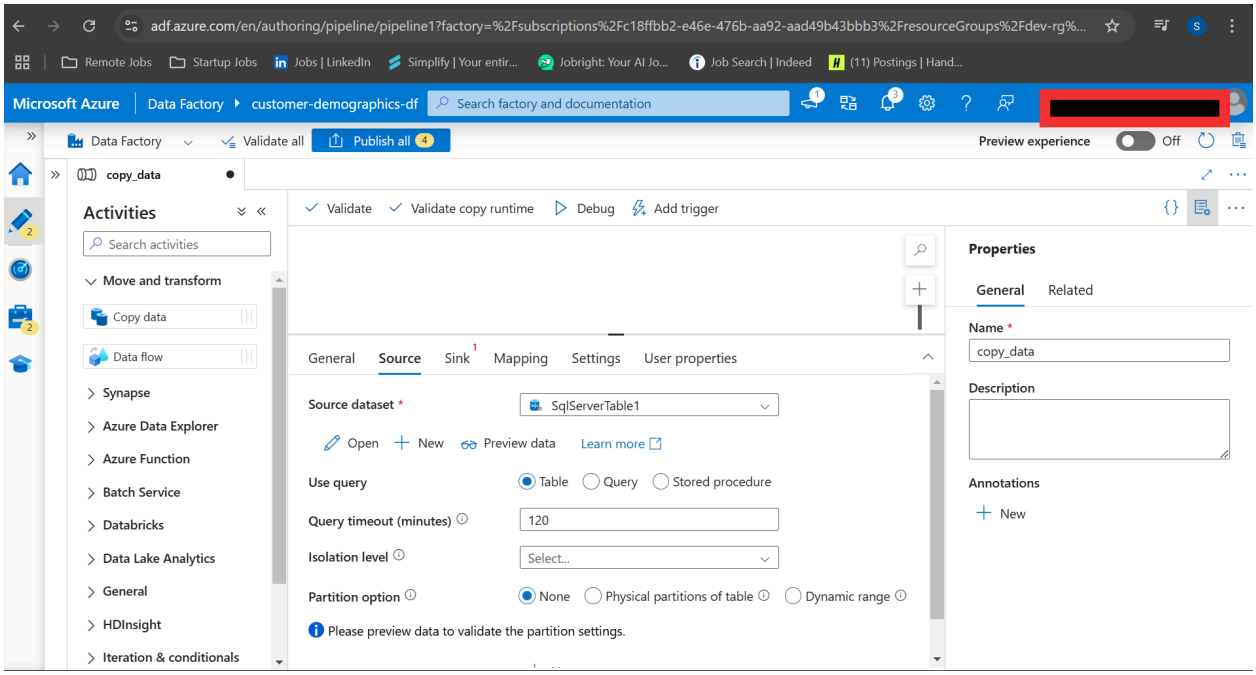
To grant permissions to the user, go to SSMS and execute the following as shown in the image below



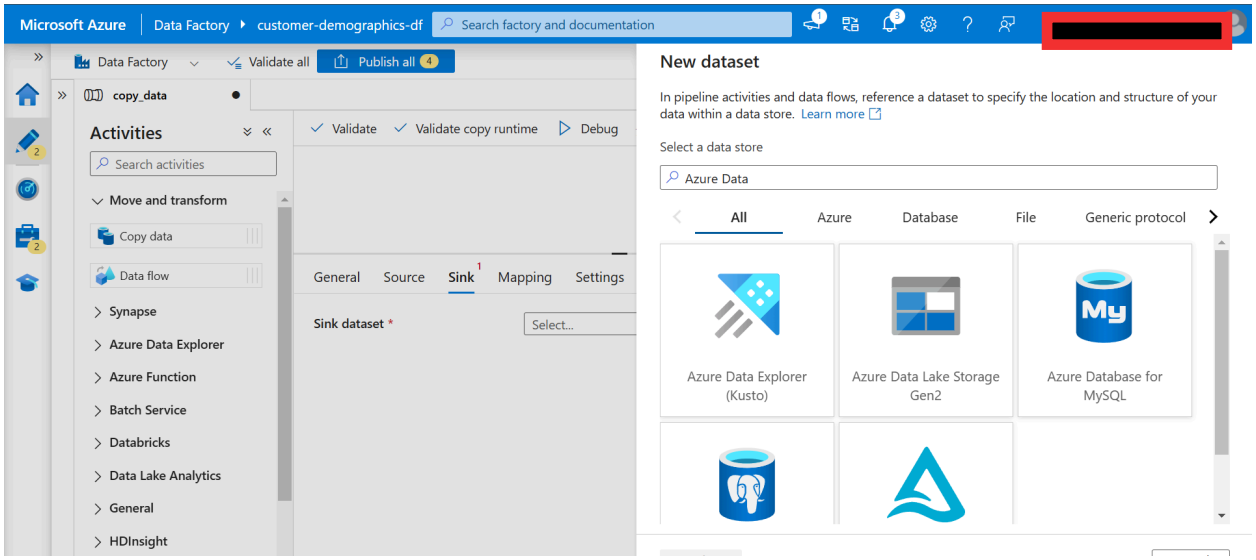
Now, if I refresh the “Set Properties” tab in azure data factory, I can see that the table is available to access



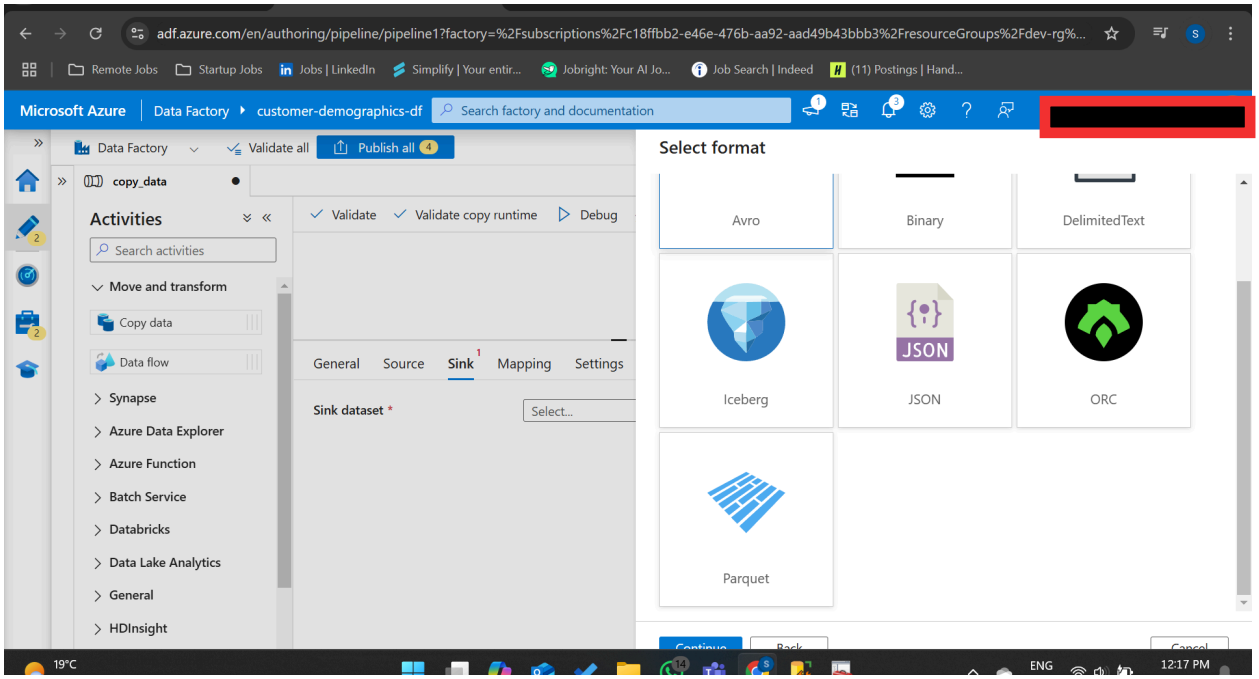
Now, I can see that the source has been provisioned



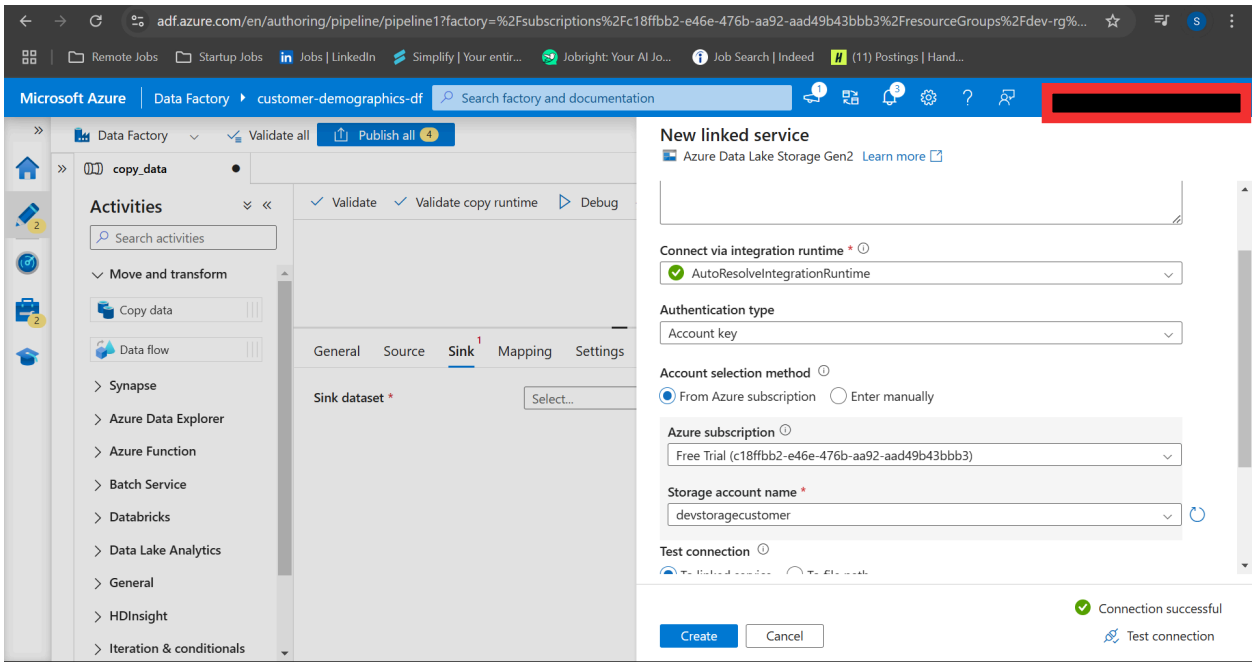
To provision the sink, go to the sink and select “Sink”, and select “New” and select “Azure Datalake storage Gen2”



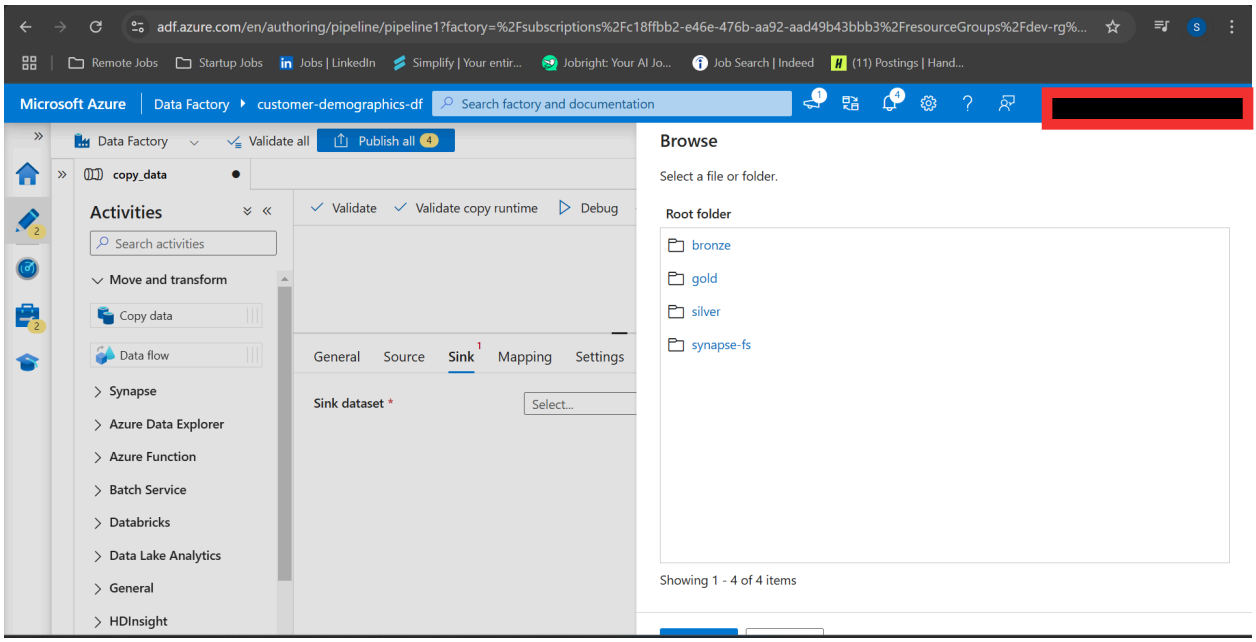
Now I select ‘Parquet' which is a columnar storage format



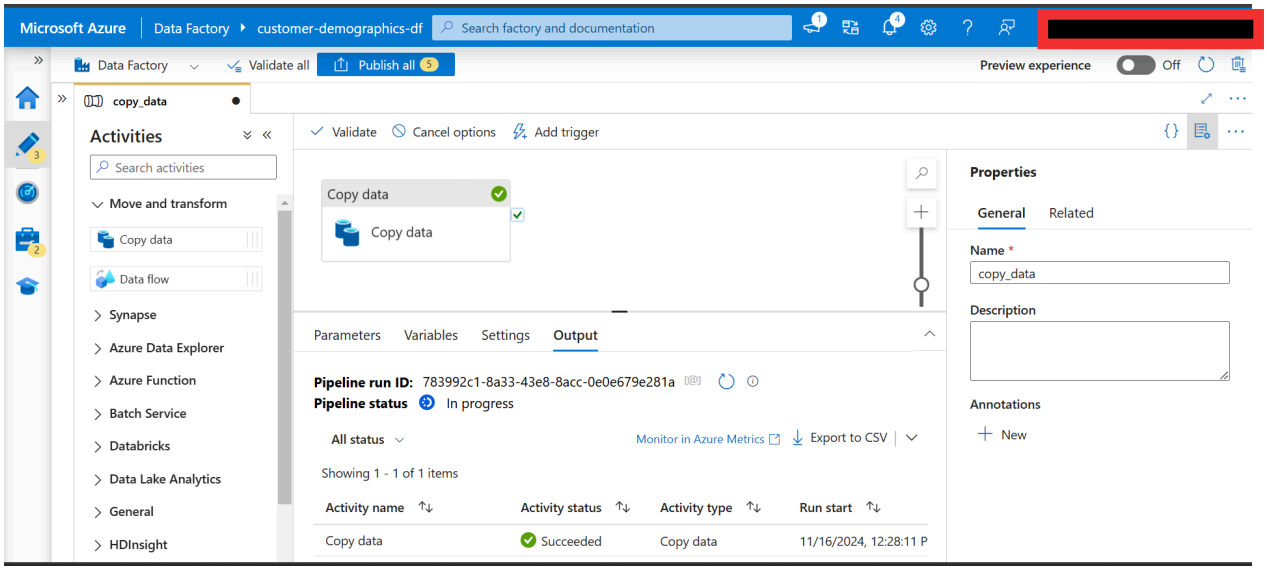
Now, I have to create one more linked service for azure data storage to link to the data factory. Here I choose “Autoresolve Integration Runtime” as the integration runtime because I am choosing another azure service to link to the data factory. Also, select the storage account name which I have created earlier while provisioning resources on azure.

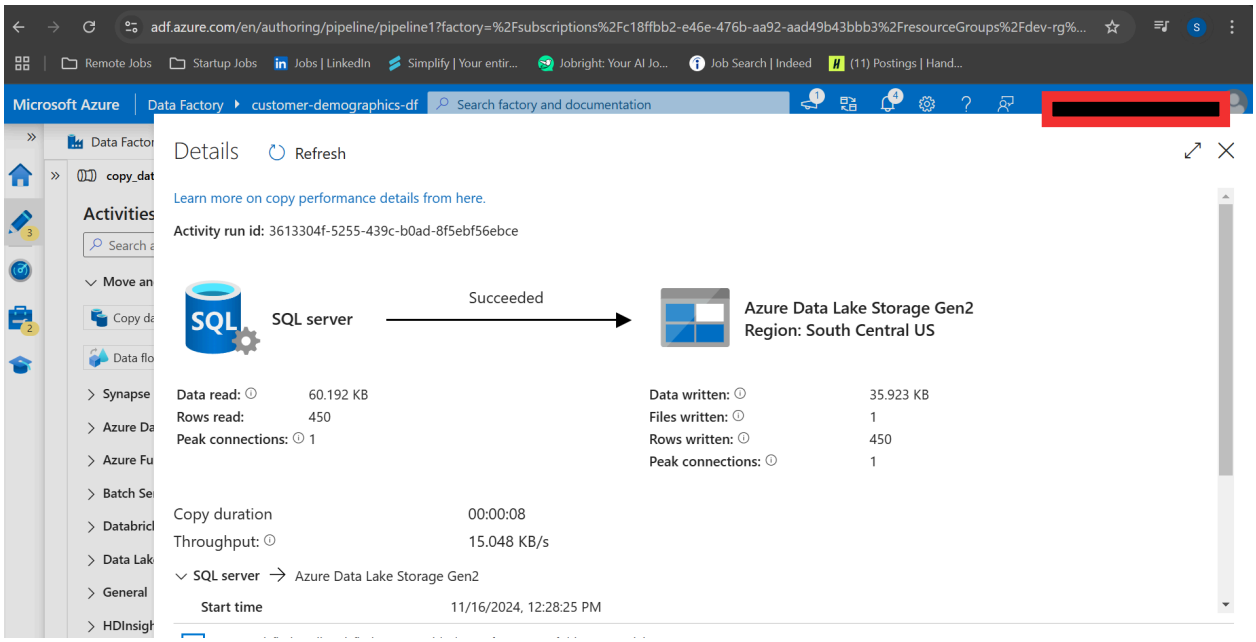


Now, I can select the destination folder as “bronze”.

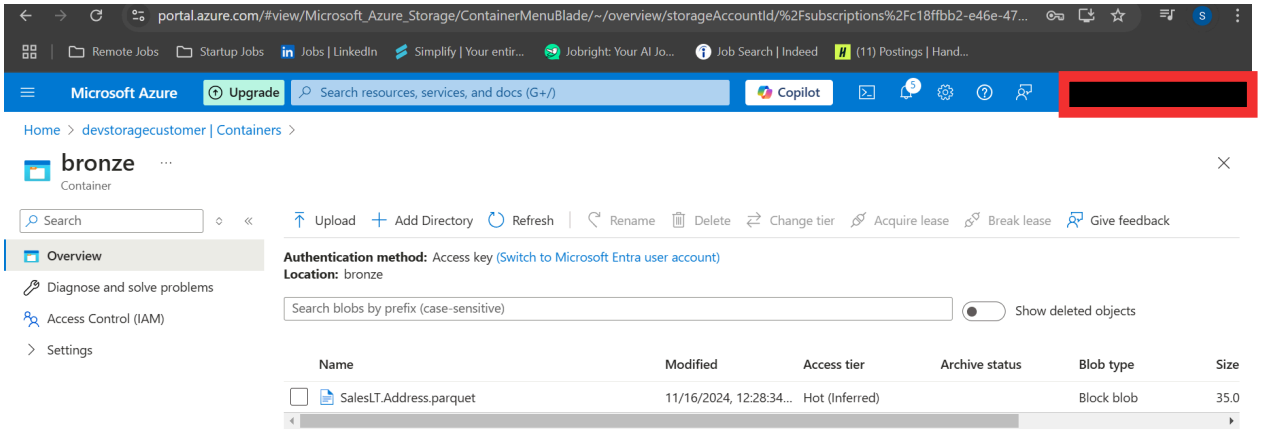


The next step is to debug the pipeline I have created. I can do this by clicking the “debug” button. I can see that the debug was successful.



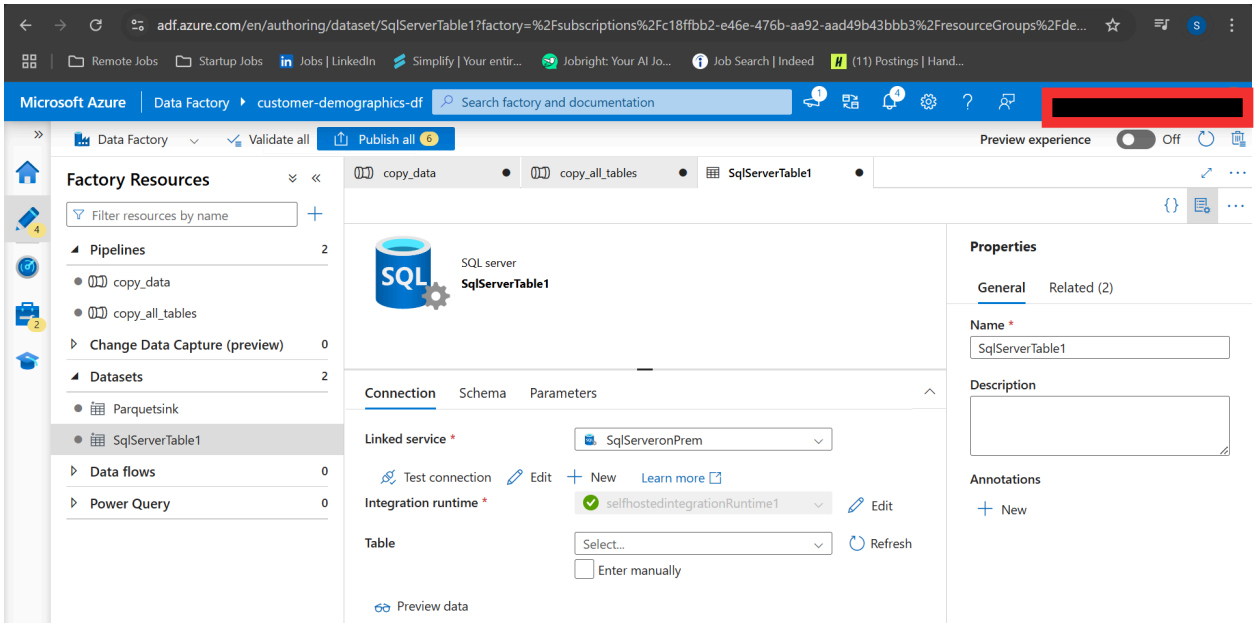


I can also see that the storage files were populated in the bronze container of azure data lake storage.



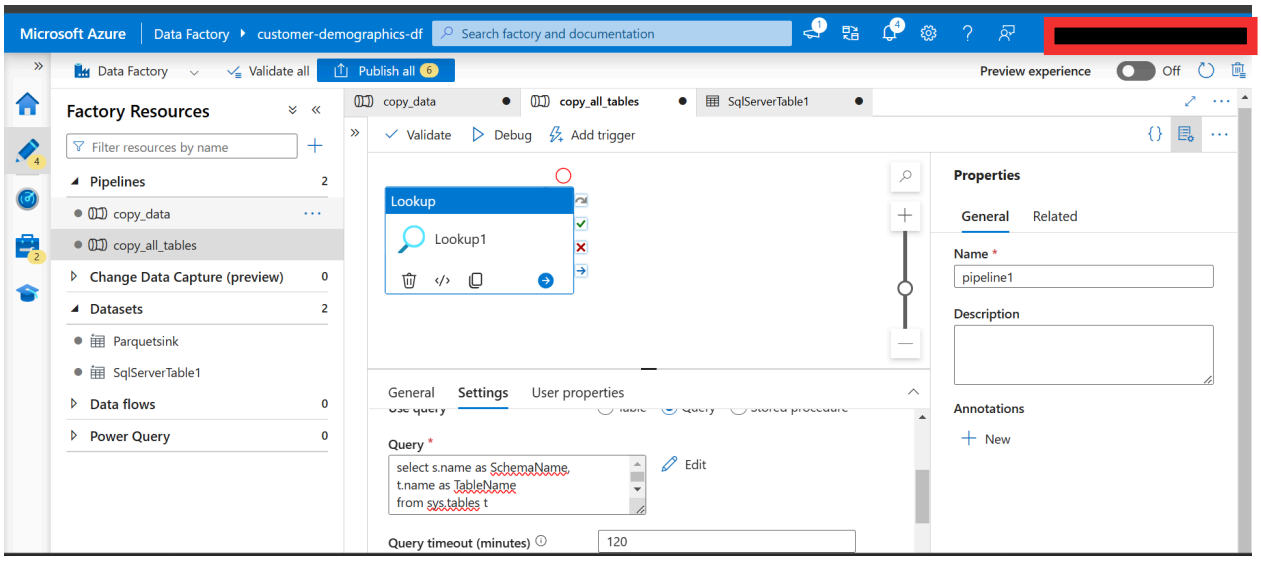
Now, I can create a new data pipeline to iterate through every table in the database and it brings all the tables into the bronze layer. To achieve this, I can create a new pipeline to copy all the tables (name it as “copy_all_tables”).

