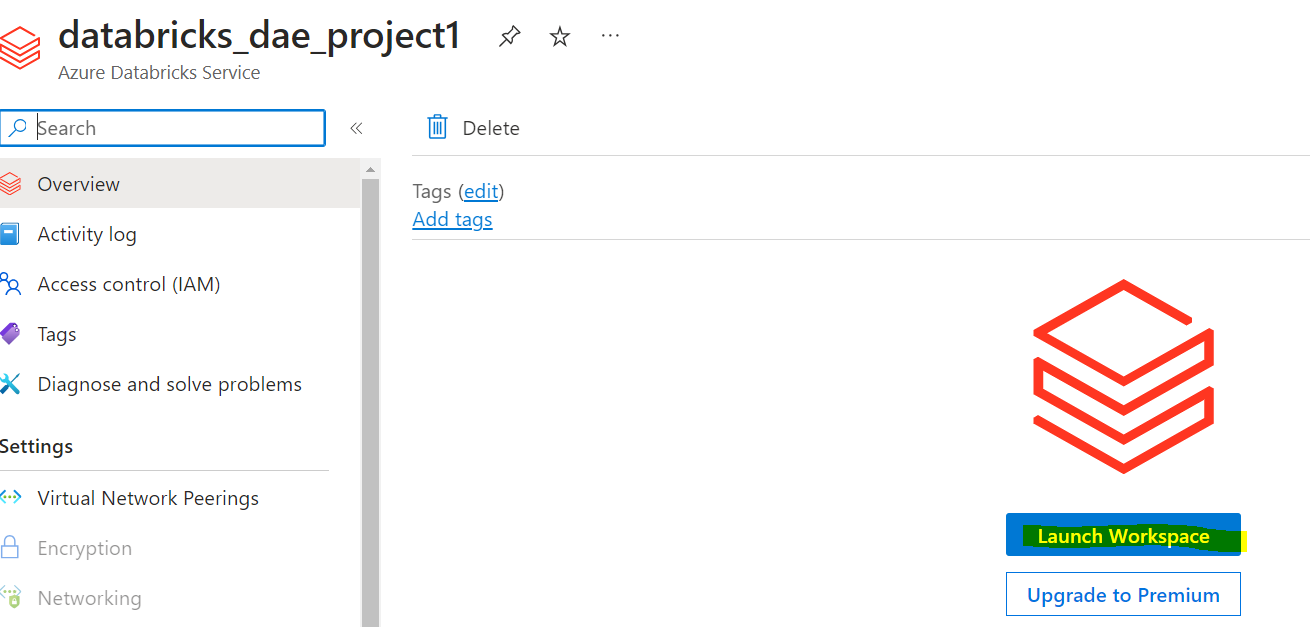
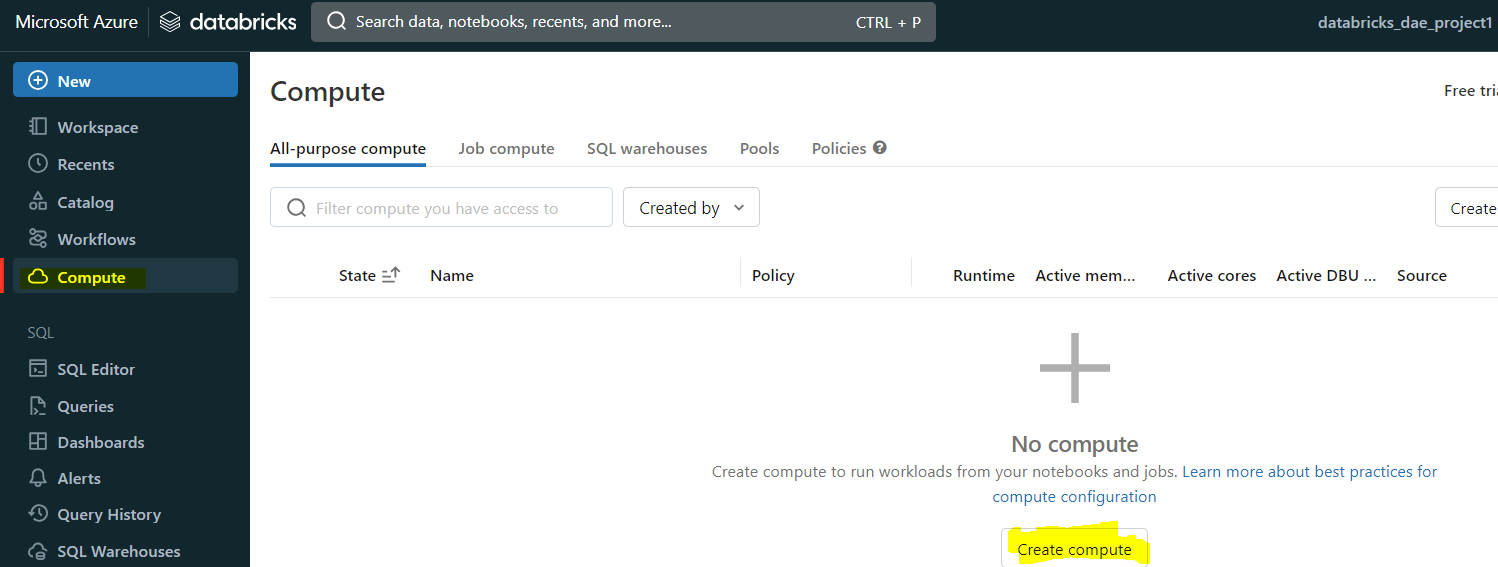
**Part 3: Data Transformation**