The first step is to get the list of all the tables automatically so that if more tables are populated in the database later, this pipeline needs to bring those newly populated tables too. For this, I have to use the “Lookup” activity in the “General” section of the “Activities” tab in Azure data factory. Select the source as “SQLServerTable1” and open the linked service.

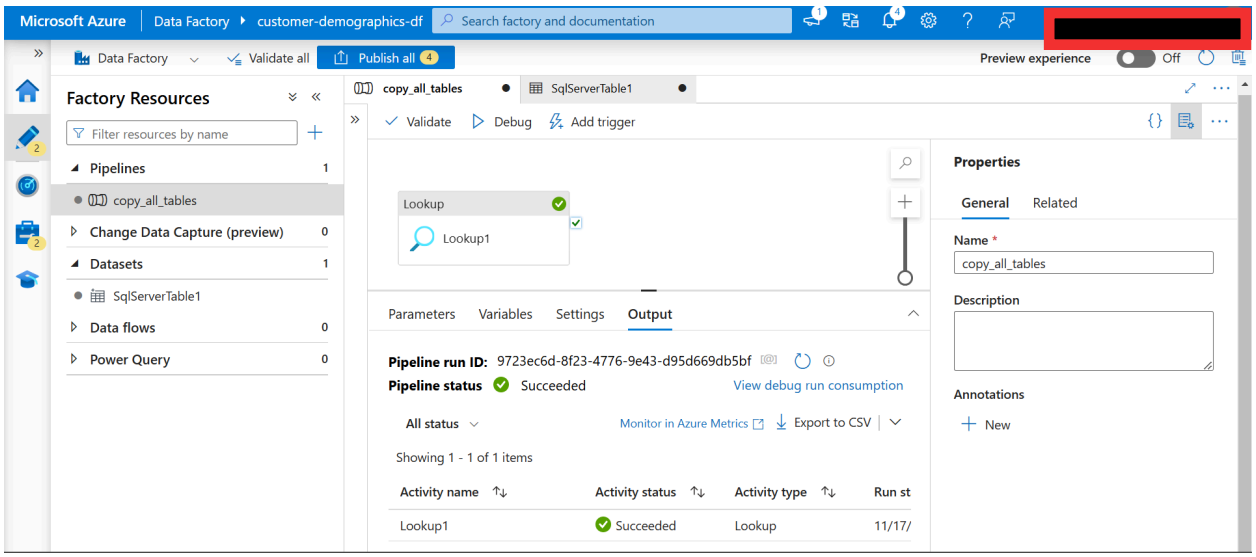


Now go to the settings of lookup activity to write a SQL query to get the list of table names in the database. Use the following SQL query to get the list of tables.

```
SELECT
   s.name as SchemaName,
   t.name as TableName
FROM
   sys.tables t
INNER JOIN
   sys.schemas s
ON
   t.schema_id = s.schema_id
WHERE
   s.name = 'SalesLT'
```



Once I debug the lookup activity, I can find the below screen

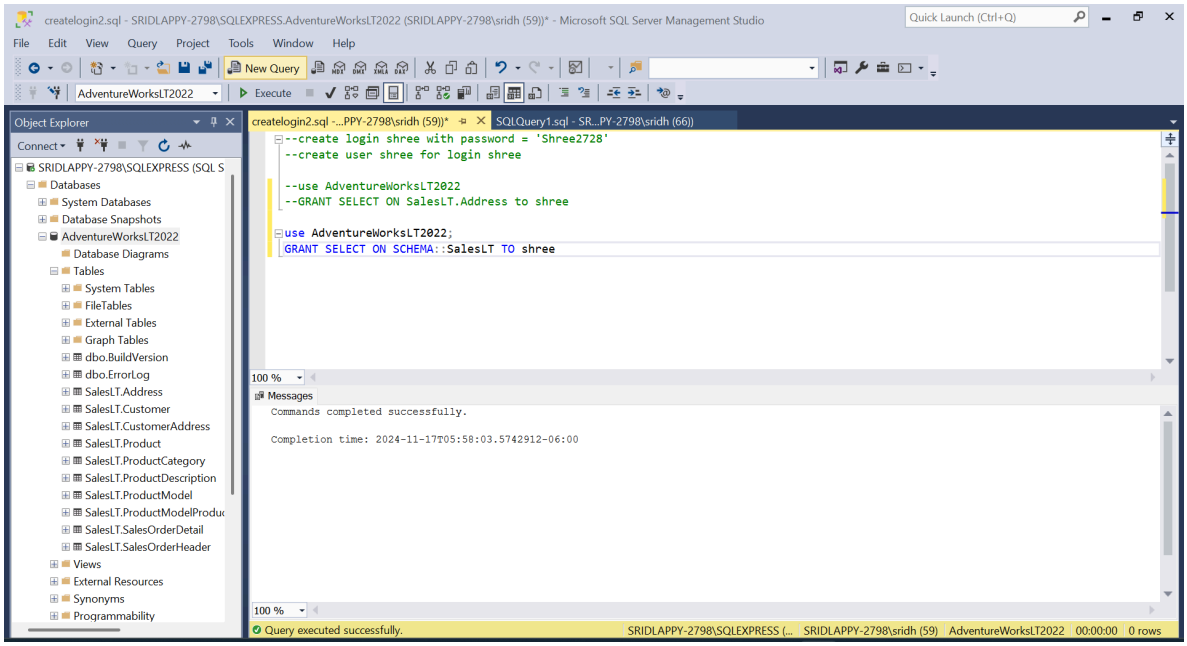


But if I take a closer look at the input and output of lookup activity, I can find that lookup activity only brought “Address” table. The output JSON of lookup activity looks like below

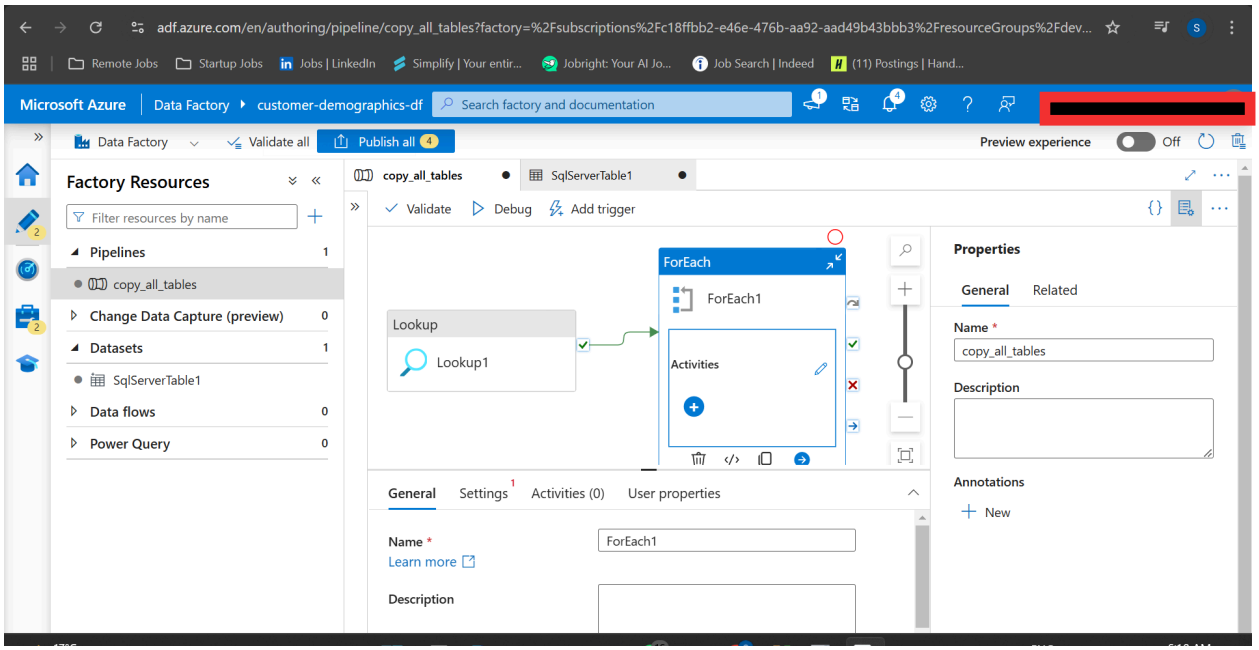

```
{
	"firstRow": {
		"SchemaName": "SalesLT",
		"TableName": "Address"
	},
	"effectiveIntegrationRuntime": "selfhostedintegrationRuntime1",
	"billingReference": {
		"activityType": "PipelineActivity",
		"billableDuration": [
			{
				"meterType": "SelfhostedIR",
				"duration": 0.016666666666666666,
				"unit": "Hours"
			}
		],
		"totalBillableDuration": [
			{
				"meterType": "AzureIR",
				"duration": 0.06666666666666667,
				"unit": "Hours"
			}
		]
	},
	"durationInQueue": {
		"integrationRuntimeQueue": 6
	}
}
```


This happened because, earlier when I granted permissions for the user in SSMS, I only granted the select permissions for the user on “Address” table. In order to pull all the tables, the user needs to have select permissions on all the tables. The permissions on all the tables can be granted using the following SQL query

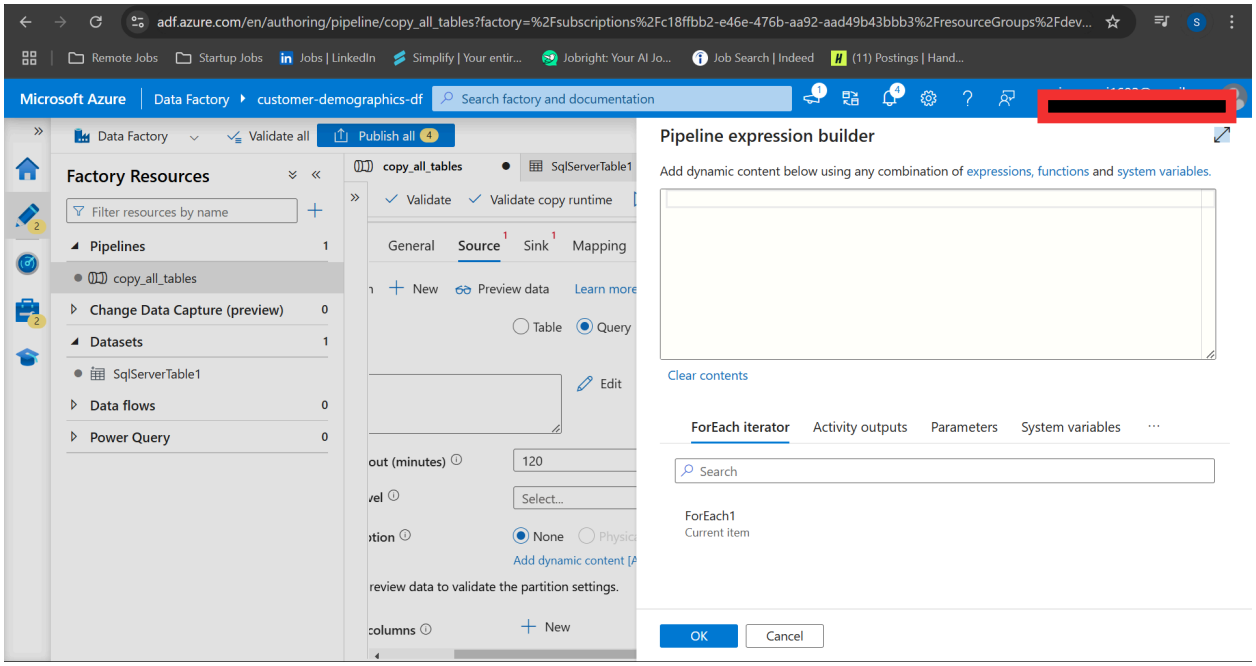
```
use AdventureWorksLT2022;
GRANT SELECT ON SCHEMA::SalesLT TO shree
```



Now, the user has all the required permissions to access all the tables. Go to the azure data factory and attach a “ForEach” activity which happens on the success of lookup activity.



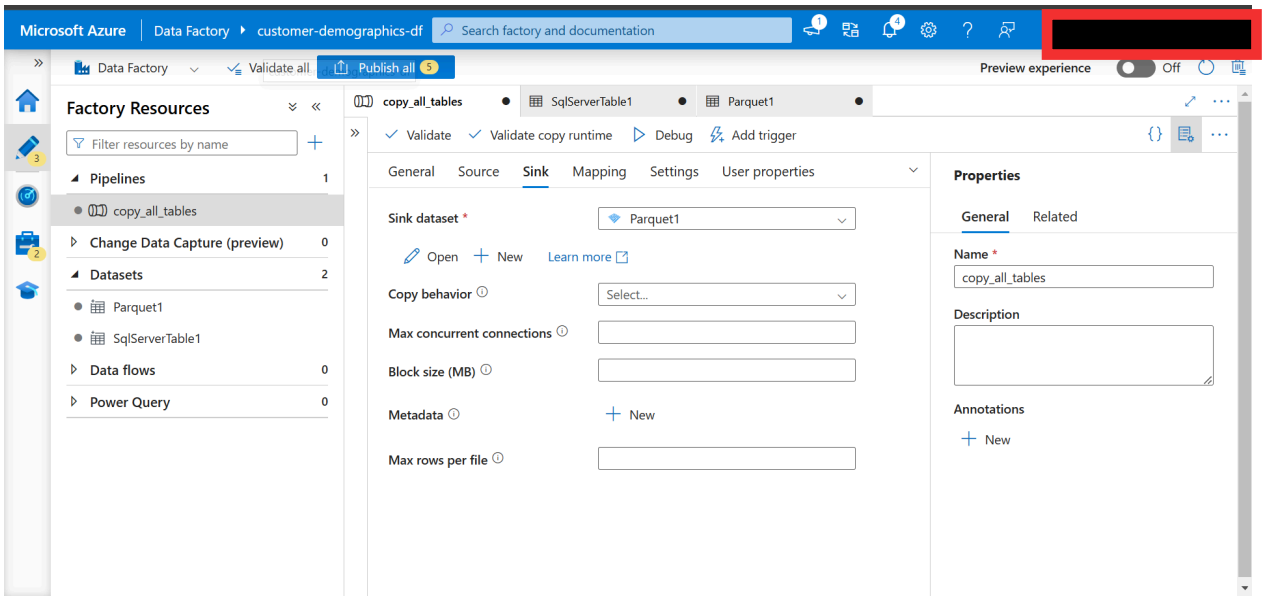
Now, I have to edit the activities in the “ForEach” activity. In the source section of the copy data activity, I have to select query and choose “Add Dynamic Content” option as I need to select the tables dynamically and pull them to the bronze layer.



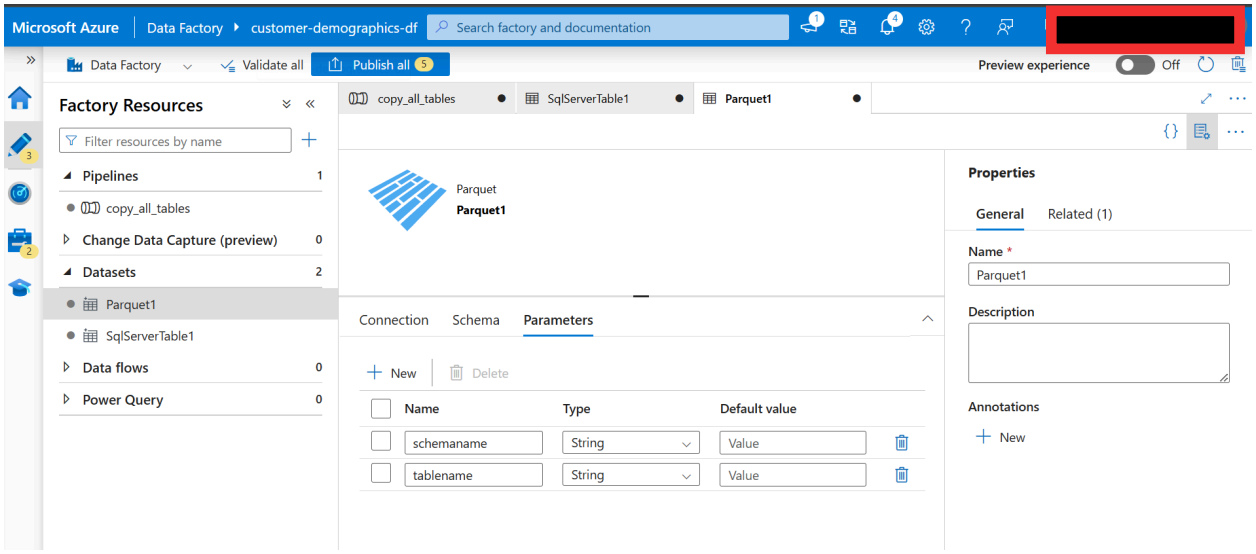
In the pipeline expression builder, use the following expression

```
@{concat('SELECT * FROM',item().SchemaName,'.',item().TableName)}
```

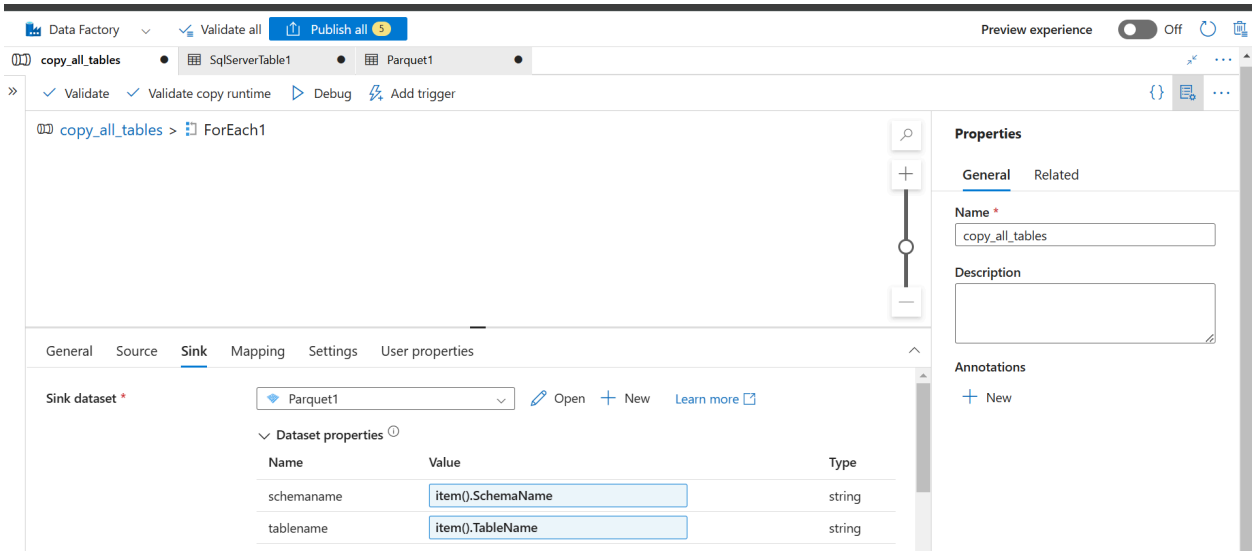
Add the sink dataset details.



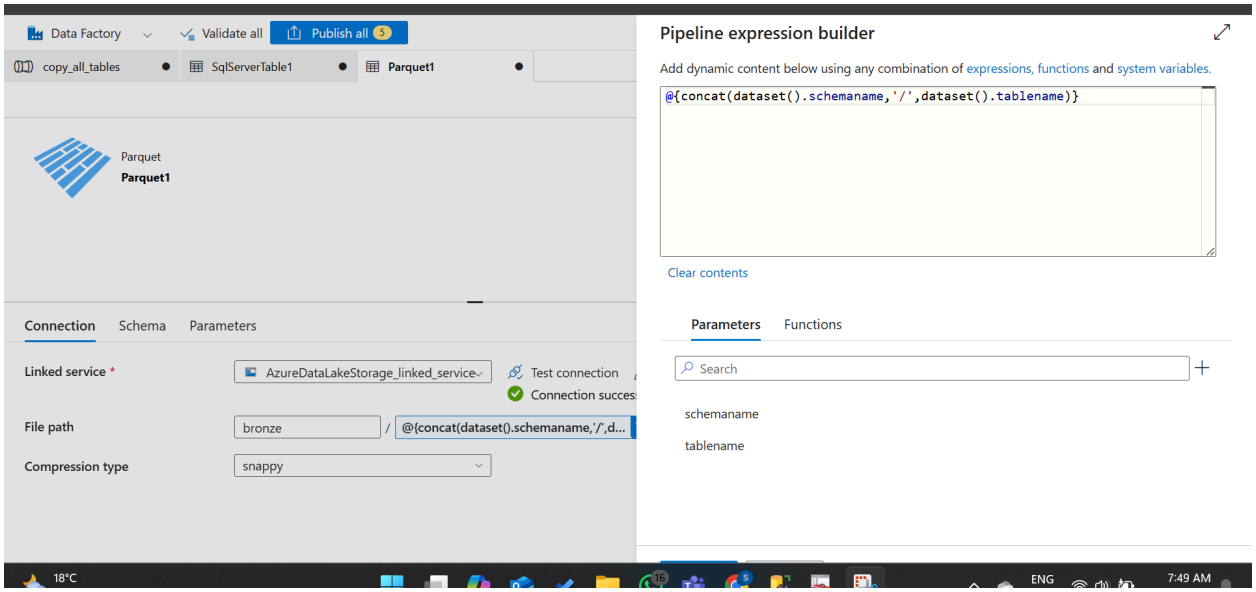
In the sink parameters, add the following parameters



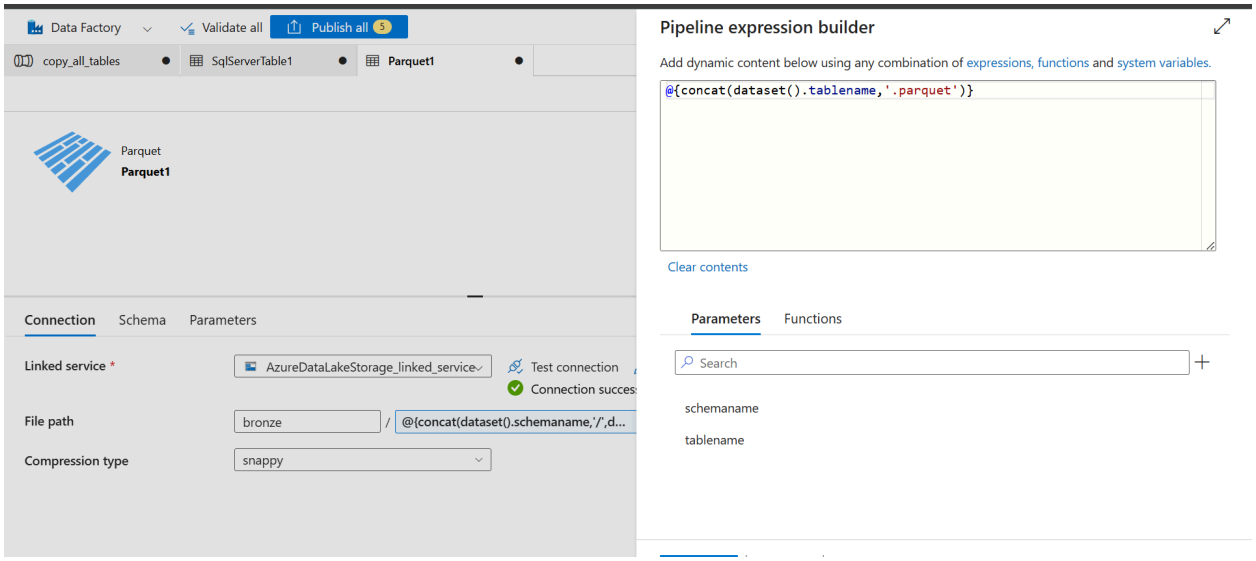
Now, I add dynamic content value for each parameter I have created. For the schema name and tablename, add the dynamic content as follows:



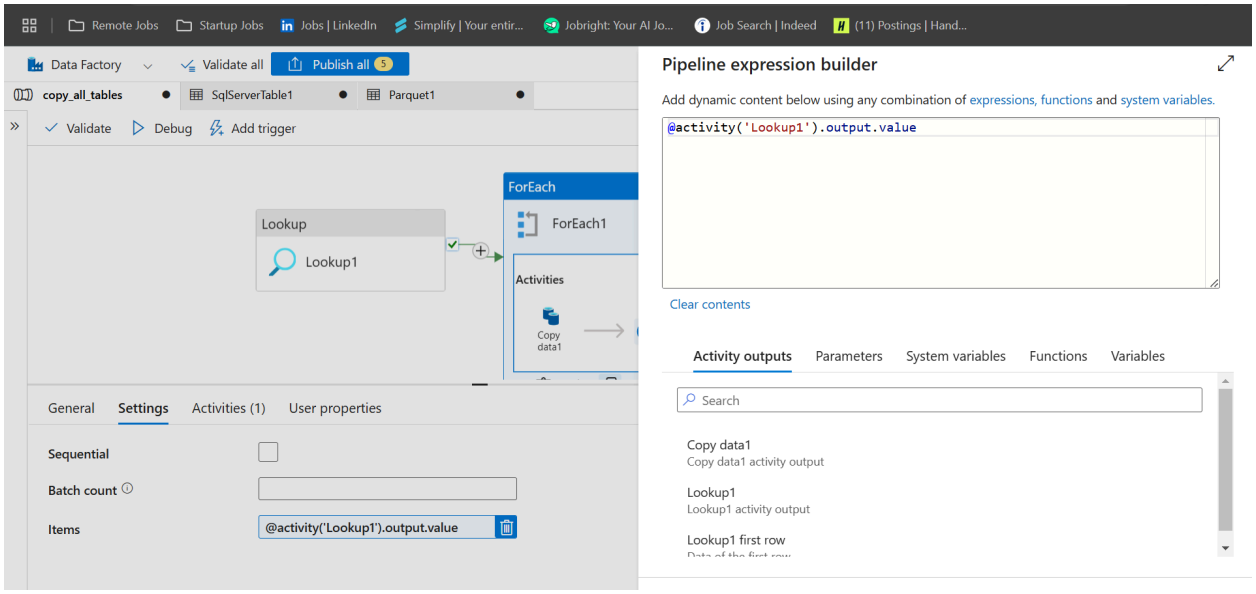
In the connection settings of the sink, in the filepath, add the following expression in pipeline builder



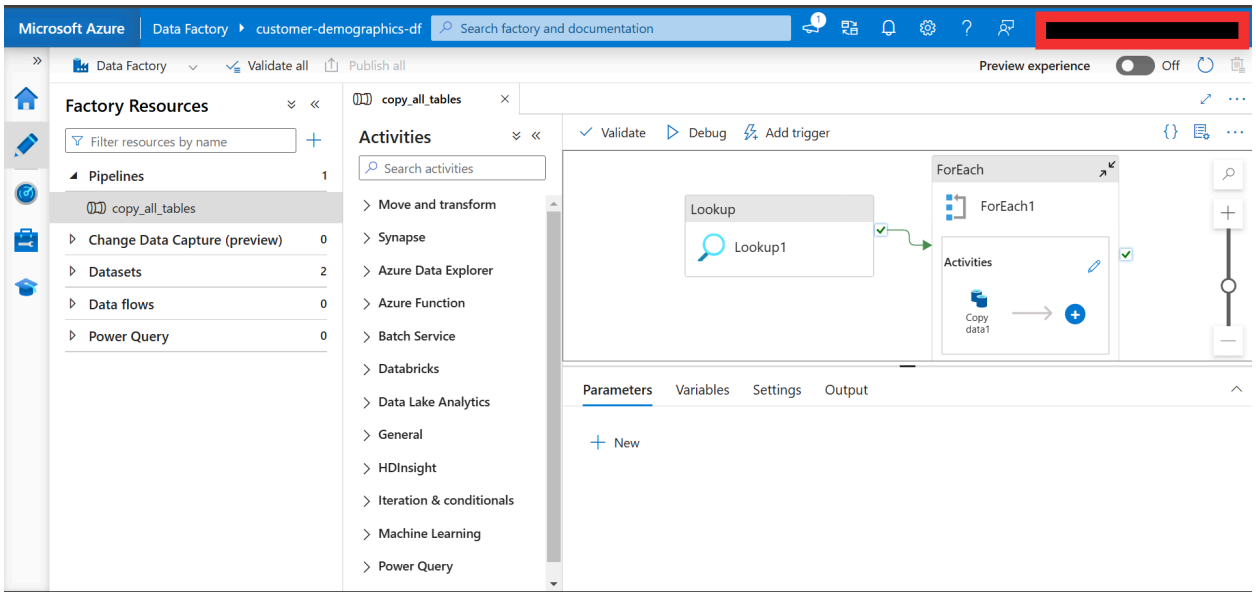
In the File Name, add the below expression in pipeline builder



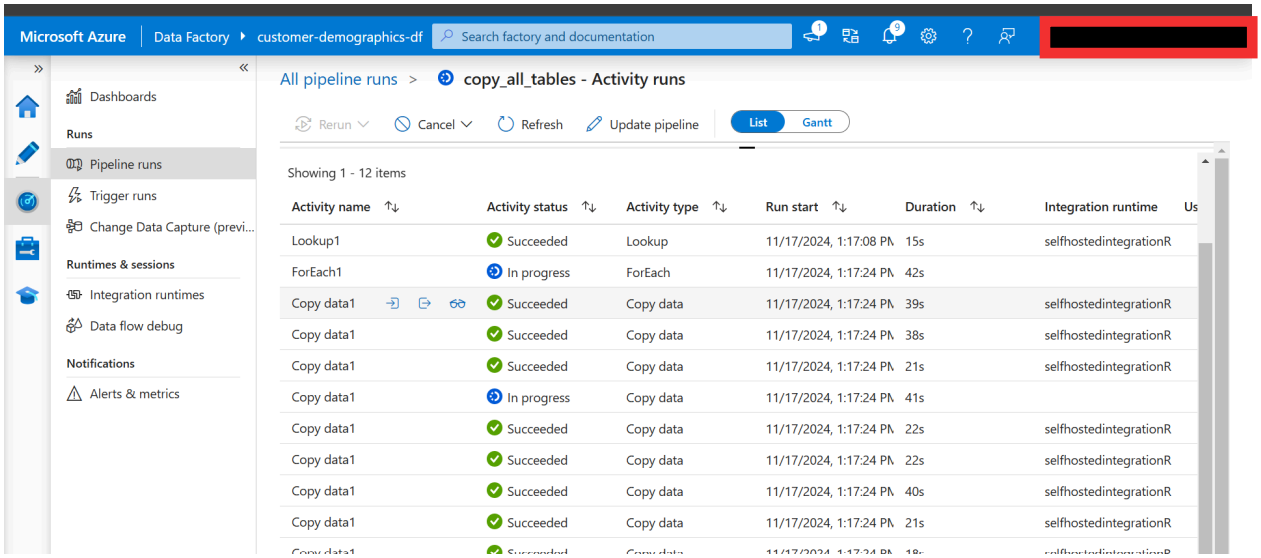
In the pipeline view, for the “ForEach” Activity, in the “Settings” option, add the following pipeline expression builder



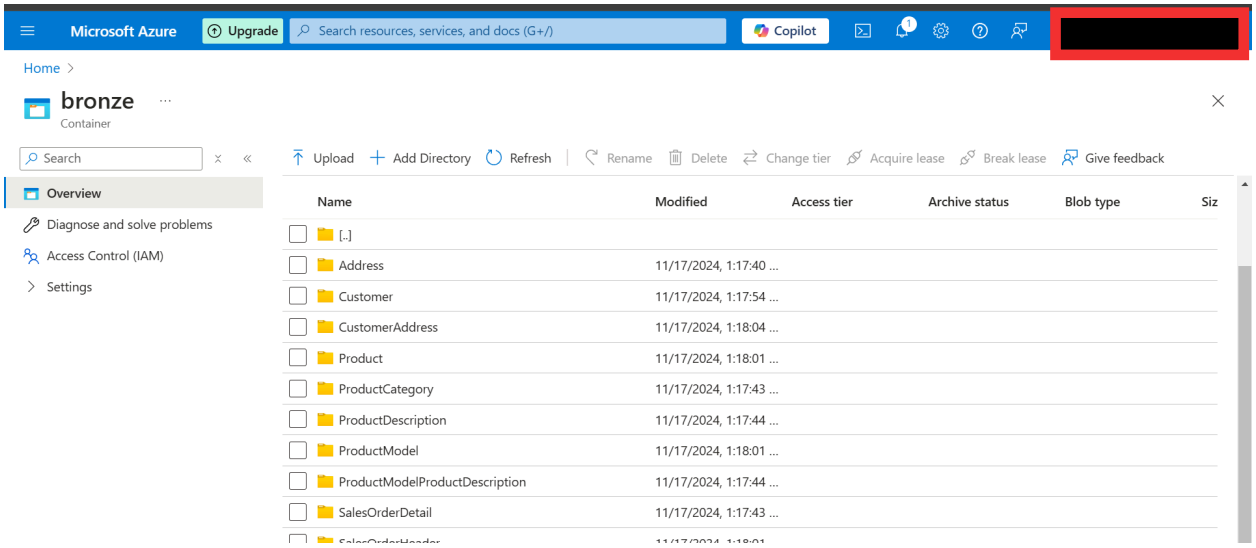
Once the changes are done, publish the pipeline so that all the resources are deployed. Now I can add the trigger. Generally, triggers are used to trigger the pipeline on a daily basis or (to trigger when there is an update in the database). I can trigger this pipeline by using “Trigger Now” option.



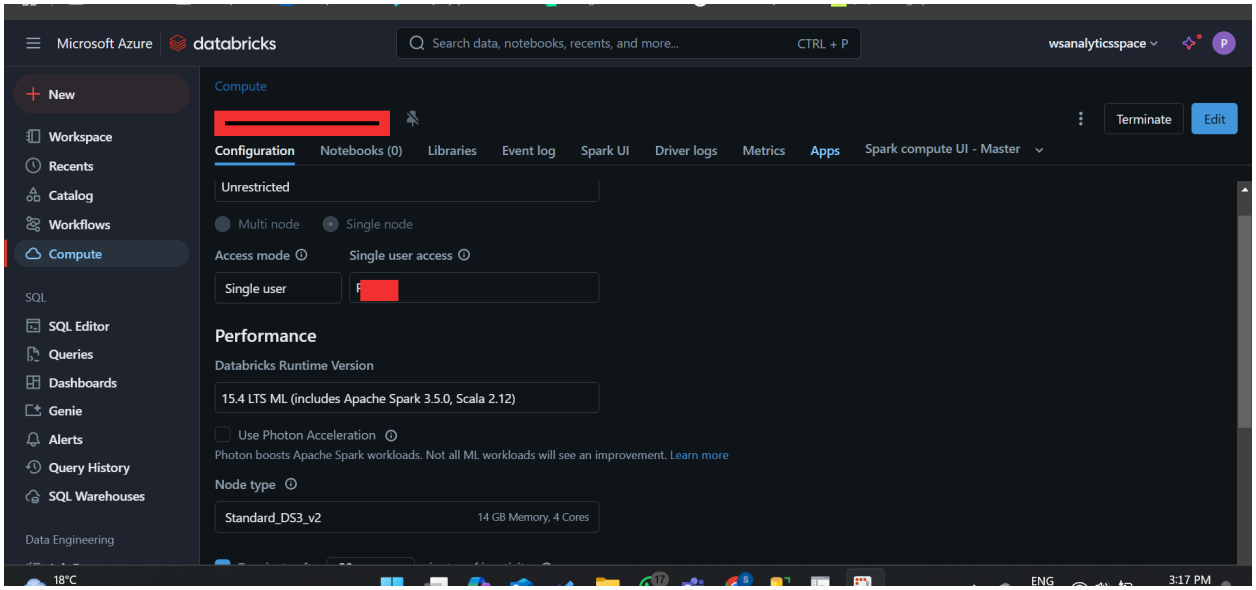
I can see below that the pipeline run was successful



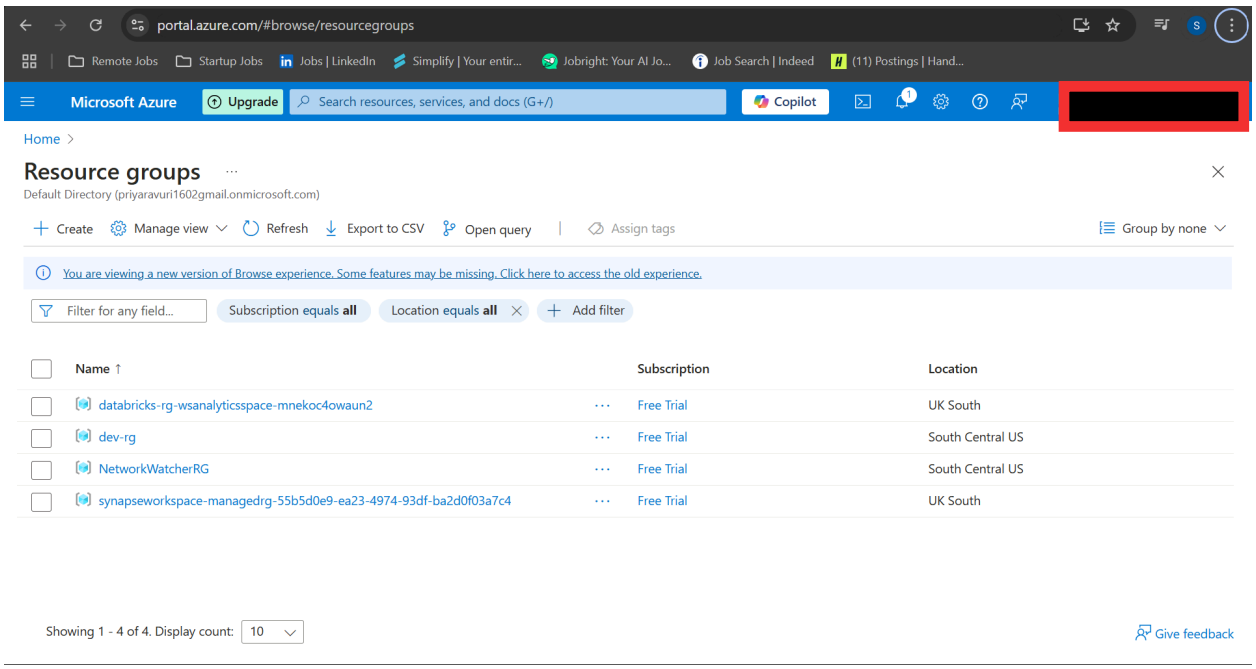
I can also see that bronze layer now consists of all the tables and all the data in bronze layer is stored in parquet format.



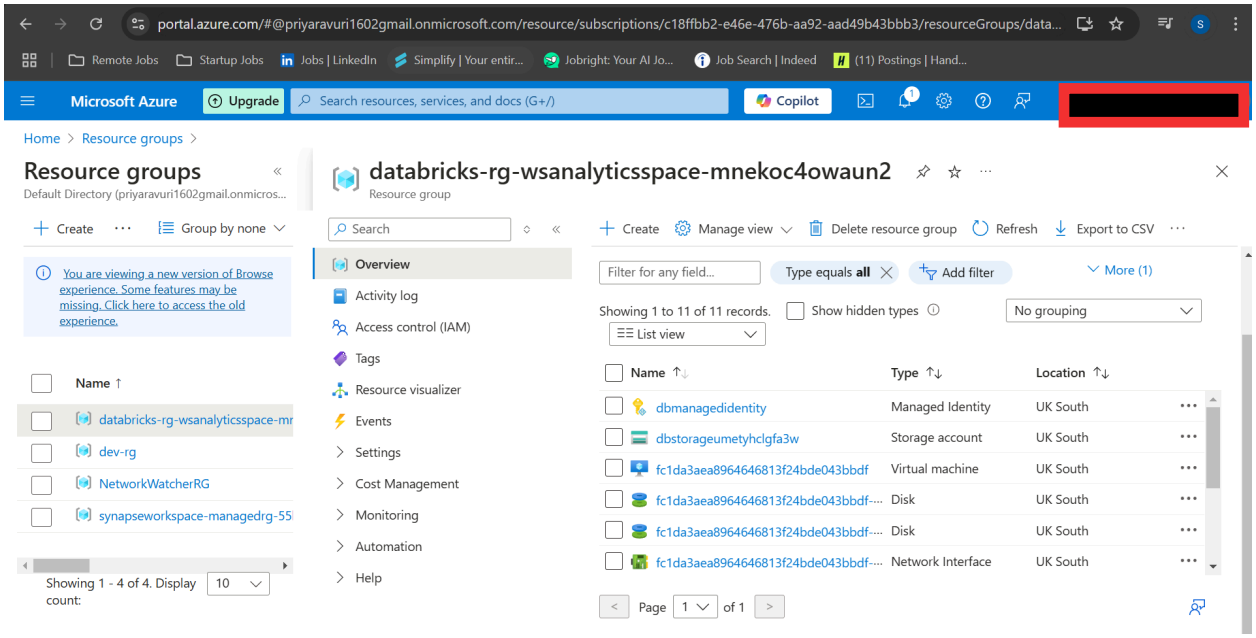
As I have the data in bronze layer, now I can apply the transformations on this data using azure databricks (which I have provisioned earlier) to create silver and gold layers. I need to create a compute which will be used by the databricks notebooks to apply transformations.
Create a compute with the below configuration.



As the cluster creation completes, I found another resource group in my azure account, with all the resources provisioned that can be used by databricks workspace



The resource group with the name"databricks-rg-wsanalyticsspace-mnekoc4owaun2" is the resource group that will be used by the databricks.
It has the below resources provisioned



Once I provisioned the compute, I created a notebook with name “mountstorage”. Before I start using the data in Azure Data Lake, I have to mount it to a cluster so that I can have access point to the storage.

Then, I mount the bronze, silver and gold layers to databricks using the following code

```
#Configuration settings for authenticating and accessing ADLS Gen2 storage
adls_configs = {
    "fs.azure.account.auth.type": "CustomAccessToken",
    "fs.azure.account.custom.token.provider.class": spark.conf.get(
        "spark.databricks.passthrough.adls.gen2.tokenProviderClassName"
    )
}

# Mount the Bronze container from ADLS to Databricks file system
# This mount point allows access to raw data from the Bronze layer
dbutils.fs.mount(
    source="abfss://bronze@devstoragecustomer.dfs.core.windows.net/",
    mount_point="/mnt/bronze",
    extra_configs=adls_configs
)

# Mount the Silver container from ADLS to Databricks file system
# This mount point allows access to cleansed and enriched data
```

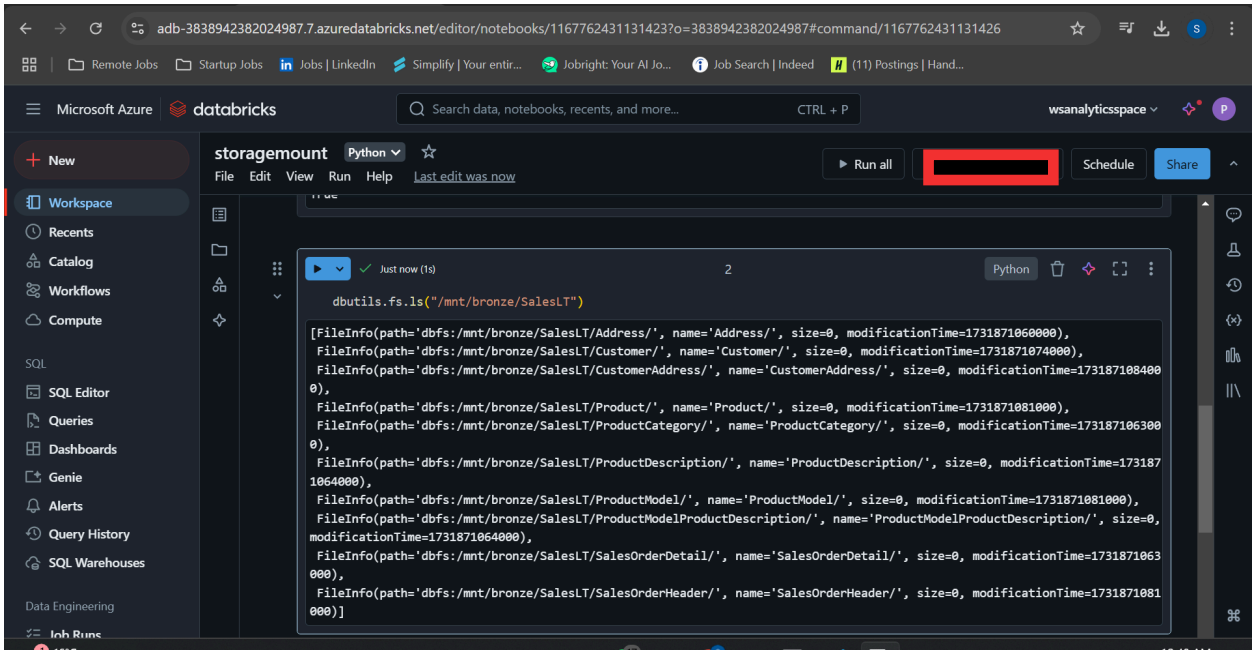
```
dbutils.fs.mount(
    source="abfss://silver@devstoragecustomer.dfs.core.windows.net/",
    mount_point="/mnt/silver",
    extra_configs=adls_configs
)

# Mount the Gold container from ADLS to Databricks file system
# This mount point allows access to business-level aggregated data
dbutils.fs.mount(
    source="abfss://gold@devstoragecustomer.dfs.core.windows.net/",
    mount_point="/mnt/gold",
    extra_configs=adls_configs
)
dbutils.fs.ls("/mnt/bronze")
```

Code Explanation:

1. **Configuration Setup**:
    - The code uses a dictionary to store configuration settings required for authentication and accessing ADLS, such as authentication type and token provider class.
2. **Mount ADLS Containers**:
    - The `dbutils.fs.mount` function is used to mount specific containers (e.g., bronze, silver, gold) from an ADLS Gen2 storage account to Databricks' file system. This allows seamless access to data in those containers via Databricks.
3. **Error Handling**:
    - The code includes logic to check for existing mounts to avoid errors when attempting to remount directories.
4. **Purpose**:
    - By mounting the storage containers, Databricks users can efficiently access and process data from ADLS without dealing directly with storage endpoints in each operation.