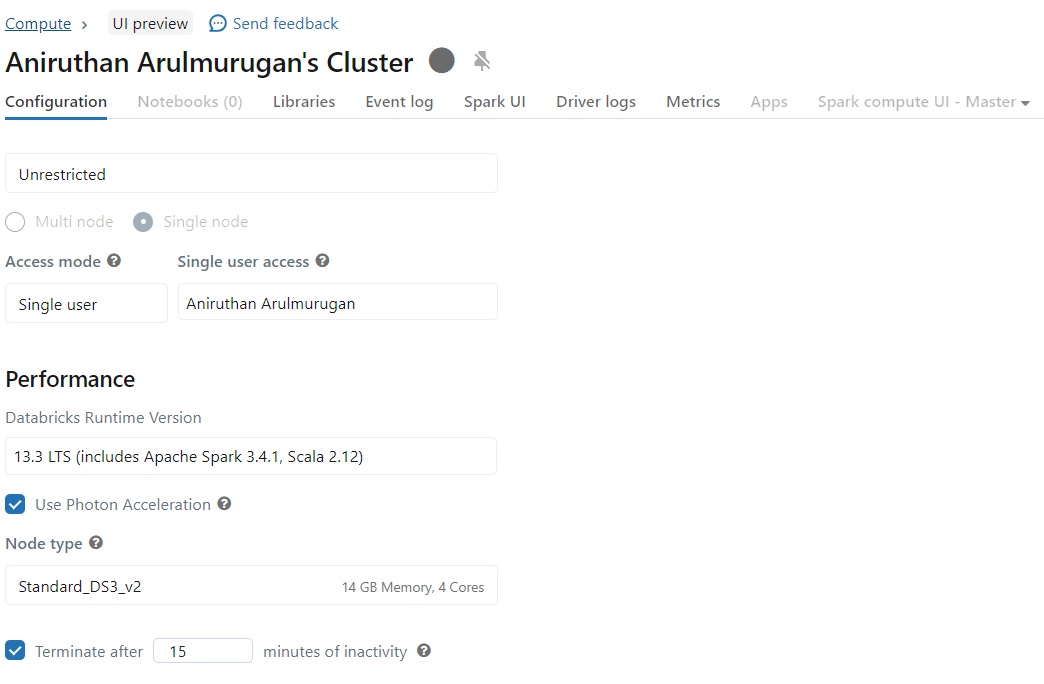
* Create Azure data bricks resource from the marketplace in the resource group.
* Go to the created resource and click on launch workspace