Once I run the notebook, I can see the following contents in the bronze layer



The next stage is to do transformations on the data which is in the bronze layer and promote the data to the silver layer.
In the transformations, I started with a single table column transformation and then I proceeded with multiple table transformations.
I used the below code

```
dbutils.fs.ls('mnt/bronze/SalesLT/')

dbutils.fs.ls('mnt/silver/') # check if the silver layer has any contents

# Load the Address data from the Bronze layer in Parquet format
address_df =
spark.read.format('parquet').load('/mnt/bronze/SalesLT/Address/Address.parquet')

# Display the contents of the Address DataFrame for a quick preview
display(address_df)

from pyspark.sql.functions import from_utc_timestamp, date_format
from pyspark.sql.types import TimestampType

# Convert the ModifiedDate column from UTC to a specific format (yyyy-MM-dd)
# 1. Cast the ModifiedDate column to TimestampType
# 2. Convert it to UTC timestamp using from_utc_timestamp
```

```
# 3. Format the timestamp to "yyyy-MM-dd" using date_format
address_df = address_df.withColumn(
    "ModifiedDate",
    date_format(

from_utc_timestamp(address_df['ModifiedDate'].cast(TimestampType()),
"UTC"),
        "yyyy-MM-dd"
    )
)

# Display the contents of the Address DataFrame
display(address_df)


# Initialize an empty list to store extracted table identifiers
table_list = []

# List all files and directories in the specified Bronze layer path
for item in dbutils.fs.ls('mnt/bronze/SalesLT/'):
    # Extract the table name by splitting the directory/file name and
append to the list
    table_list.append(item.name.split('/')[0])

# Display the list of table identifiers
table_list

from pyspark.sql.functions import from_utc_timestamp, date_format
from pyspark.sql.types import TimestampType

# Iterate through the list of tables
for i in table_list:
    # Define the path to the source parquet file in the bronze layer
    path = '/mnt/bronze/SalesLT/' + i + '/' + i + '.parquet'

    # Load the parquet file into a DataFrame
    df = spark.read.format('parquet').load(path)

    # Get the column names of the DataFrame
    column = df.columns
```

```
    # Check each column for "Date" or "date" and reformat it
    for col in column:
        if "Date" in col or "date" in col:
            # Convert column to UTC timestamp and reformat to "yyyy-MM-dd"
            df = df.withColumn(col,
date_format(from_utc_timestamp(df[col].cast(TimestampType()), "UTC"),
"yyyy-MM-dd"))

    # Define the output path for the silver layer in Delta format
    output_path = '/mnt/silver/SalesLT/' + i + '/'

    # Write the DataFrame to the silver layer in Delta format with
overwrite mode
    df.write.format('delta').mode('overwrite').save(output_path)

# display the results
display(df)
```

Code Explanation:

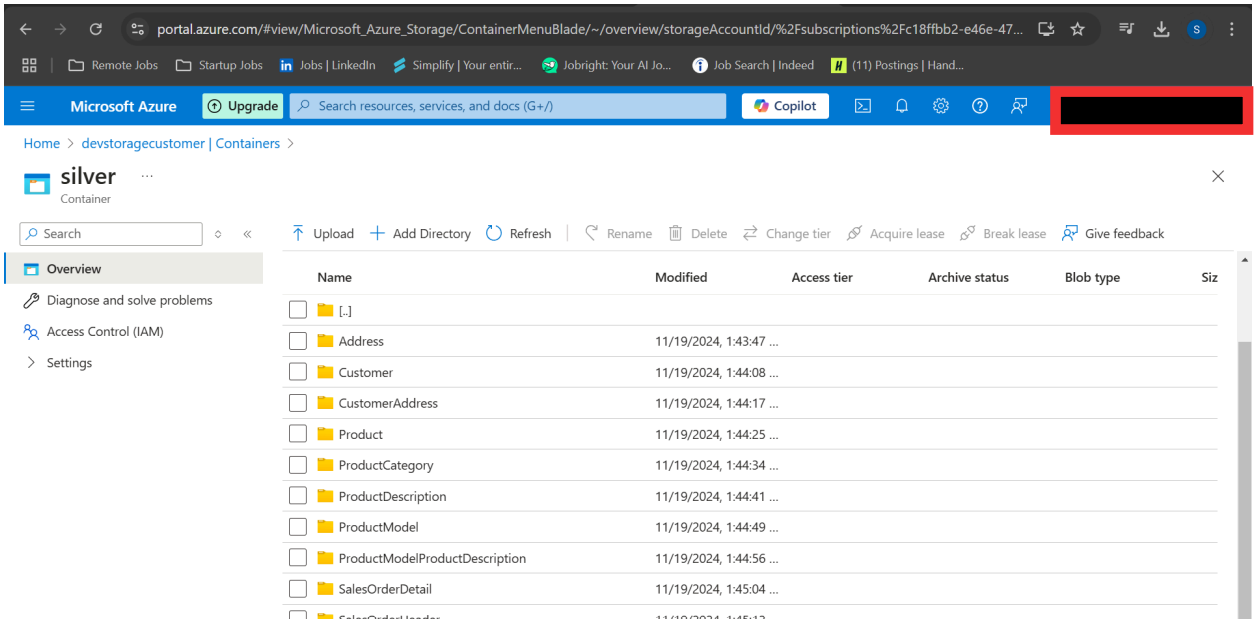
1. **Check Bronze and Silver Layers**:
    - The `dbutils.fs.ls` command lists the contents of the directories in the Bronze and Silver layers to verify their structure and contents.
2. **Load and Preview Address Data**:
    - The `address_df` DataFrame loads the "Address" table from the Bronze layer (in Parquet format).
    - The `display(address_df)` command previews the contents of the loaded table.
3. **Date Column Transformation for `ModifiedDate`**:
    - The `ModifiedDate` column in `address_df` is:
        - Cast to `TimestampType`.
        - Converted to a UTC timestamp using `from_utc_timestamp`.
        - Reformatted to "yyyy-MM-dd" using `date_format`.
4. **Extract Table Names from the Bronze Layer**:
    - The code lists all files/directories in the Bronze layer.
    - Extracts table names by splitting the directory names and stores them in the `table_list` list.
5. **Process All Tables from the Bronze Layer**:
    - Iterates through the `table_list`.

- Loads each table as a DataFrame from the Bronze layer.
- Checks for columns containing "Date" or "date" in their names.
- Reformats these columns to "yyyy-MM-dd" in UTC.

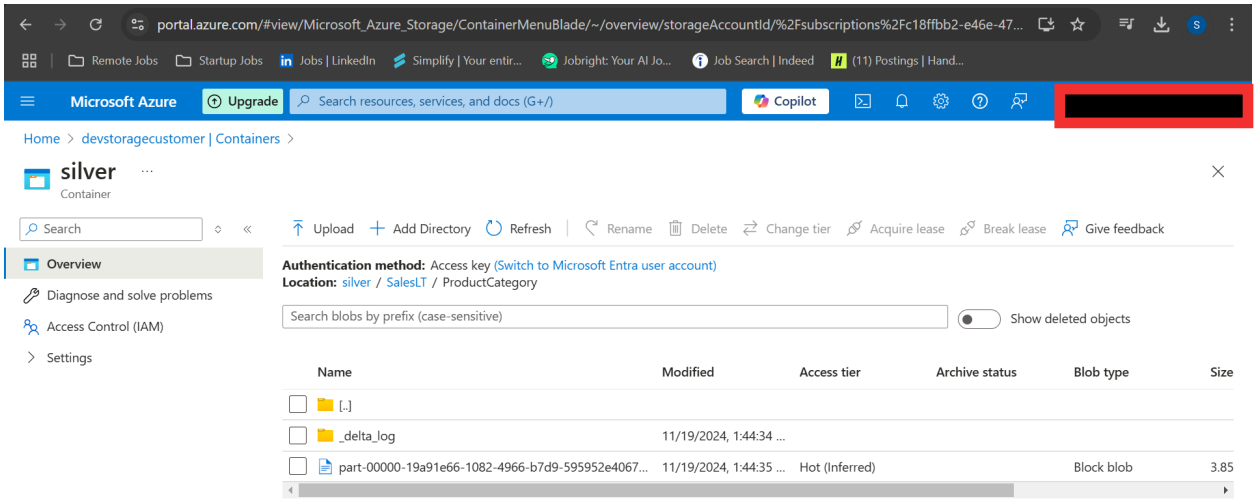
6. **Write Transformed Data to Silver Layer**:
   - Writes the transformed DataFrame in Delta format to the Silver layer at the corresponding path.
   - Uses `mode('overwrite')` to replace existing files.
7. **Preview Final DataFrame**:
   - The final DataFrame (`df`) from the last iteration is displayed using the `display(df)` command.

This code processes and migrates data from the Bronze layer (raw Parquet files) to the Silver layer (cleaned and standardized Delta files) while ensuring date columns are properly formatted.

I can see the contents in the silver layer after the above code is executed in databricks



In the silver layer, the data is stored in delta format as shown below



The next step is to create a gold layer from the silver layer.
I used the below code to generate the gold layer.

```
# List all files and directories in the specified path in the Databricks File System (DBFS)
# 'mnt/silver/SalesLT/' refers to a mounted directory in DBFS, often linked to external storage
dbutils.fs.ls('mnt/silver/SalesLT/')

dbutils.fs.ls('mnt/gold/') # list the files and directories in the gold layer

# Reading a Delta table into a Spark DataFrame
# '/mnt/silver/SalesLT/Address/' specifies the path to the Delta table in the Silver layer
df = spark.read.format('delta').load('/mnt/silver/SalesLT/Address/')

display(df)

def convert_column_names_to_snake_case(input_df):
    """
    Convert column names in a PySpark DataFrame from PascalCase or camelCase to snake_case.

    Args:
```

```
        input_df (DataFrame): The input PySpark DataFrame with columns to
be converted.

    Returns:
        DataFrame: A new DataFrame with column names converted to
snake_case.
    """
    # Fetch the list of original column names
    original_column_names = input_df.columns

    # Dictionary to store the mapping of old to new column names
    column_name_mapping = {}

    # Iterate over the column names
    for original_name in original_column_names:
        # Generate the snake_case version of the column name
        snake_case_name = "".join([
            "_" + character.lower() if (
                character.isupper()                  # Check if the
character is uppercase
                and index > 0                        # Ensure it's not the
first character
                and not original_name[index - 1].isupper()  # Ensure the
previous character isn't uppercase
            ) else character.lower()  # Convert the character to lowercase
otherwise
            for index, character in enumerate(original_name)
        ]).lstrip("_")  # Remove leading underscore, if any

        # Ensure the new column name is unique
        if snake_case_name in column_name_mapping.values():
            raise ValueError(f"Duplicate column name found after renaming:
'{snake_case_name}'")

        # Map the original column name to the new snake_case column name
        column_name_mapping[original_name] = snake_case_name

    # Apply the column name mapping to the DataFrame
    for old_name, new_name in column_name_mapping.items():
        input_df = input_df.withColumnRenamed(old_name, new_name)
```

```
    return input_df

df = convert_column_names_to_snake_case(df)

display(df)

#transformations on multiple tables
# Initialize an empty list to store table names
table_names_list = []

# Iterate over the file system objects in the specified directory
for file_info in dbutils.fs.ls('mnt/silver/SalesLT'):
    # Append each file or folder name to the list
    table_names_list.append(file_info)

# Display the collected table names
table_names_list


# Initialize an empty list to store the table names
table_names = []

# Iterate over the file system objects in the specified directory
for file_object in dbutils.fs.ls('mnt/silver/SalesLT'):
    # Extract the name of the file or folder, removing any trailing
slashes, and add it to the list
    table_names.append(file_object.name.split('/')[0])

# Display the list of table names
table_names


# Iterate over each table name in the list
for table_name in table_names:
    # Construct the input file path for the current table
    input_path = '/mnt/silver/SalesLT/' + table_name
    print(f"Reading data from: {input_path}")

    # Read the Delta table into a DataFrame
```

```
    input_df = spark.read.format('delta').load(input_path)

    # Convert column names to snake_case
    df = convert_column_names_to_snake_case(input_df)

    # Construct the output file path for the transformed table
    output_path = '/mnt/gold/SalesLT/' + table_name + '/'
    print(f"Writing transformed data to: {output_path}")

    # Write the transformed DataFrame back to Delta format with overwrite
mode
    df.write.format('delta').mode('overwrite').save(output_path)

display(df)
```

List Files:

- Shows the files and folders in the Silver and Gold layers (`mnt/silver/SalesLT/` and `mnt/gold/`).

Read a Delta Table:

- Loads a specific table (`Address`) from the Silver layer into a Spark DataFrame.

Standardize Column Names:

- A function (`convert_column_names_to_snake_case`) is used to rename DataFrame columns to `snake_case` for consistency.

Get All Table Names:

- Collects all table names in the Silver layer directory (`mnt/silver/SalesLT/`).

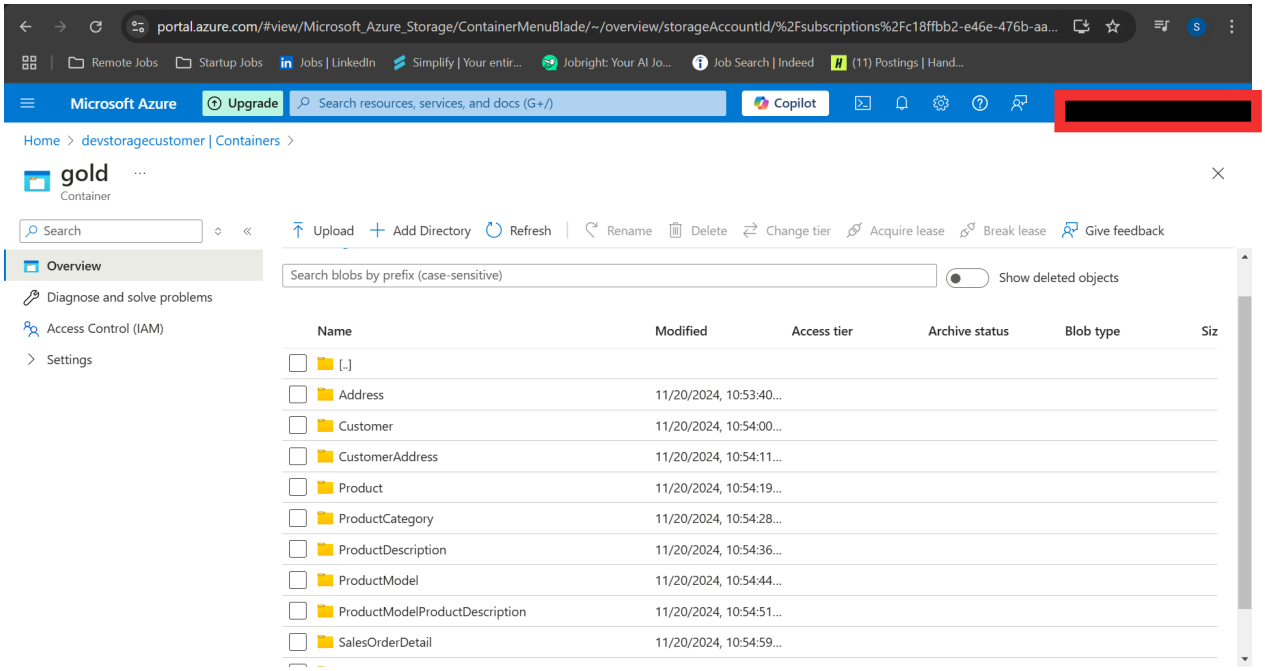
Transform and Save Data:

- For each table in the Silver layer:
    - Reads the data into a DataFrame.
    - Converts column names to `snake_case`.
    - Saves the transformed data as a Delta table in the Gold layer (`mnt/gold/SalesLT/`).

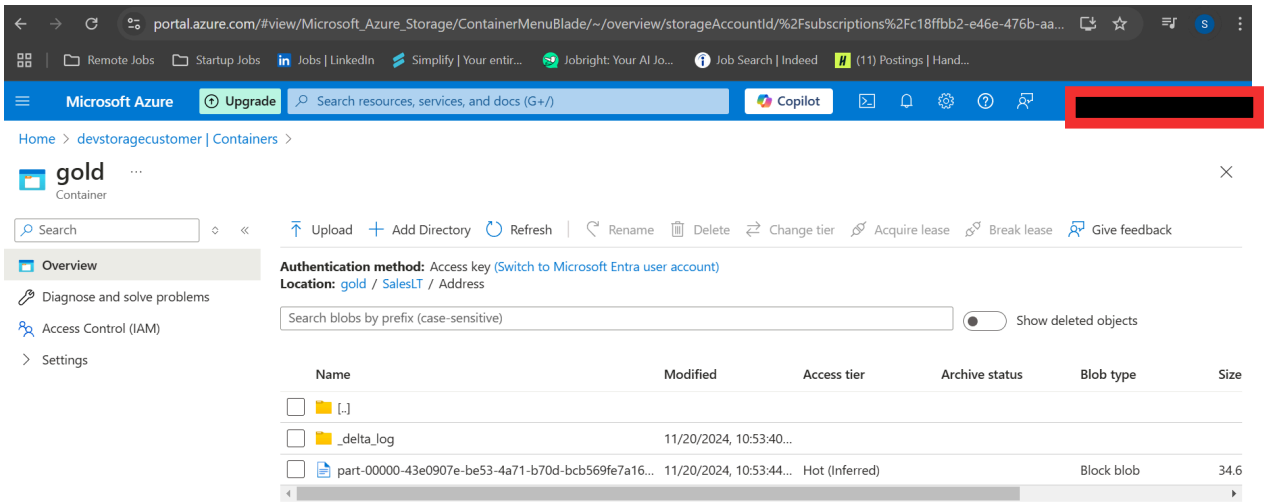
Display Transformed Data:

- Displays the final transformed DataFrame to verify the output.

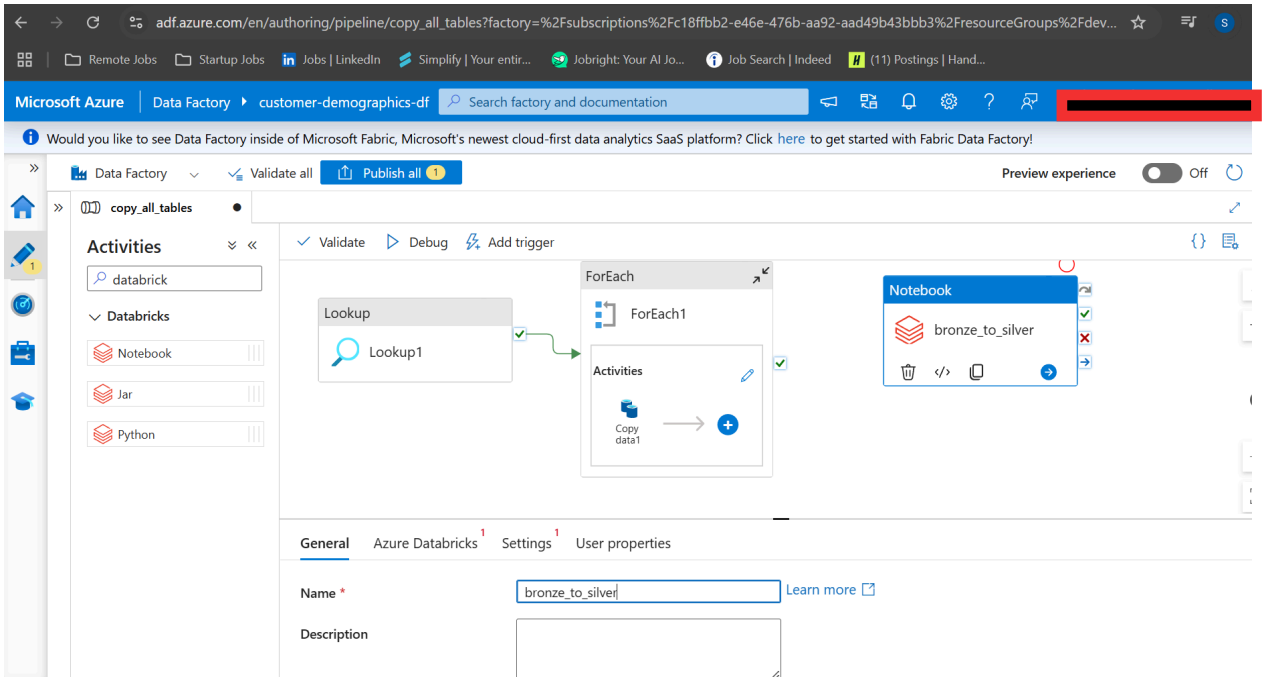
I can also find the files in the gold containers as below.



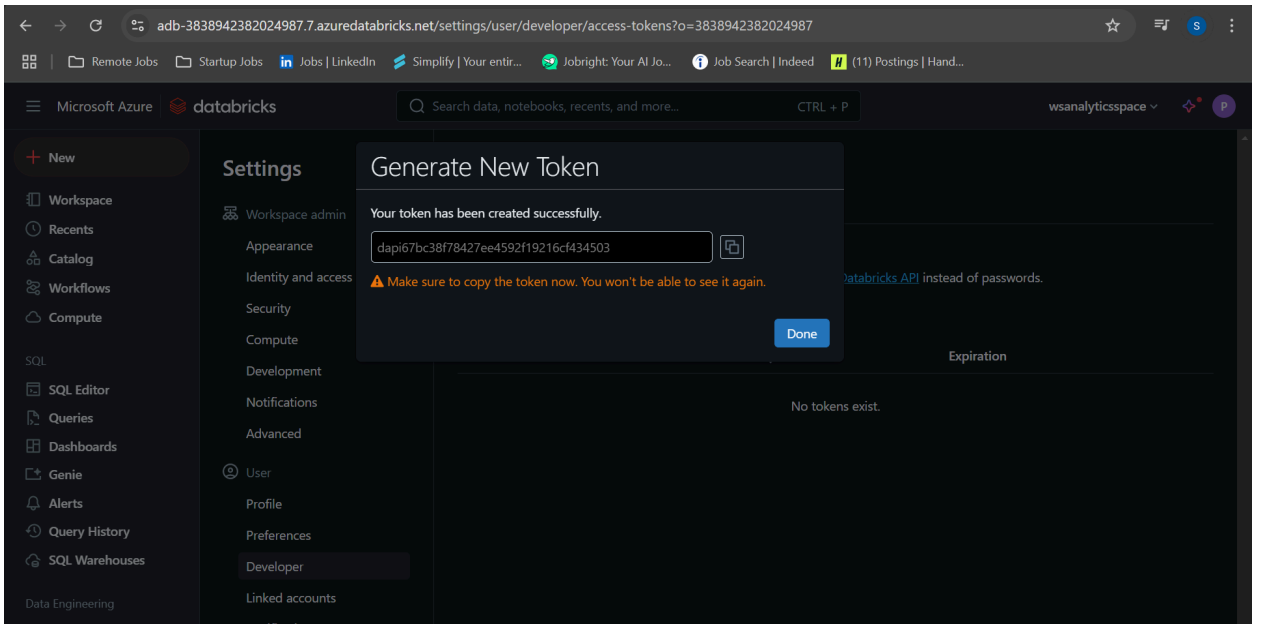
In the gold layer, the files are stored in delta format as shown below



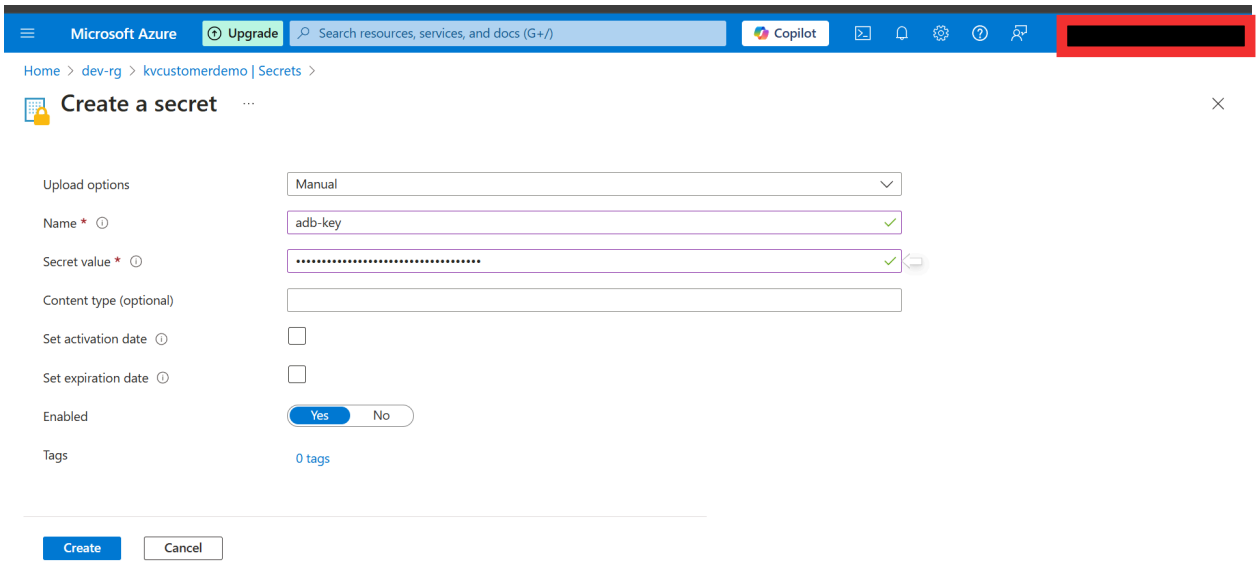
All the above transformations need to be performed as a part of the data pipeline I have created earlier in the data factory. So, the notebook needs to be a part of the pipeline. Attach the notebook to ForEach Activity in the Azure Data Factory.



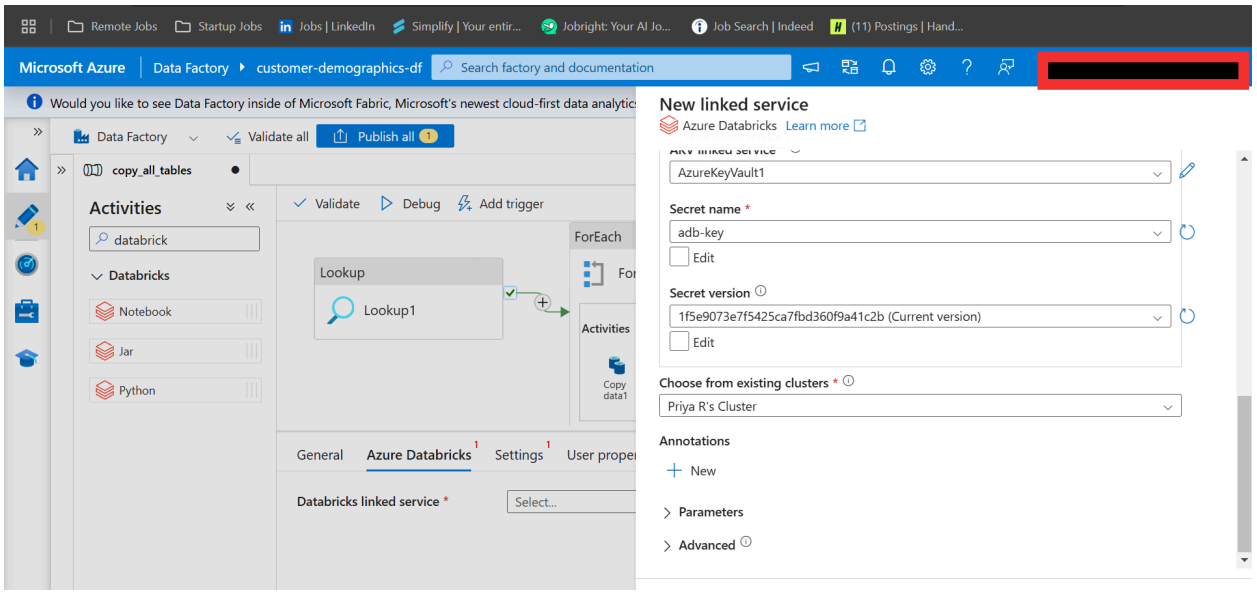
Now, I created a Linked Service for Databricks. Used AutoResolveIntegrationRuntime as I connected to a service within Azure. For the cluster, I have chosen “Existing Interactive Cluster”. For the Authentication Type, I have chosen “Azure Key Vault”. Access tokens can be generated from Developer settings in databricks. These tokens are stored as secrets in the azure key vault. Azure Key vault can be leveraged for authentication purposes.



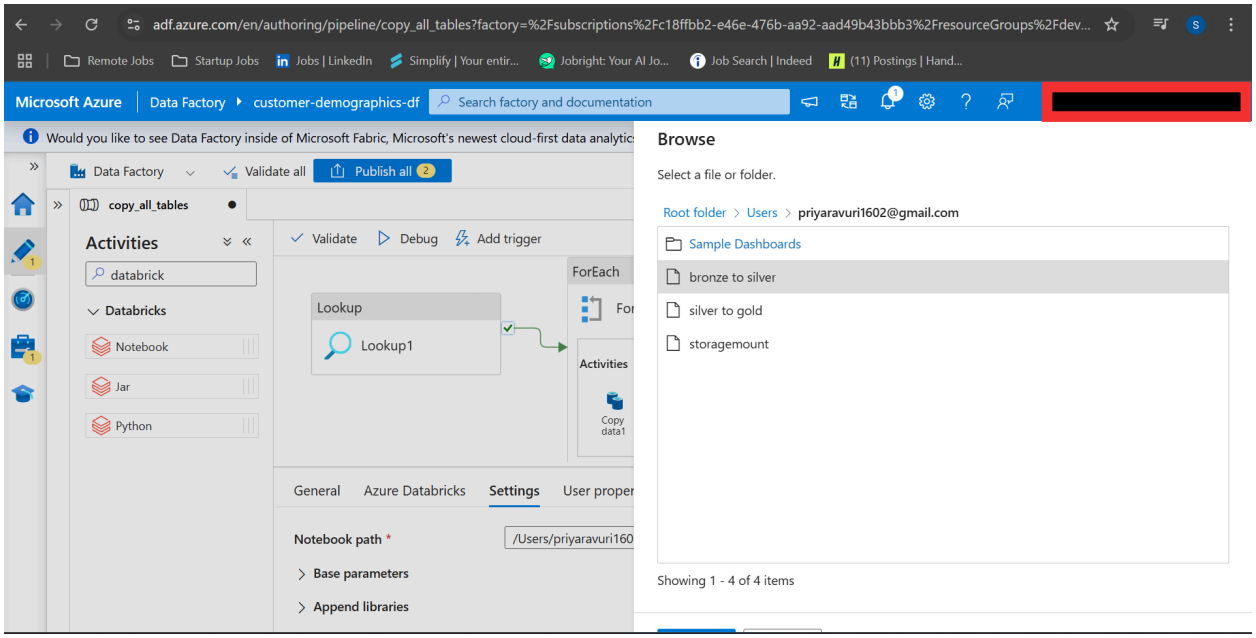
As the token can’t be seen again, I created a secret in the Azure Key Vault as shown below



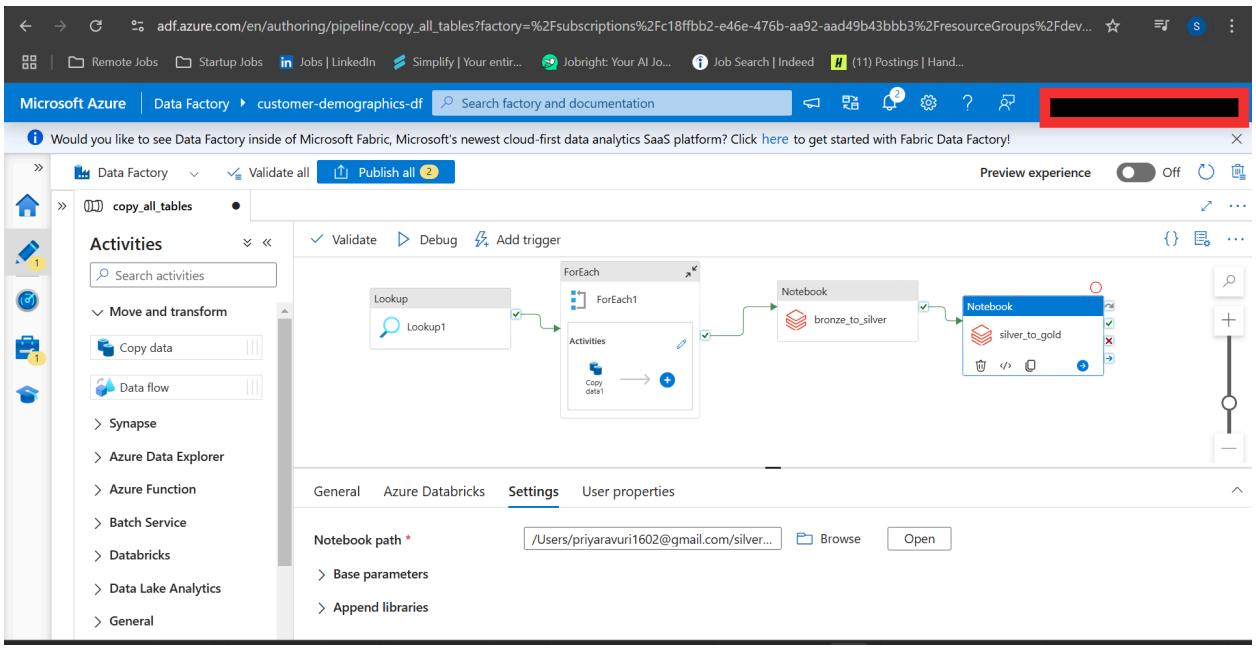
Now, in the linked service authentication, I have selected the Azure Key vault which I have created earlier and also the secret I generated with the databricks access token stored in it. I have chosen the existing cluster in the cluster options as shown below



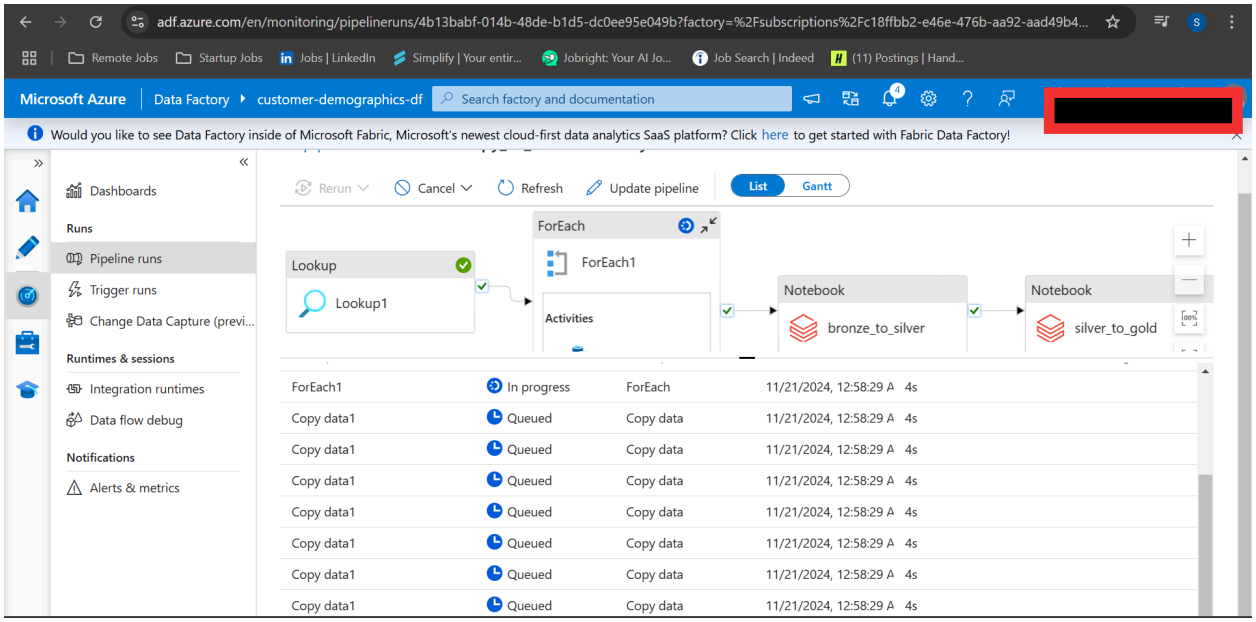
In the Settings, I have chosen the notebook path which points to the “bronze to silver” notebook in the databricks workspace as shown below

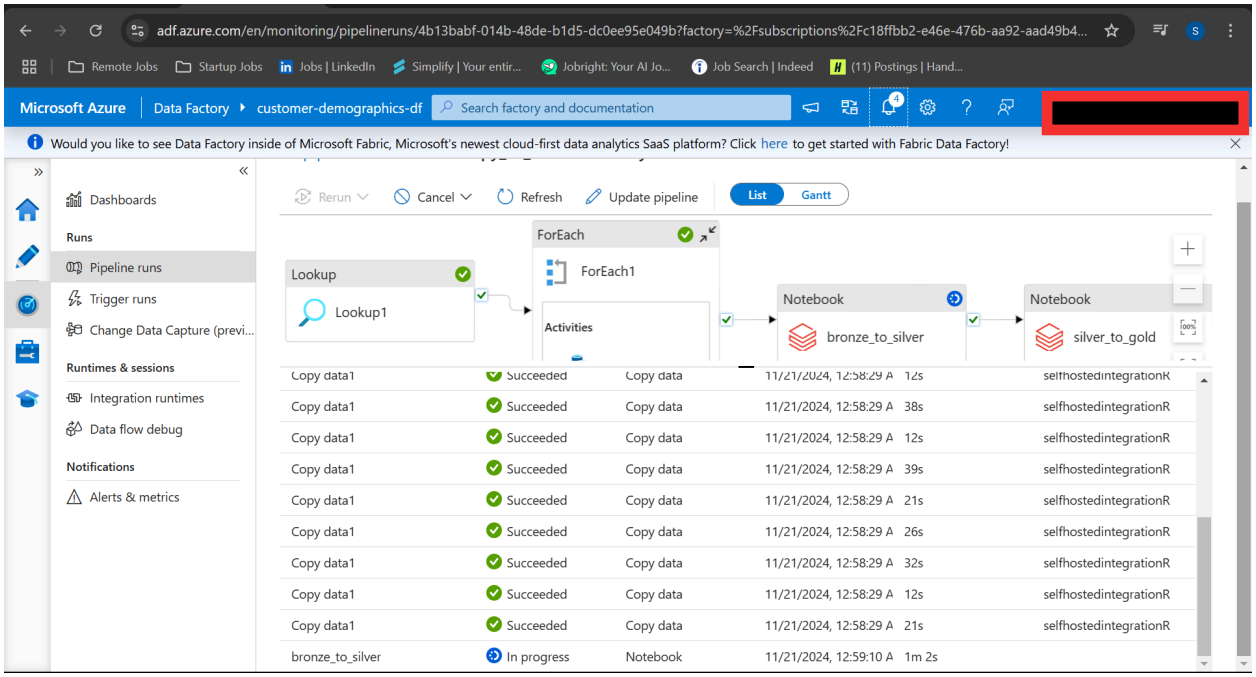


Repeat the above steps in order to add one more notebook “silver to gold” to the data pipeline. After adding the “silver to gold” notebook to the pipeline, the ADF looks like below. Once all the pipeline settings are configured, I published the pipeline.



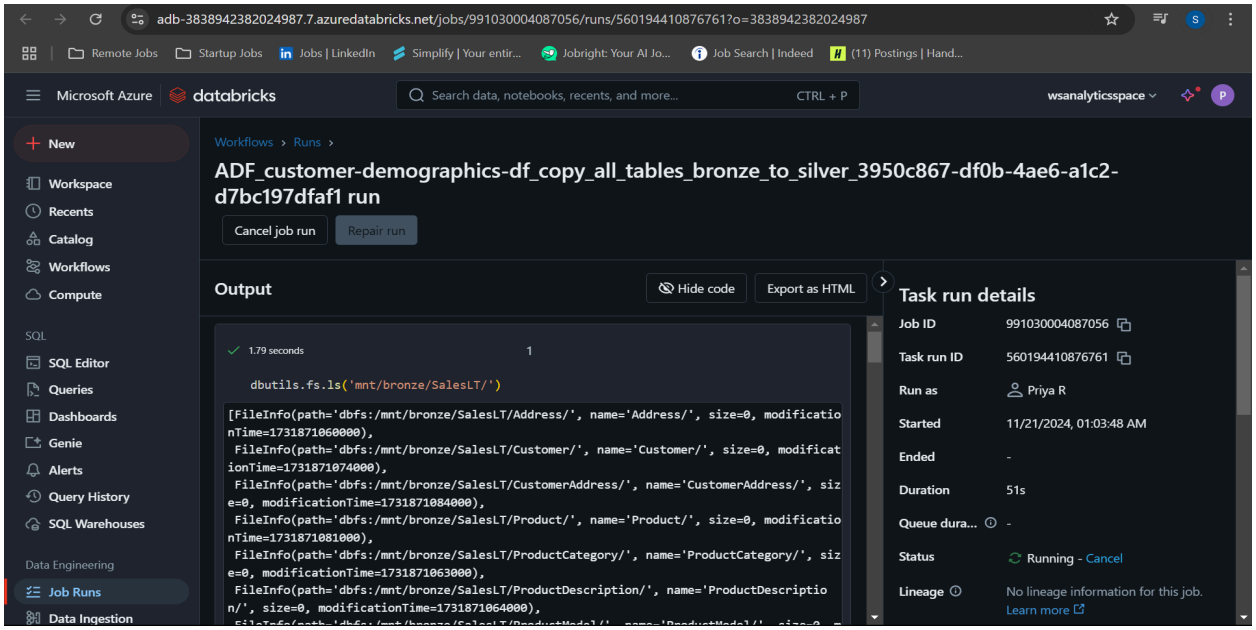
I triggered the pipeline using an on-demand trigger and I can see that the pipeline activities are queued.



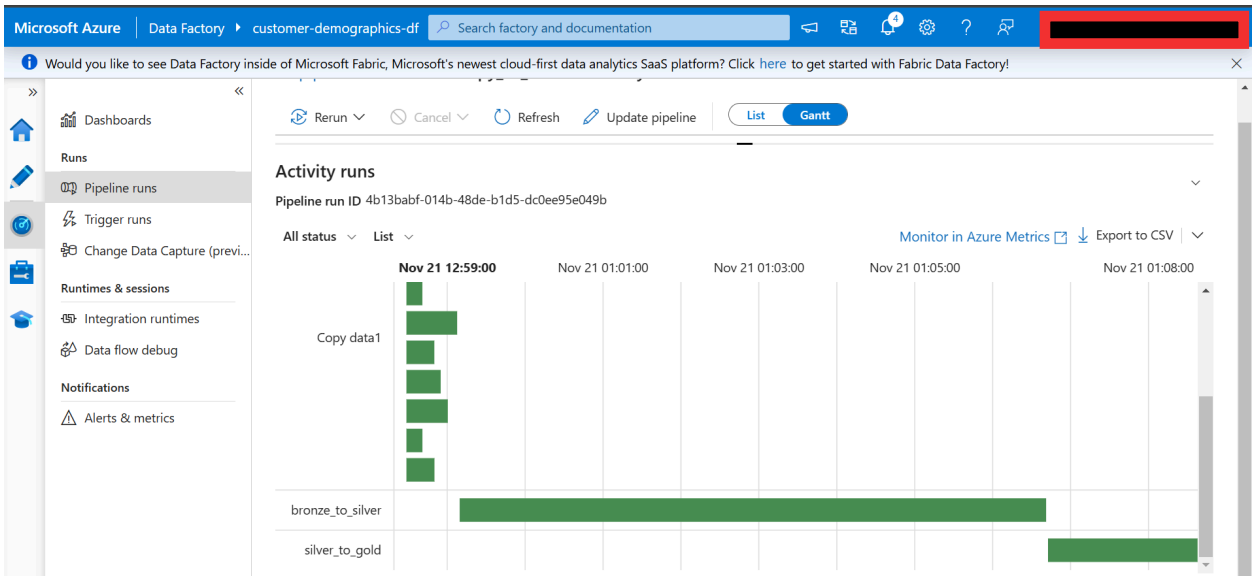


Note that the time taken might be long if it takes longer time for the cluster to start

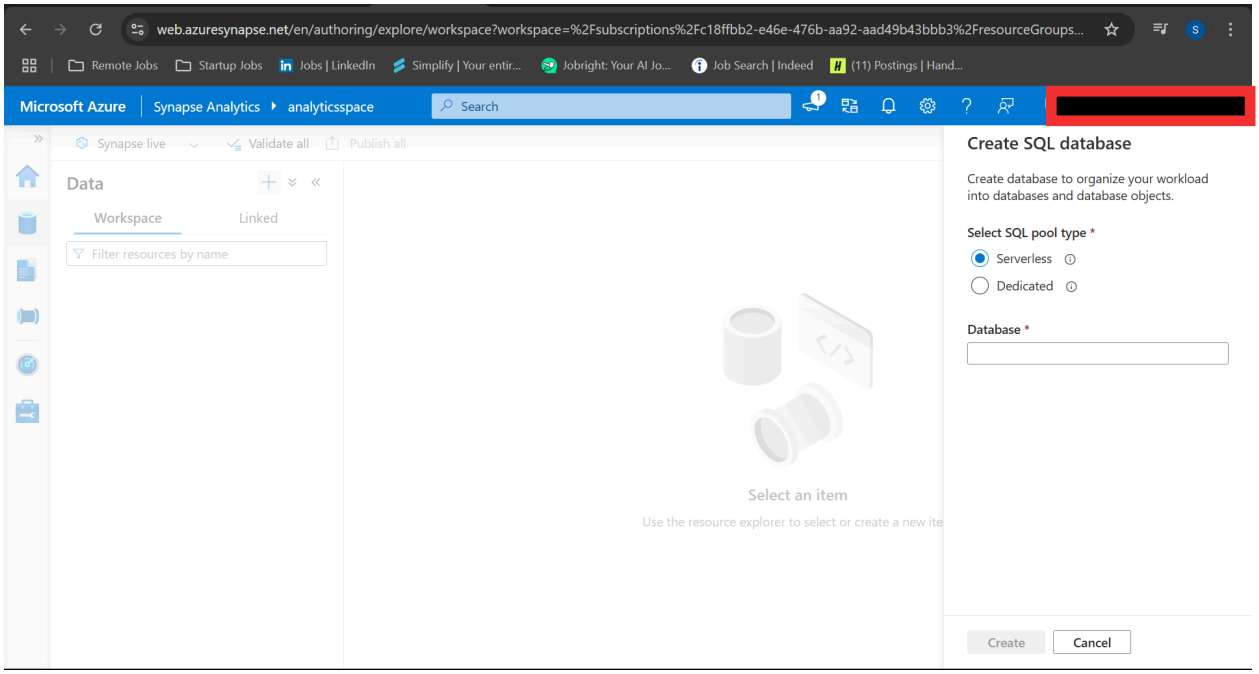
I can monitor the execution as shown below



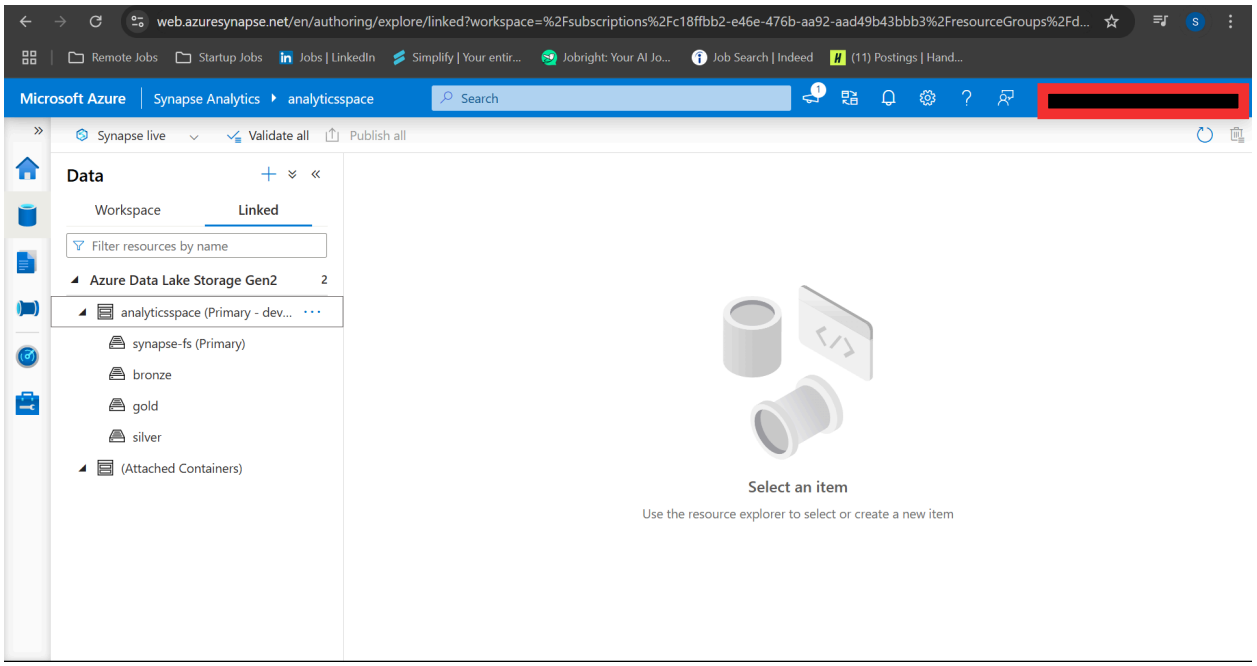
The same progress can be tracked using the job runs in databricks. Also, the timeline of the activities can be monitored from Gantt Charts in Azure Data Factory