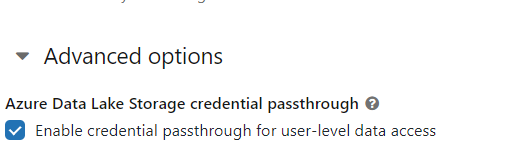
* Click on the compute tab on the left side and create compute



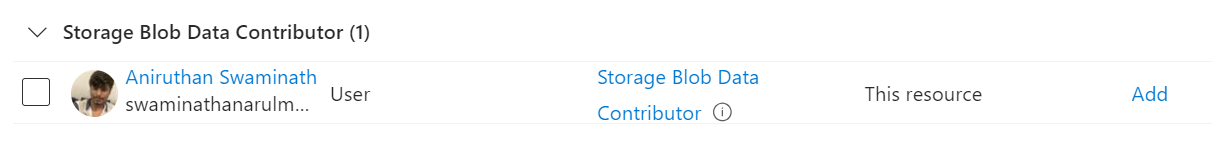
* You can use the default settings in the compute settings since we are only handling 10 tables in this project



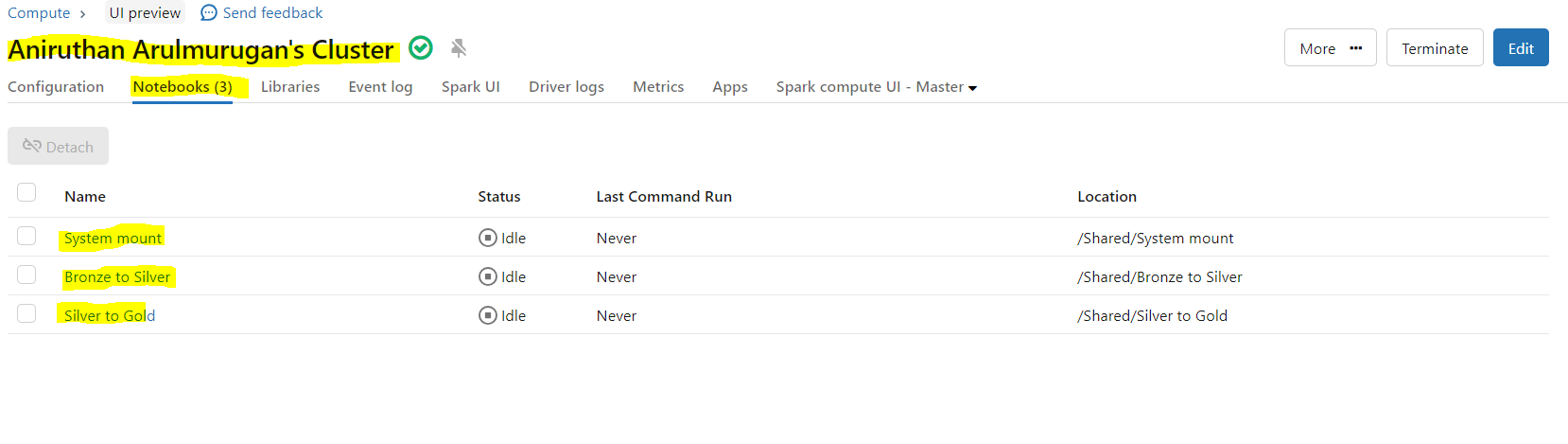
* Set the terminate after to 15 mins
* In the advanced settings ensure enable credential passthrough for user-level data access is enabled



* This will allow the Databricks user to access the data through his credentials since my user account has Storage blob data contributor access in Datalake. There are other methods to ensure Databricks can access Datalake as well.