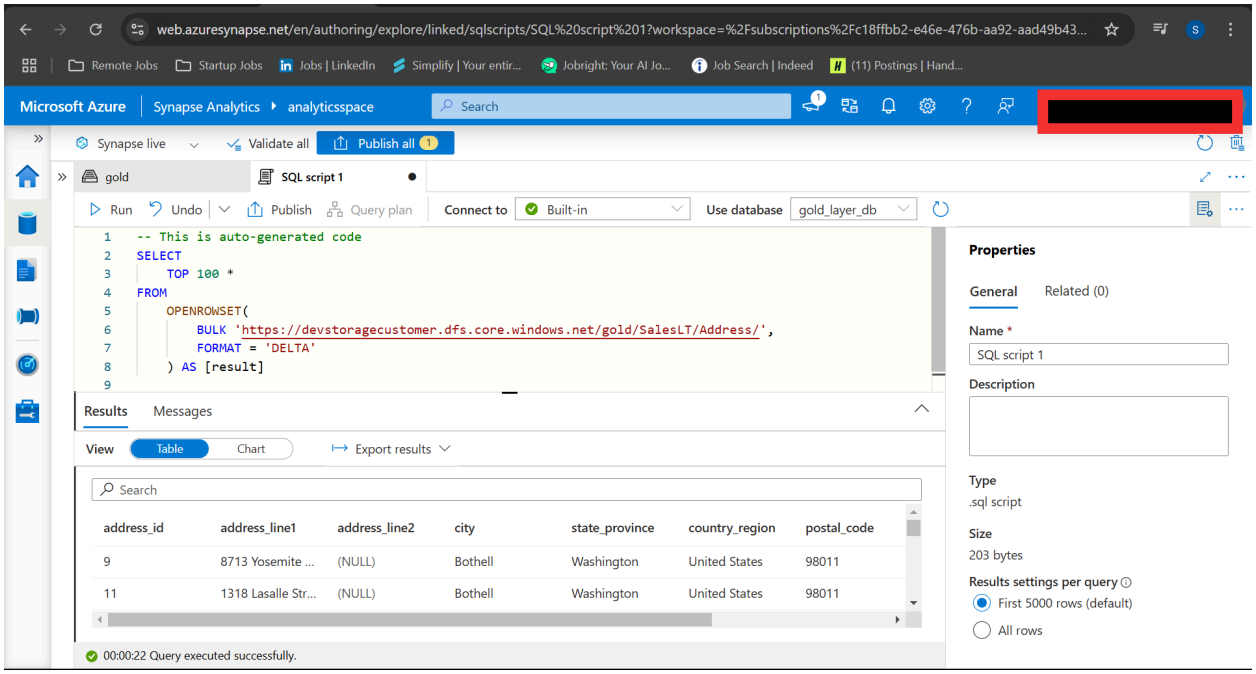
The next step is to embed Azure Synapse Analytics into the data pipeline. Azure Synapse Analytics can ingest, store, and analyze data from various sources. It can also be used to create machine learning models. In the Azure Synapse resource which I have created earlier, I created a SQL database to organize the workload. Here, I have two options to create a SQL database. Serverless and Dedicated. Serverless is a compute option (Azure manages this SQL pool relative to the workload). Dedicated is both compute and storage option. I have chosen the serverless option and created a database name with “gold_layer_db”.



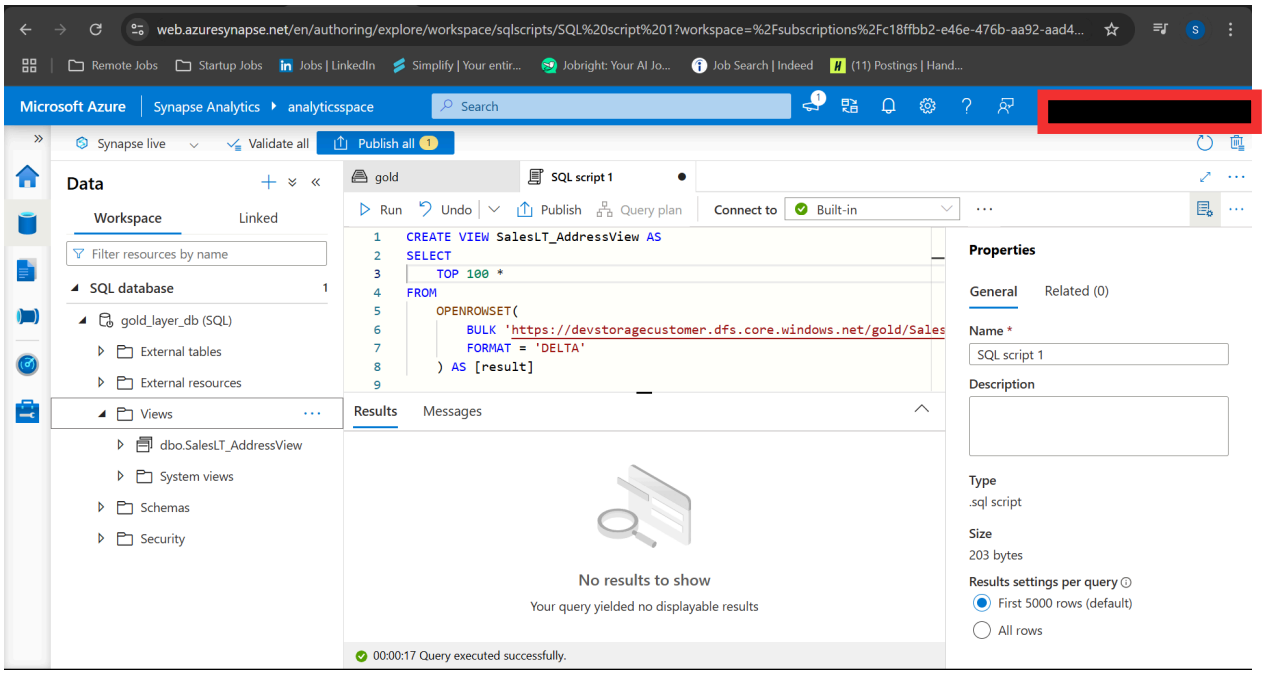
In the “Linked” section, I can see that the Azure Data Lake storage is already linked to synapse studio.



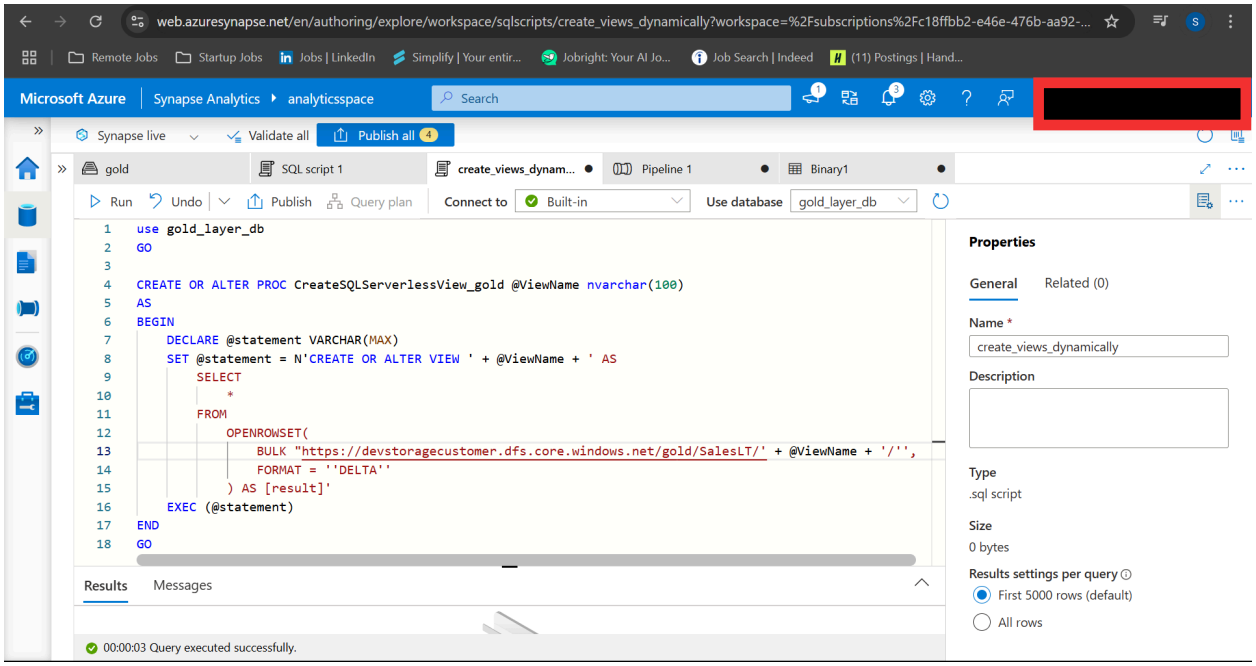
For an example query, I previewed the top 100 rows from the “Address” Table.



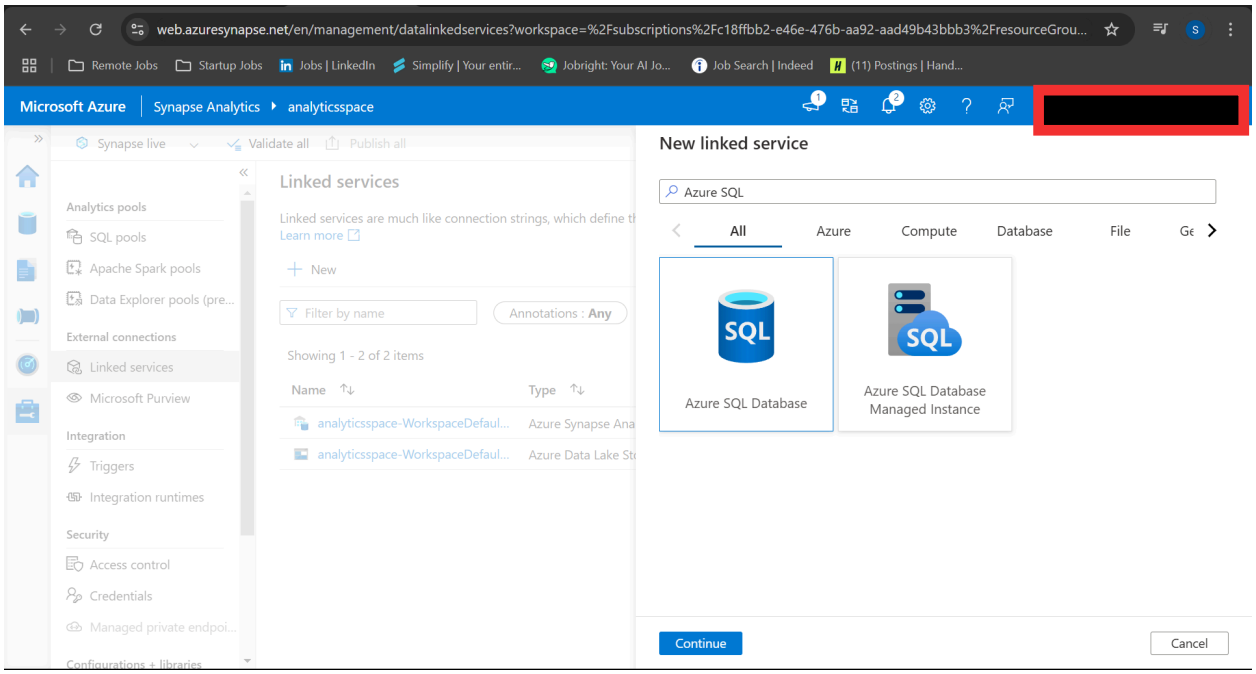
Now, I created a view on top of this result to register a view. I can find this view in the workspace as below



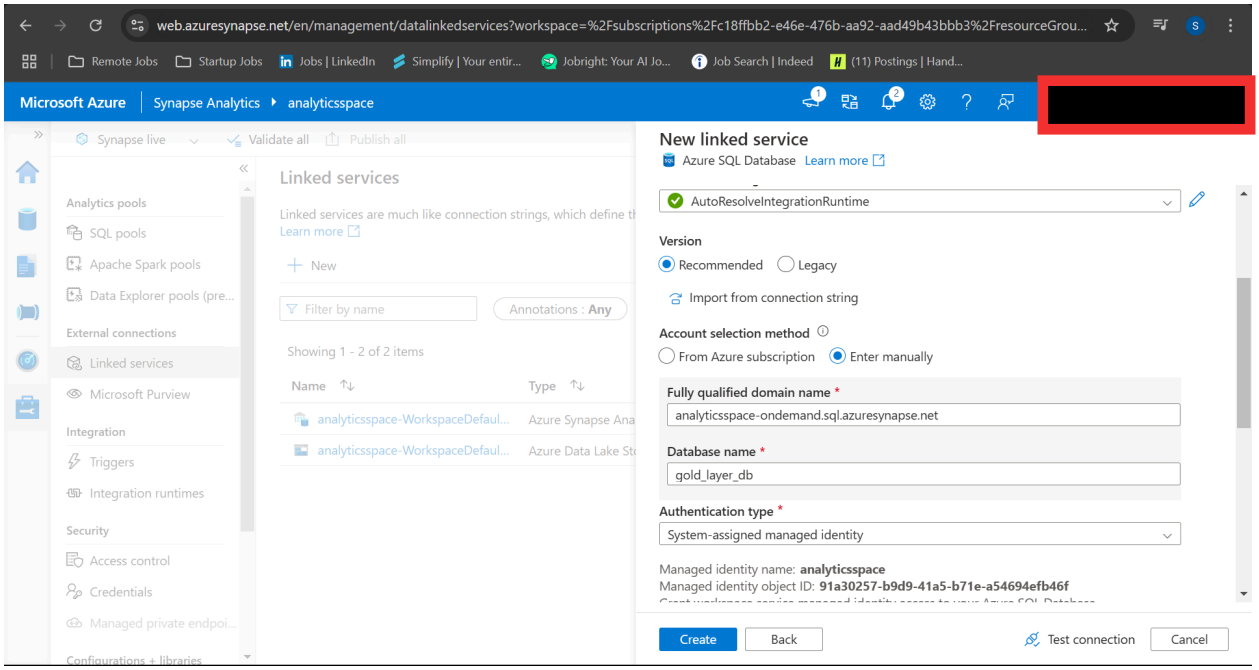
To create the views for each table dynamically, I used the following query in Azure Synapse



After publishing the script, I created a pipeline that uses the above stored procedure. Before that, I created a linked service to connect this serverless SQL DB.

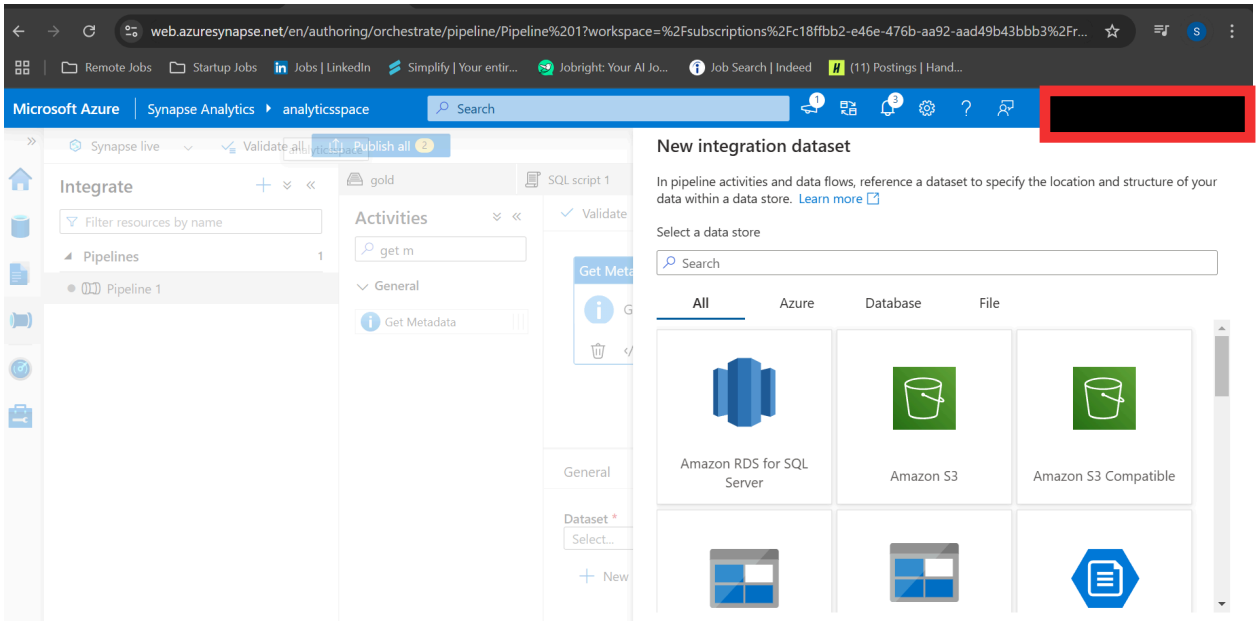


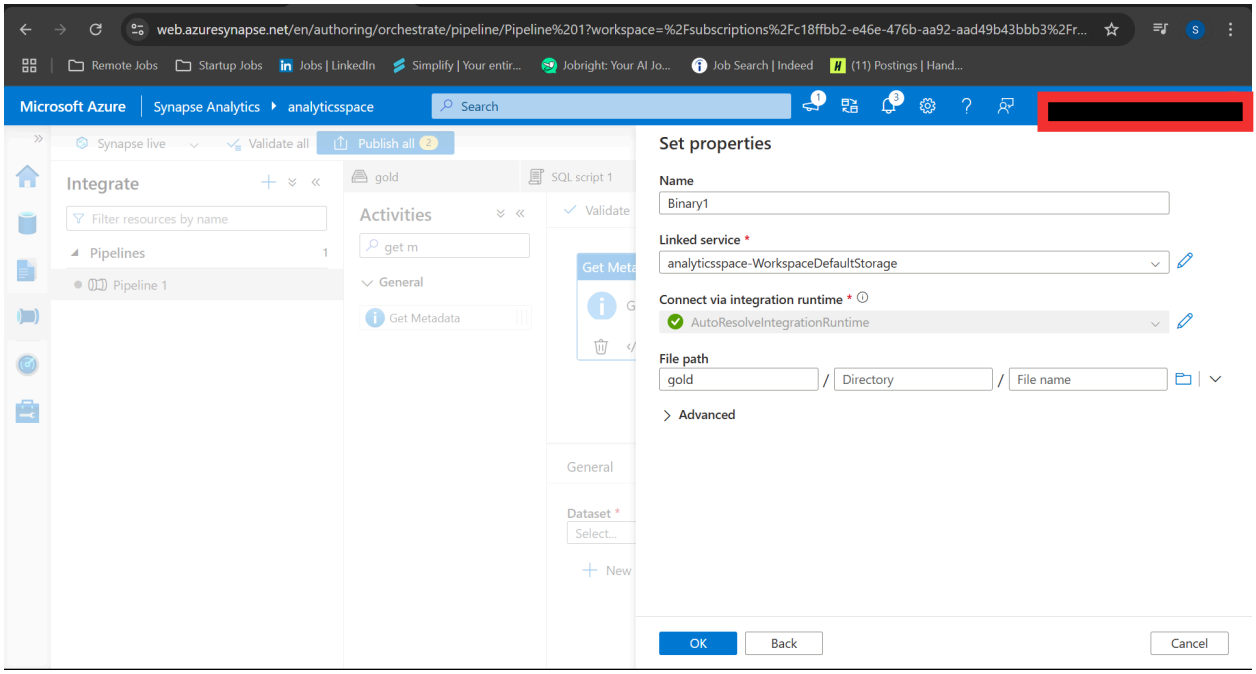
I have chosen “Autoresolve Integration Runtime” as I am connecting with another service in Azure. In the Account selection method, I chose to give the details manually. For the fully qualified domain name, I copied the details from the “serverless SQL endpoint” from “settings” of workspace properties. For the database name, I have given “gold_layer_db” as I am currently using this database. For the authentication, I have selected “System Assigned Managed Identity” . This uses my email address to connect to the database (gold_layer_db)

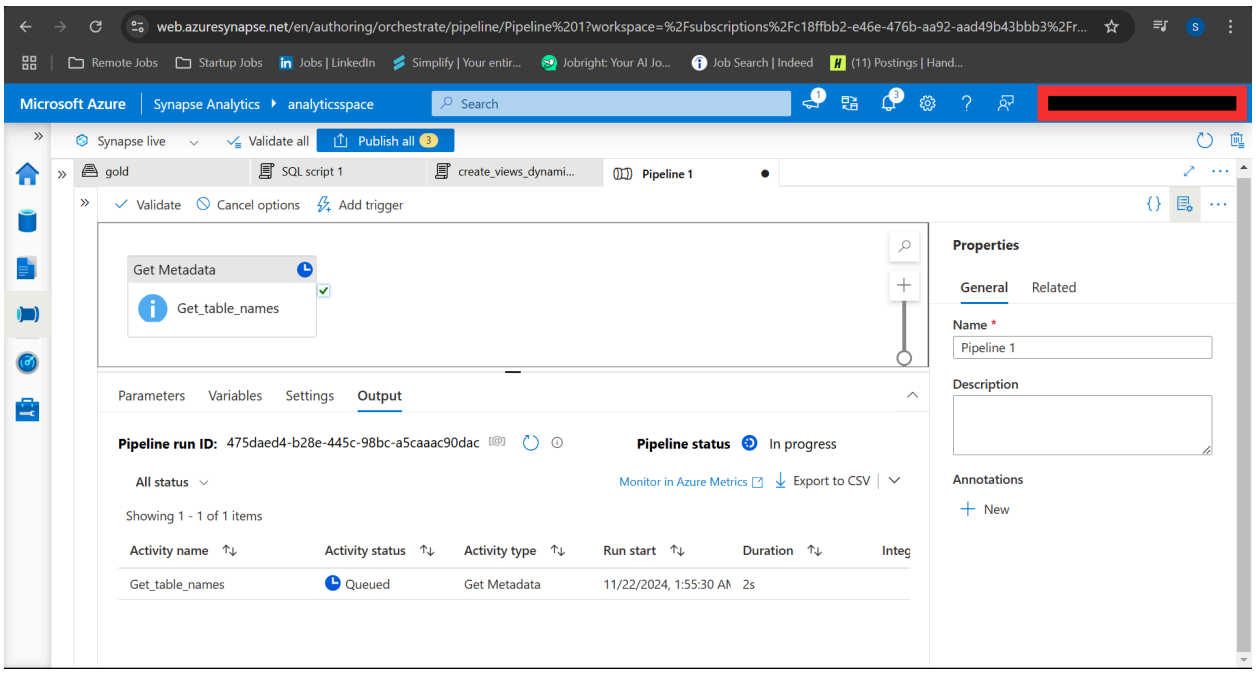


I can use this linked service to access the stored procedure from the pipeline.

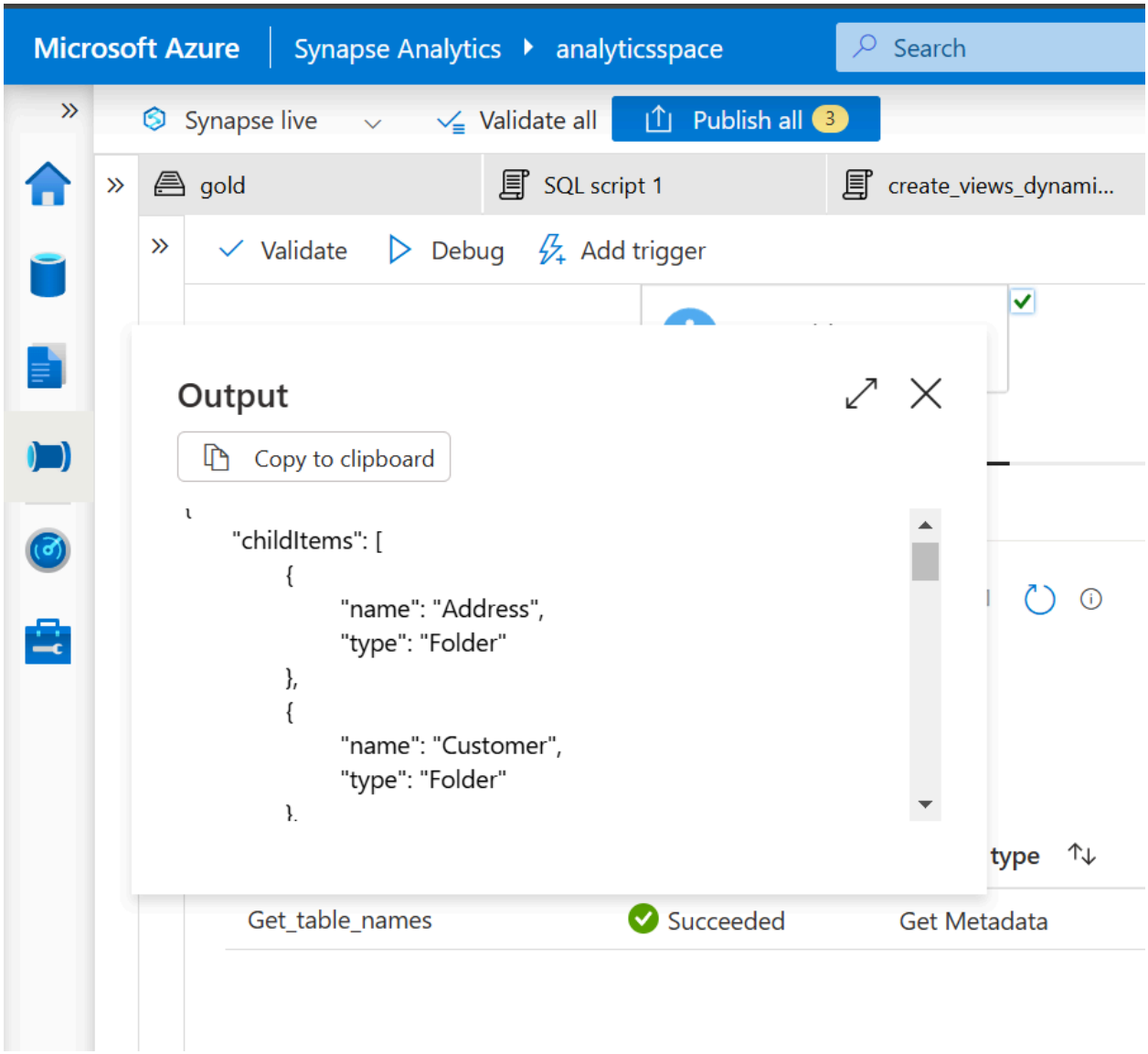
Then, I created a pipeline using the “integrate” option in synapse studio. Here I had to add a new dataset. As there was no existing dataset, I created a new dataset by selecting “Azure Data Lake storage”. The format of the file would be “binary”. The path would be pointing to the gold layer in the container.





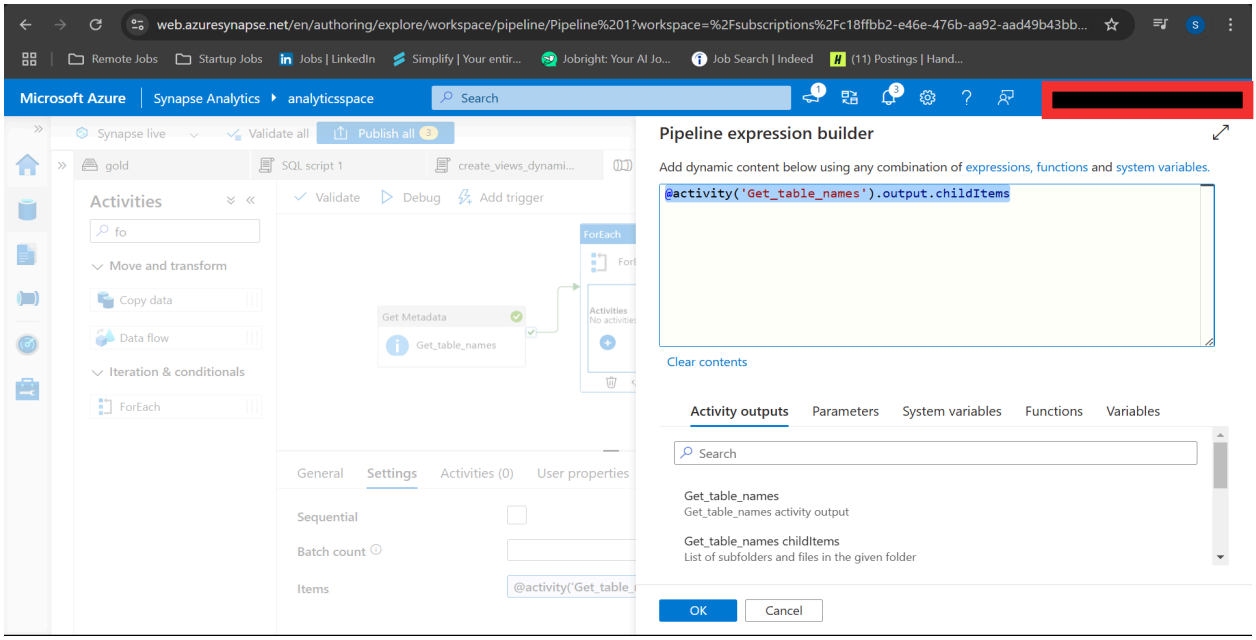


The output JSON after debugging will look like below

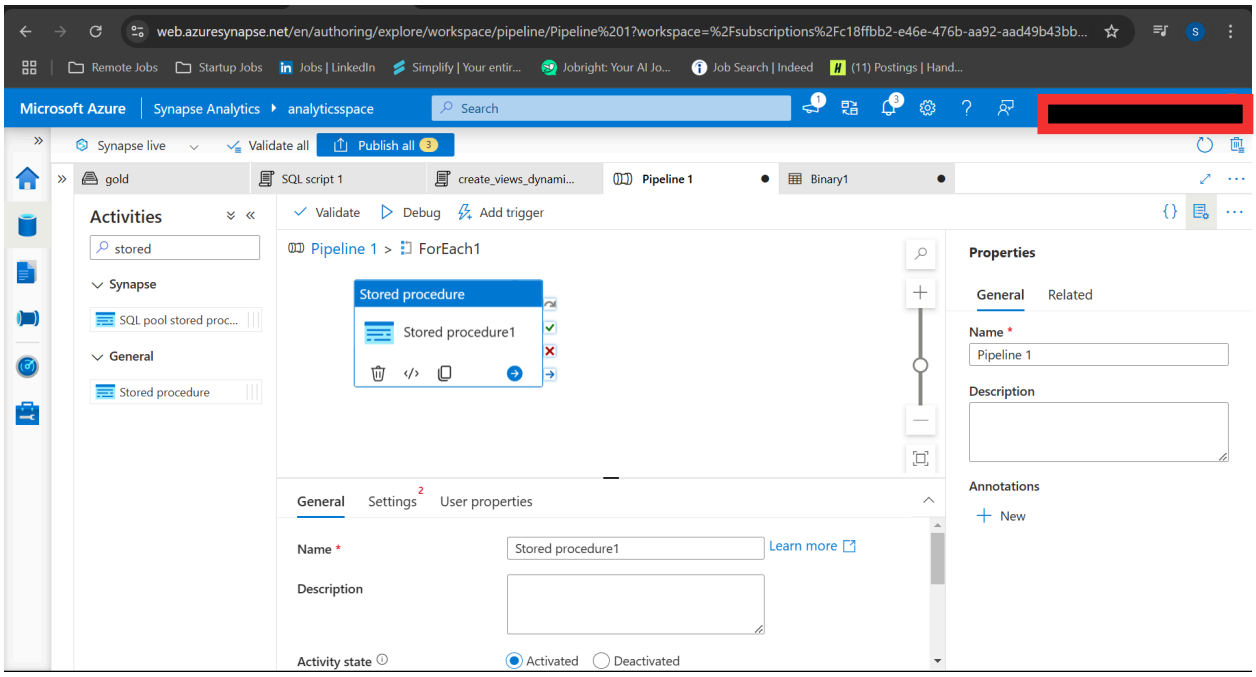


Now, I have the list of the table names that I can use. Now, I added a for each activity as I need to create a view for each table which was in the list. I need the “For Each” activity to run on the success of the “Get Metadata” activity. In the settings of the “For Each” activity, choose “Add Dynamic Content” and select “Get_table_names childitems”. The pipeline expression builder contains the following expression

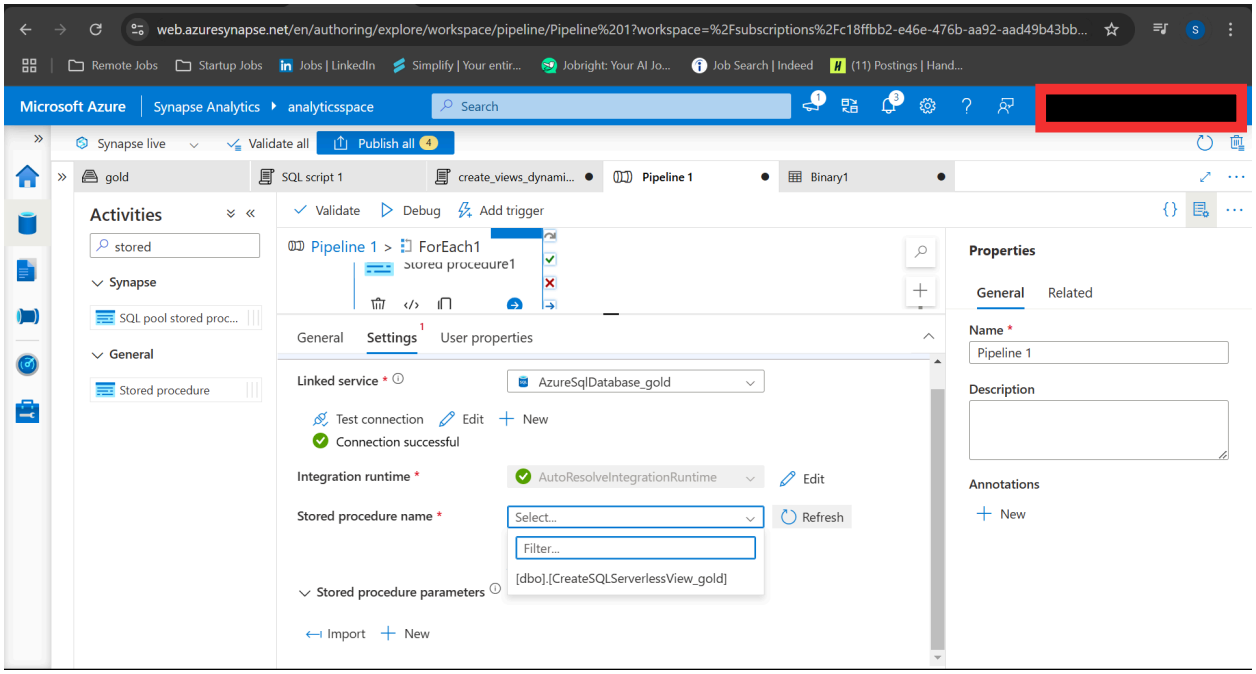
```
@activity('Get_table_names').output.childItems
```



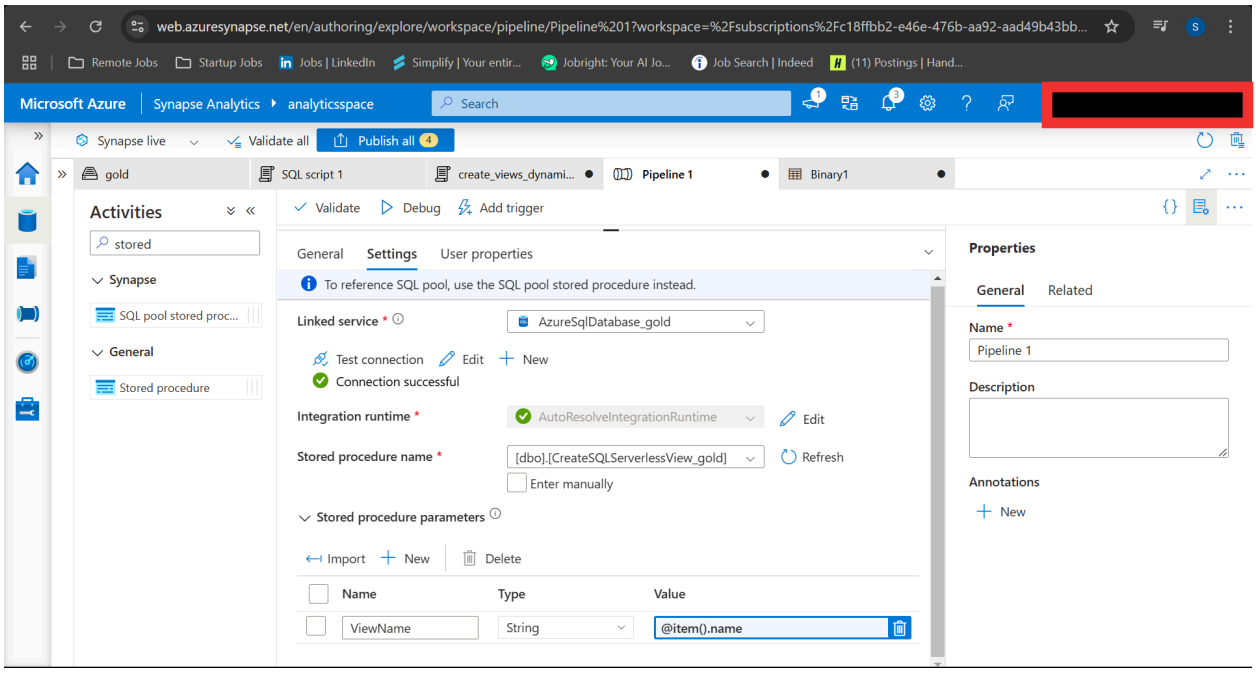
Now, I created a new activity within the foreach activity to call the stored procedure for each table. so,I embedded a stored procedure in the “foreach” activity.

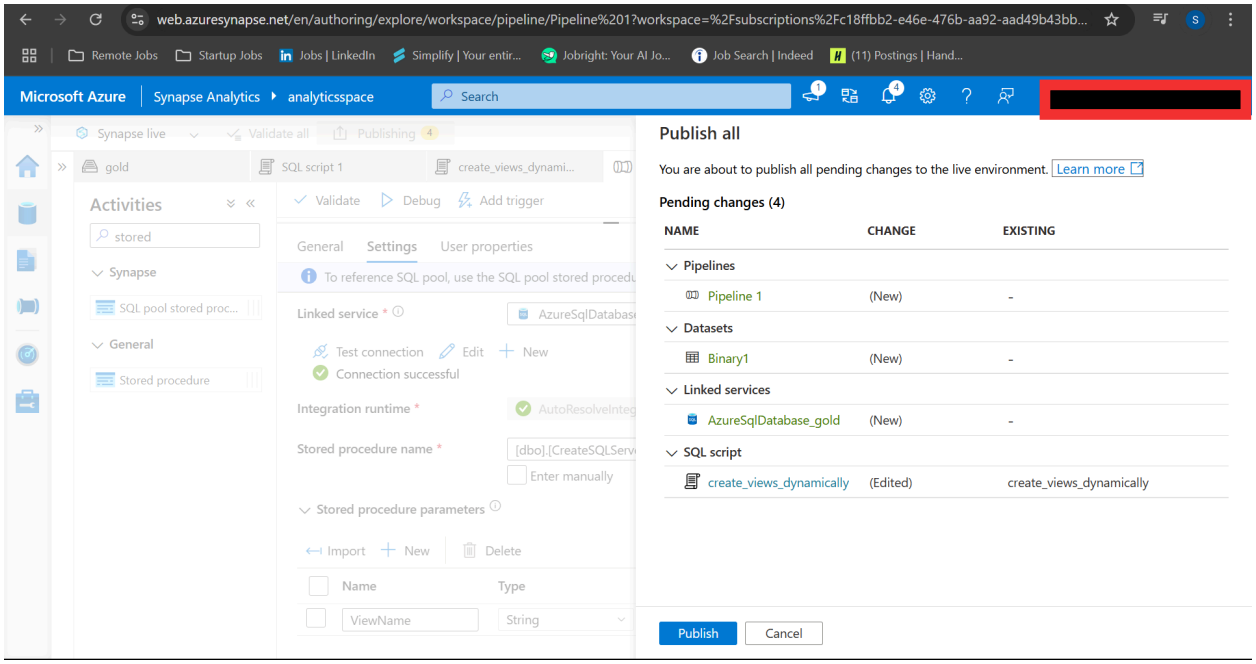


Choose “AzureSqlDatabase_gold” in the linked services available. In the stored procedure name, I selected the stored procedure which I had created earlier.

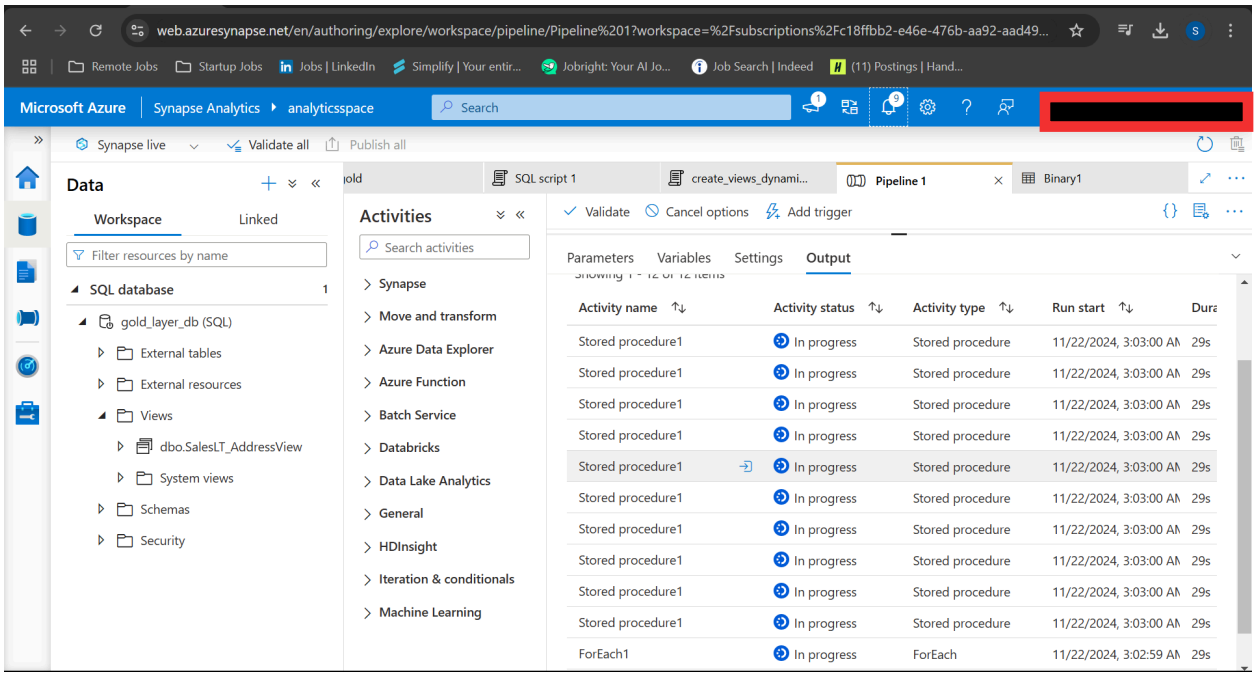


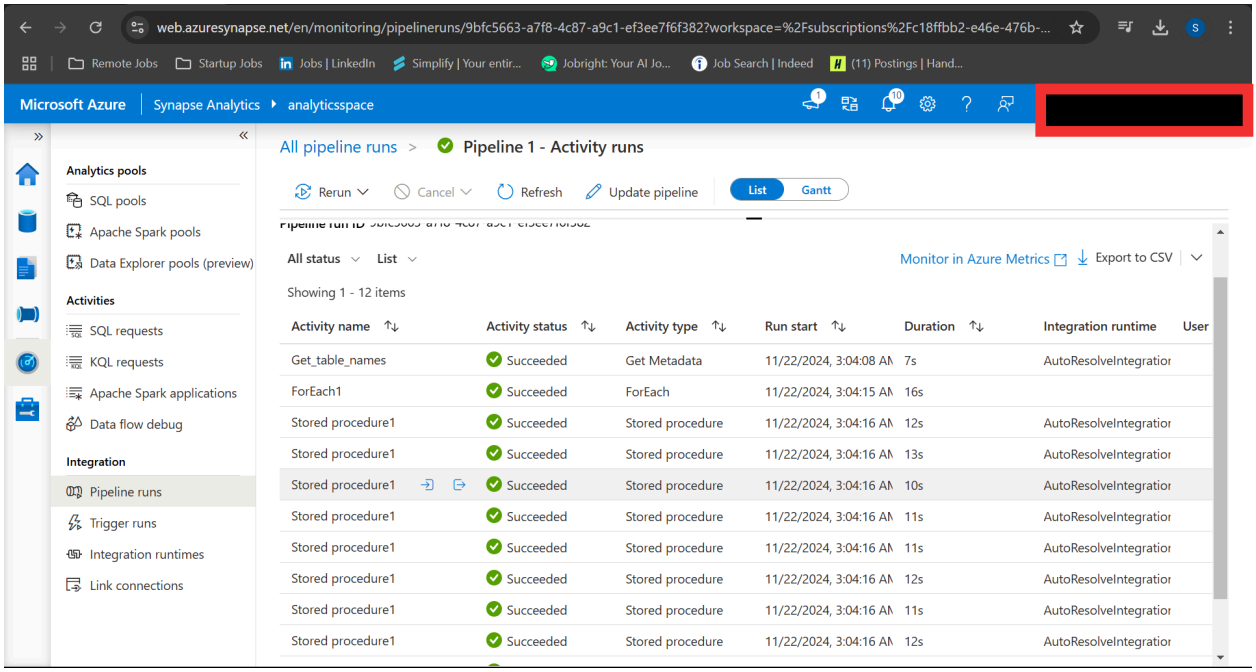
I created a new parameter for the stored procedure as shown below