* Click on Create Compute to create the compute cluster. This can take up to 5 mins.
* Once the compute cluster is created, go to the workspace tab-> shared folder and create 3 notebooks- System mount, Bronze to Silver, and Silver to Gold.
* The created notebooks will be available in the compute cluster under the notebooks tab



* To mount the Access Azure Data Lake Storage directly using credential passthrough into DBFS, go to the [link](https://learn.microsoft.com/en-us/azure/databricks/archive/credential-passthrough/adls-passthrough)
* In the website go to the Mount Azure Data Lake Storage to DBFS using credential passthrough section and copy the python code

Find the code below:

configs = {

"fs.azure.account.auth.type": "CustomAccessToken",

"fs.azure.account.custom.token.provider.class": spark.conf.get("spark.databricks.passthrough.adls.gen2.tokenProviderClassName")

}

# Optionally, you can add <directory-name> to the source URI of your mount point.

dbutils.fs.mount(

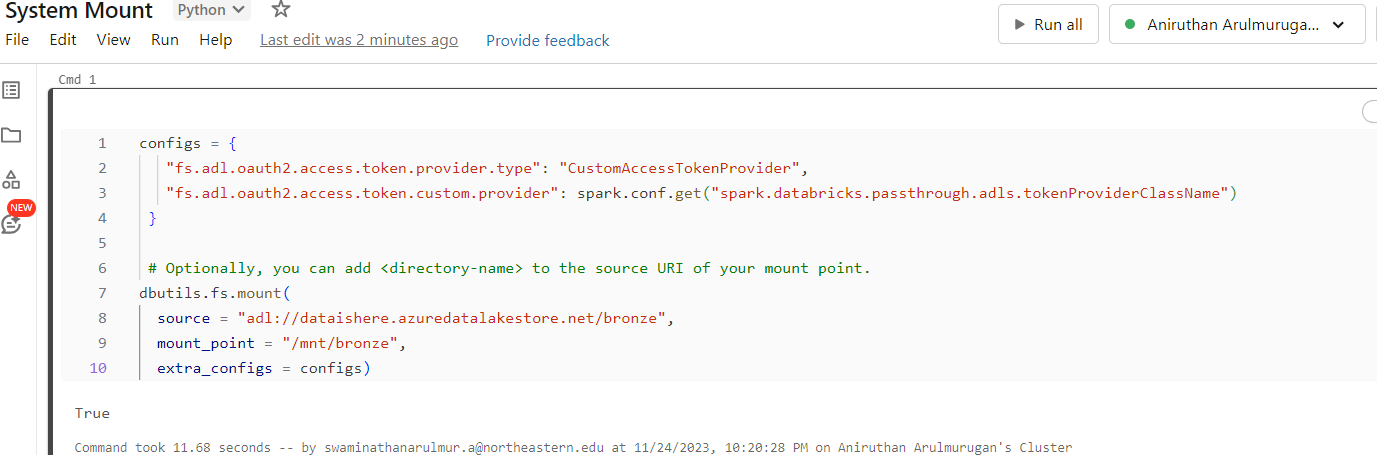
source = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/",

mount\_point = "/mnt/<mount-name>",

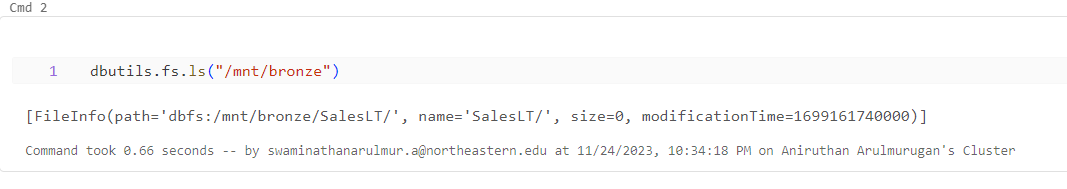
extra\_configs = configs)