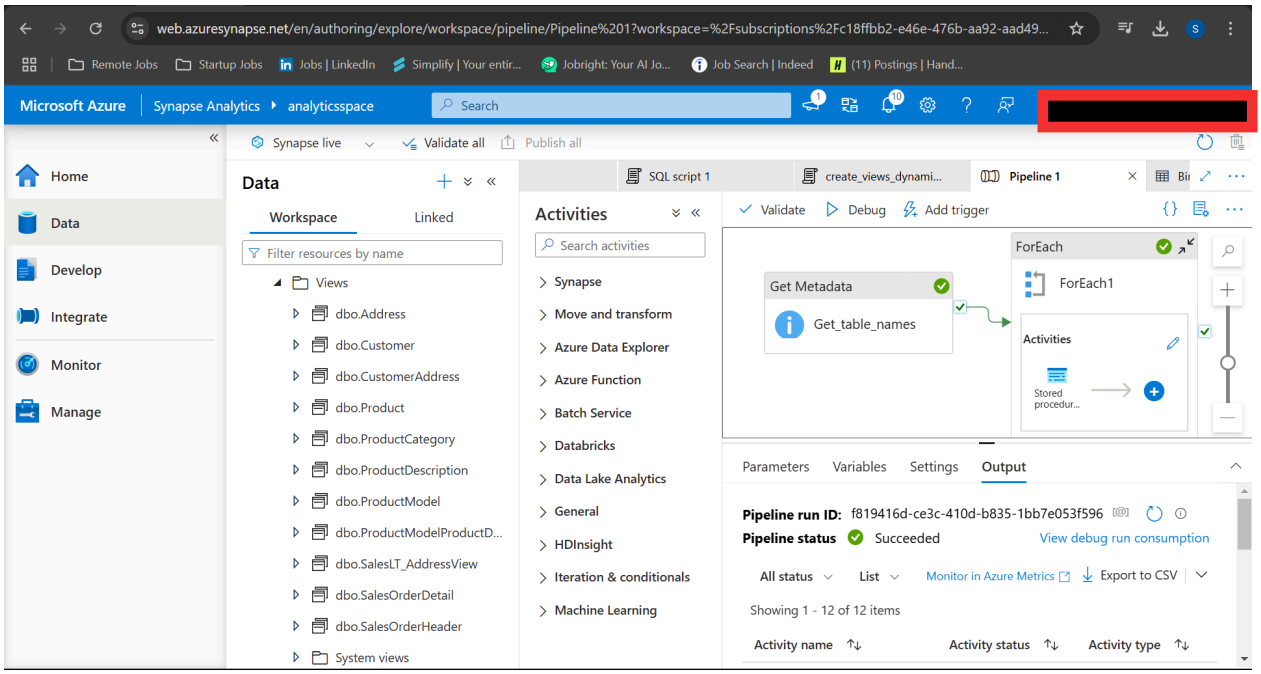


I triggered the pipeline using the “Trigger Now” option



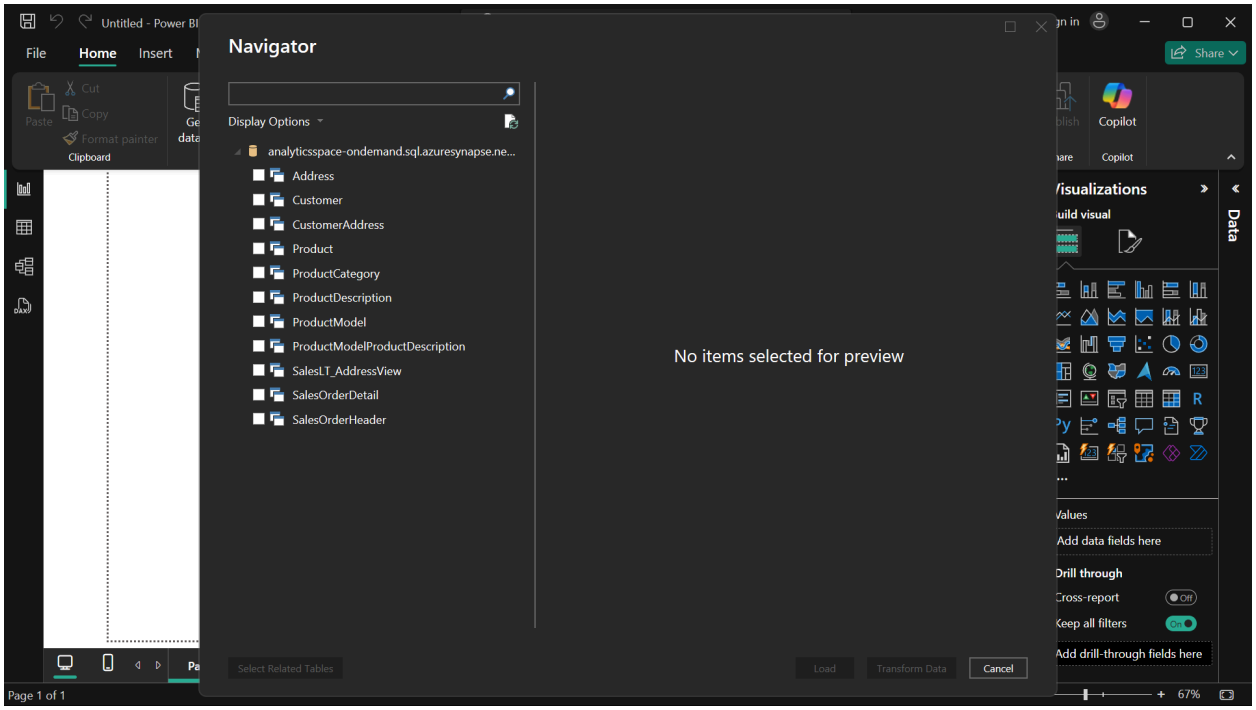


I can also see that the views are created dynamically as shown below

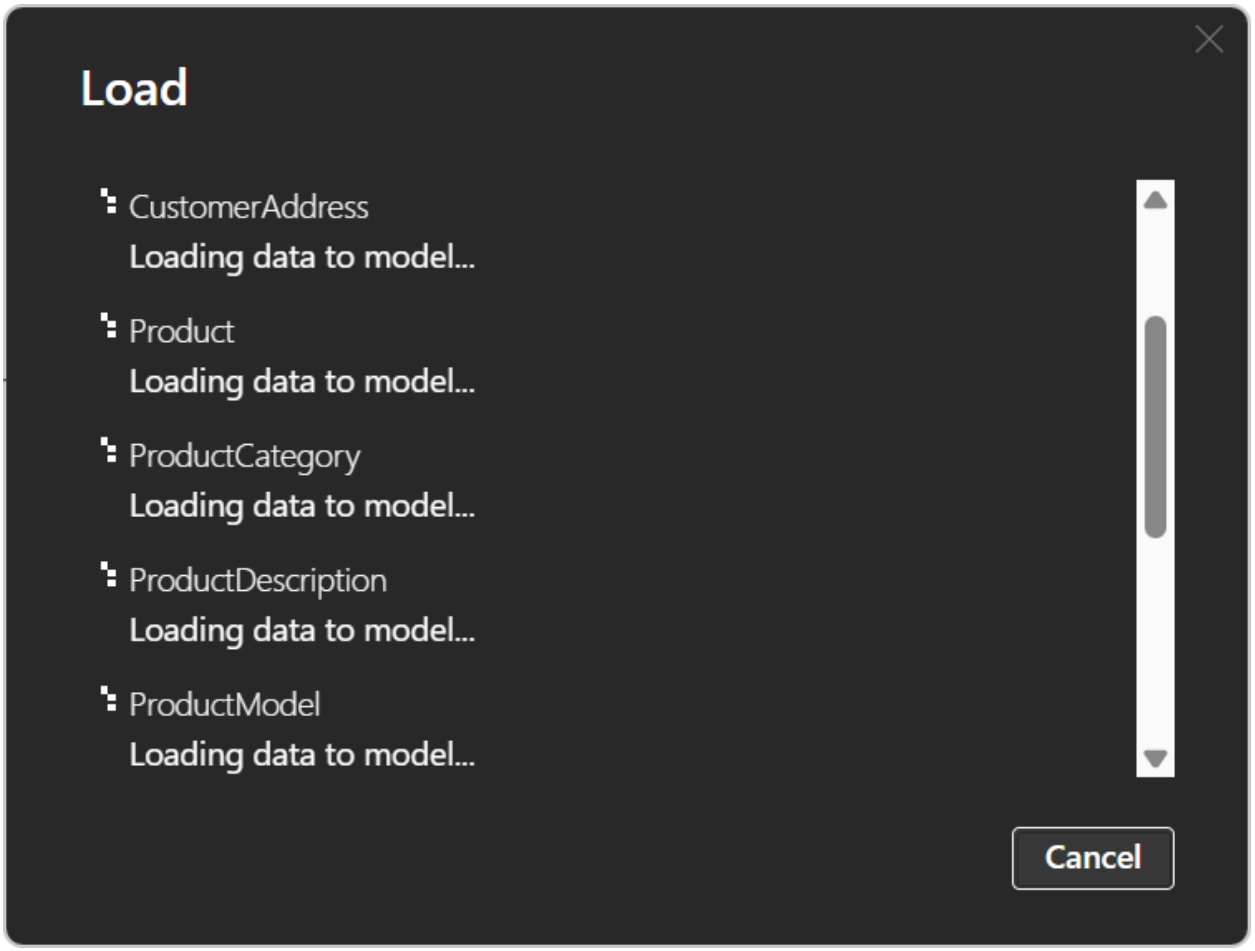


In the next step, I connected the azure synapse to the Power BI desktop. While connecting, I used the following credentials in the authentication

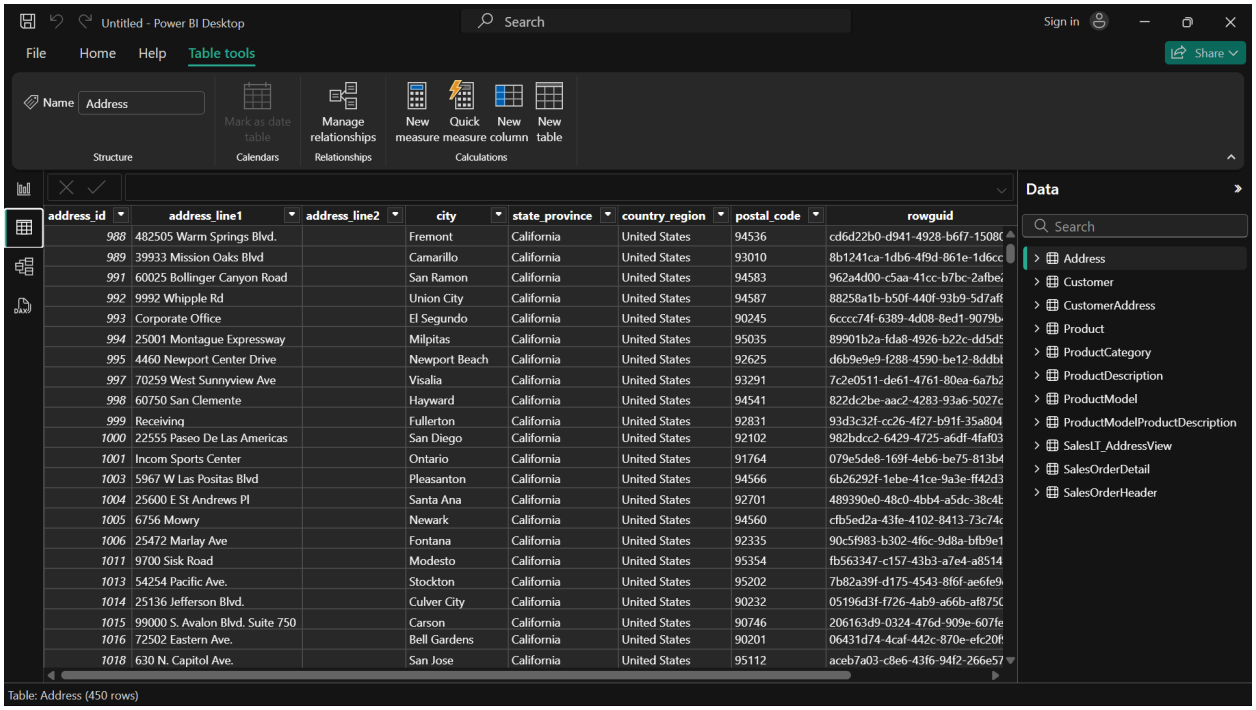
In the server name , I have added the details of SQL serverless endpoint. In the database details, I gave the name “gold_layer_db” (the database which I have created in synapse earlier).once the synapse is connected to power BI, I can see all the tables in power BI as below

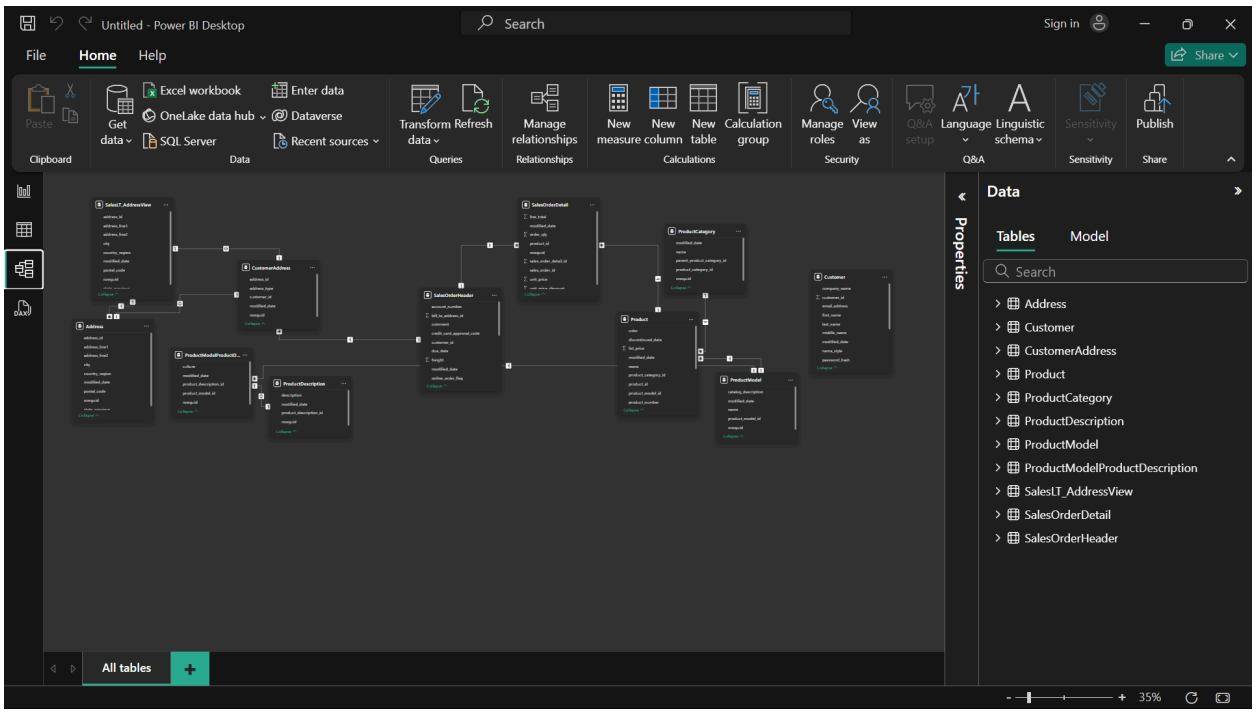


Also, I have chosen the “import” mode in Power BI, which means the data is essentially in the internal memory of Power BI. I proceeded to load all the tables

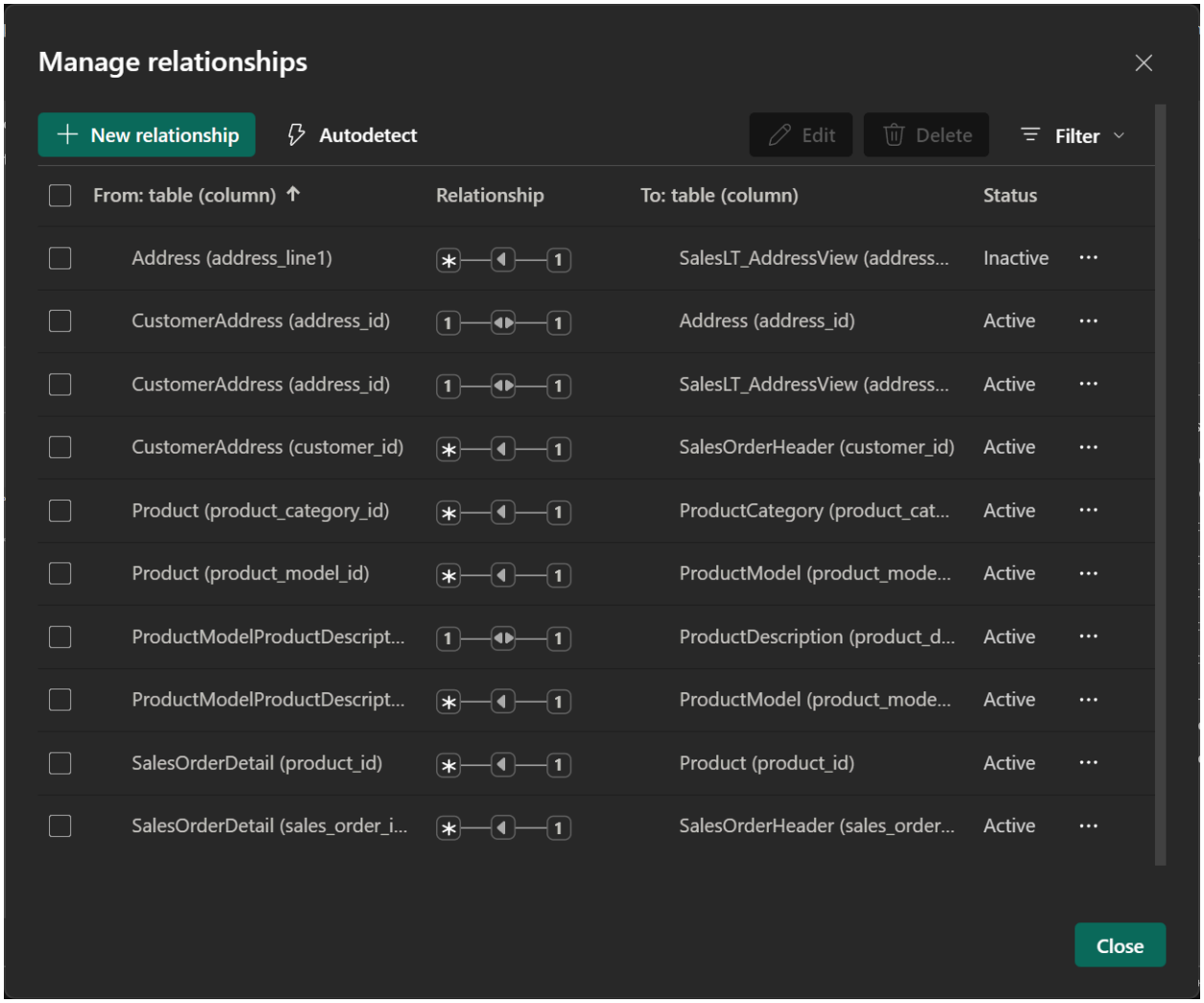


Once the data is loaded, I can see all the data in table view section of Power BI





I can find the relationships between the tables using “Manage” relationships table as below



Before proceeding to design any dashboard, I thoroughly read the business requirement and tried to narrow down the questions that the business is interested in.

From the requirement, following questions can be more relevant

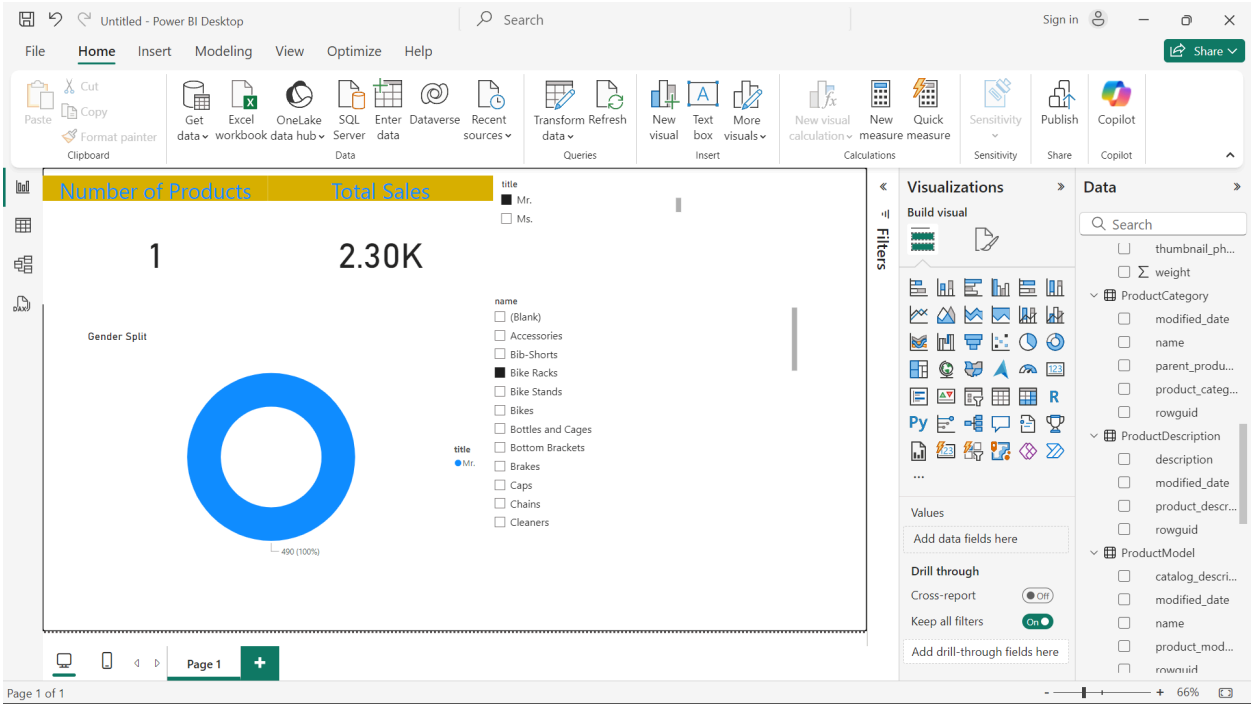
What are the distinct products that are available?

What are the total sales?

For a given gender, what are the sales for a particular product?

To answer the above, I created the following dashboard. I created the dashboard in such a way that the user can apply filters on both product category and gender to show the sum of sales.

Below is the snapshot of the dashboard



Resume bullet points

Designed and automated a scalable ETL pipeline using **Azure Data Factory** to extract customer and sales data from an **on-premises SQL Server**, loading it into **Azure Data Lake Storage** in the **Bronze layer** of the **Medallion Architecture**, ensuring secure integration through **Azure Key Vault**.

Transformed and modeled datasets using **Azure Databricks**, leveraging **PySpark** and **SQL** to process data from **Bronze to Silver and Gold layers**, enabling dynamic **data warehousing** and enriched analytics through structured **data modeling**.

Delivered actionable insights by developing an interactive **Power BI dashboard** that visualized KPIs such as **sales revenue, products sold, and gender distribution**, utilizing **Azure Synapse Analytics** for efficient querying and implementing **data governance** to ensure compliance and security

Future Scope:

Future Scope of the Project

1. **Real-Time Data Integration and Analysis:**
    - Enhance the pipeline by incorporating **real-time data streaming** capabilities using tools like **Azure Stream Analytics** or **Apache Kafka**. This will allow stakeholders to access up-to-the-minute sales metrics and demographic insights, providing greater agility in responding to emerging trends and customer behavior.
2. **Advanced Predictive Analytics and Personalization:**
    - Implement **machine learning models** using **Azure Machine Learning** and **Databricks** to forecast sales trends, identify potential product preferences by demographic segments, and recommend personalized product offerings. This will elevate the utility of the KPI dashboard from reporting to decision support.
3. **Scalability and Multi-Region Support:**
    - Extend the pipeline to handle data ingestion and reporting for multiple regions or stores, enabling a unified view of customer demographics and sales trends across geographies. Utilize **Azure Synapse Analytics**' dedicated pools for optimized performance on high-volume data processing.
4. **Enhanced Dashboard Features:**
    - Integrate **geographic analysis** and drill-down capabilities into the **Power BI dashboard** to offer richer visualizations, such as region-specific gender distribution and product sales insights. Add predictive visualizations, like projected sales by category and demographic trends over time.
5. **Broader Data Source Integration:**
    - Expand data sources to include **social media analytics**, **CRM systems**, and **web traffic logs** to enrich customer demographic data, providing a more comprehensive understanding of customer preferences and engagement.
6. **Data Governance and Compliance:**
    - Implement advanced data governance tools such as **Azure Purview** to manage data lineage, ensure compliance with regulations like GDPR, and maintain data quality across the pipeline. Introduce **fine-grained access controls** to secure sensitive customer data effectively.
7. **Incremental and Historical Data Processing:**
    - Transition the pipeline from a daily overwrite model to an **incremental load approach** using **Delta Lake**'s time-travel features. This will enable efficient updates and maintain a history of changes for longitudinal analysis.
8. **Automation and CI/CD Pipelines:**
    - Introduce **CI/CD pipelines** for the data engineering workflows using tools like **Azure DevOps** or **GitHub Actions**, ensuring seamless deployment, testing, and version control for all components of the data pipeline.
9. **Self-Service Analytics for Stakeholders:**
    - Develop self-service capabilities within **Power BI** or as APIs, allowing business users to perform ad hoc queries, customize reports, and explore data insights independently.
10. **Integration with Business Processes:**

- Leverage insights from the pipeline and dashboard to trigger automated workflows using **Azure Logic Apps** or **Power Automate**, such as sending inventory alerts or initiating targeted marketing campaigns based on sales trends and gender-based preferences.