* We need to make few changes in the above code,

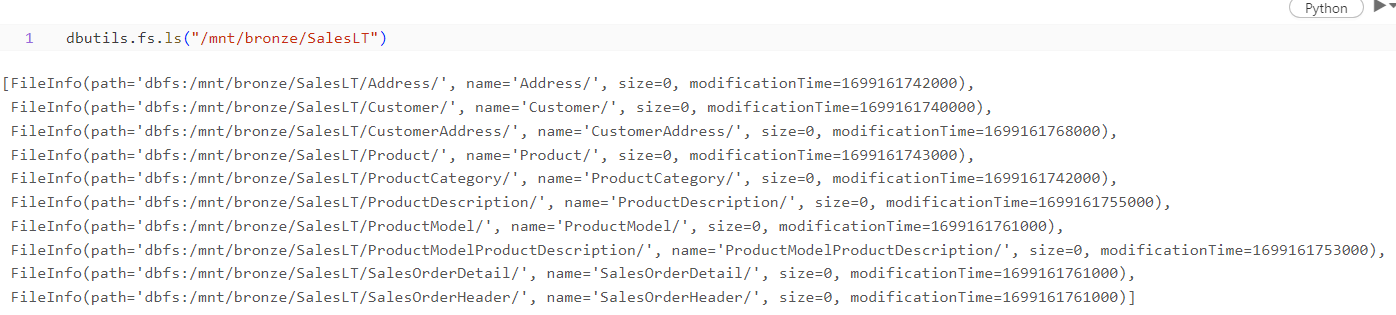
1. Replace <storage-account-name> with the name of your datalake. Here we will use Dataishere.
2. Directory name is the folder name from which you want to get the data from. Here we will replace it with bronze.
3. Mount point name is the name of the mount point through which the data in thr cloud storage can be accessed as a folder.



* Copy and paste the command in System mount notebook and run it. You will get True as output on successful execution.
* We can now use the mountpoint to view the data inside the bronze folder of Datalake
* To view all the data in the container use the below code

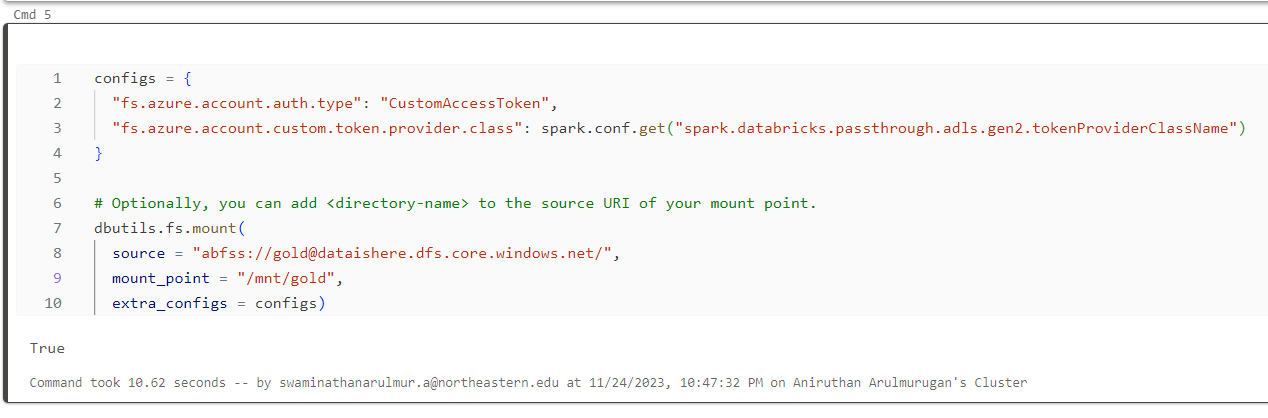


* The command shows the folder inside the mount point which is SalesLT
* If you run the below command, you will get the tables inside SalesLT



* The name=’Address/’ will be used in the bronze to silver transformation to get the table names
* Similarly we will mount the other two containers in the datalake- silver and gold





* Now we are going to transform the data in bronze mount point and load it into silver mount point
* We will use the bronze-to-silver notebook for the first set of transformation. The transformations in this project are going to be basic to understand how it can be done using databricks
* First in the tables we have a few date columns but the values of these columns are in DateTime format. If the column name has date or Date in it we are going to change the column data values to date format.

**Note: we can run a file with one or more programming languages in Databricks and this is called magic commands- In ipynb file empty cell %sql command will allow the cell to run SQL commands.**

* We first get names of all the tables inside salesLT, we run the below code in bronze to silver notebook for this



* It initializes an empty list named `table\_name` and then iterates over the contents of a directory in Databricks File System (DBFS) located at `/mnt/bronze/SalesLT`. For each item in this directory, it extracts the name, splits it at the first slash (`/`), takes the first part of this split (as mentioned earlier), and appends it to the `table\_name` list. This process effectively collects the names of all files or directories directly under the `/mnt/bronze/SalesLT` path in DBFS.
* Now to perform the transformation we use the following code



* **This Python code snippet is designed to be run in a Spark environment, and it performs a series of data transformation steps on Parquet files that are located in a Databricks File System (DBFS).**
* **Here's a breakdown of its functionality:**

1. The necessary functions and types are imported from the PySpark SQL library, which will be used for manipulating date and timestamp data within the Spark DataFrame.

2. The `for` loop iterates over a list called `table\_name`, which presumably contains the names of tables or datasets.

3. For each item in `table\_name`, the code constructs a file path to a Parquet file located in the `/mnt/bronze/SalesLT/` directory. The naming convention for the Parquet files appears to follow the pattern of `[table\_name]/[table\_name].parquet`.

4. It then reads the Parquet file into a Spark DataFrame `df`. Spark's DataFrame API allows for direct reading of Parquet files, which are a columnar storage format.

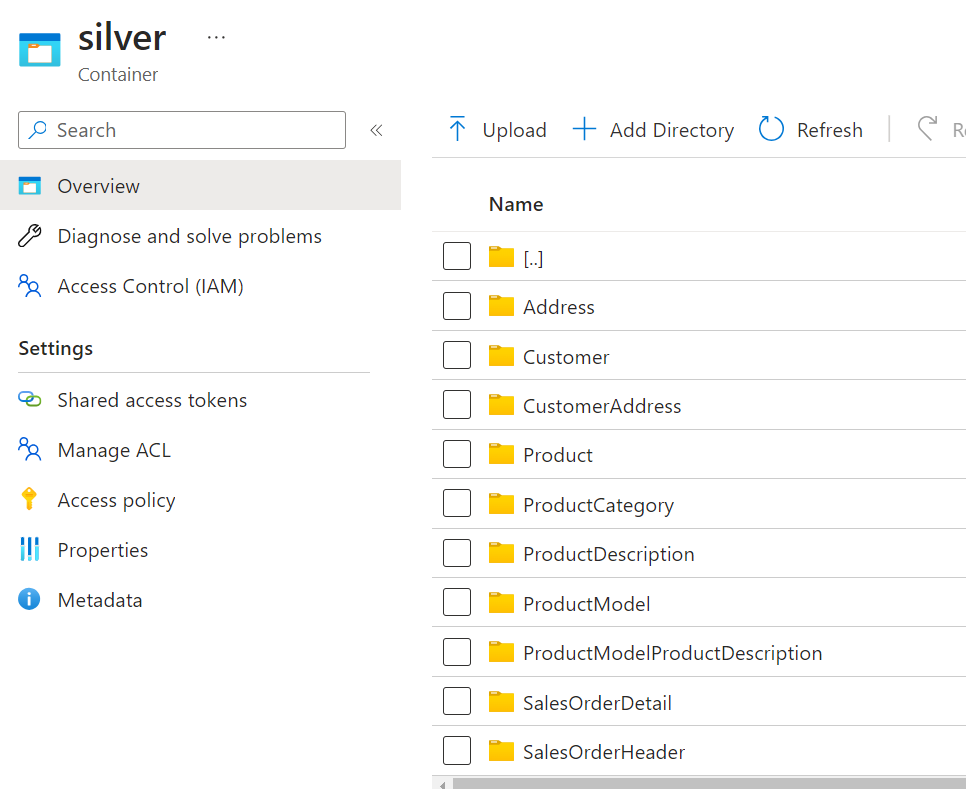
5. The code retrieves the columns of the DataFrame and iterates over them to identify any column names that include "Date" or "date".

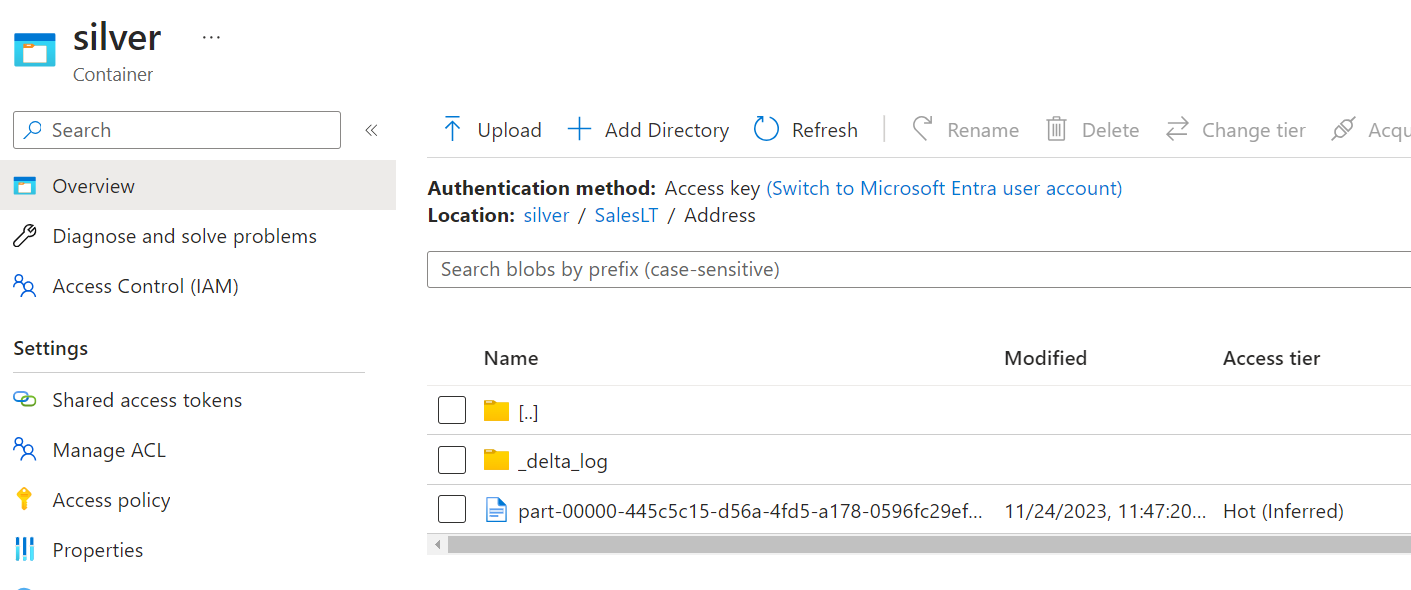
6. For each column that matches the condition (i.e., it is a date column), the code converts the timestamp from UTC to a string representation in the format "yyyy-MM-dd". This conversion is achieved by casting the column to a `TimestampType` and then applying the `from\_utc\_timestamp` function followed by `date\_format`.

7. After all the date columns have been processed, the code defines an output path under the `/mnt/silver/SalesLT/` directory, using the same table name as a directory.

8. Lastly, the transformed DataFrame is written out to the defined output path in the Delta format with the write mode set to "overwrite". This means if the output location already contains data, it will be replaced with the new, transformed data.

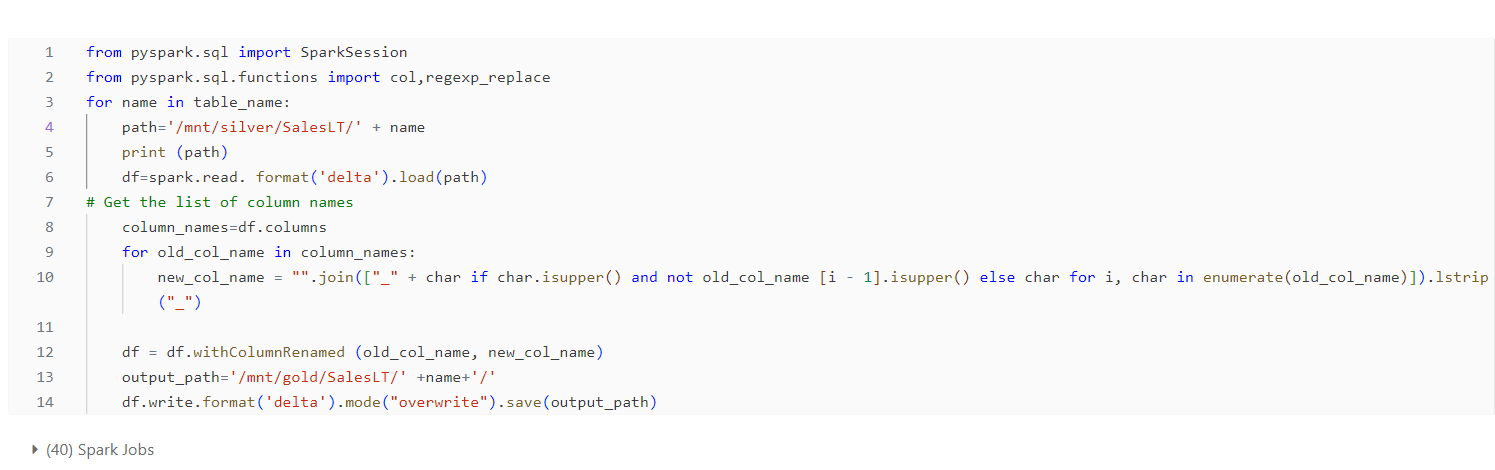
* The delta format is built over the parquet format. It has all the features of parquet format
* **The Delta format is chosen because it is an advanced storage layer that brings ACID transactions to Apache Spark and big data workloads. It allows for features like schema enforcement and time travel, which makes managing and understanding data evolution much easier. The use of Delta format implies that the data is likely being prepared for a more robust and reliable data pipeline, enabling advanced analytics and machine learning capabilities on top of the existing data infrastructure.**
* Now you can the transformed data stored in the silver container of the Datalake in delta format. You can also see the delta log files which are used for version tracking.





* Next we are going to perform Transformation 2. The data in Silver container is going to be further transformed.
* In this project this transformation is going to be simple as well.
* If there is a column name with ProductNumber, we are going to change it to Product\_Number.
* The below code can be used





* Here's a step-by-step explanation of the code:

1. It initializes an empty list called `table\_name`.

2. The `for` loop iterates over the directories found at the path `/mnt/silver/SalesLT` in the Databricks File System (DBFS) using `dbutils.fs.ls`. For each directory, it extracts the directory's name, strips the trailing slash (assuming there is one), and appends it to the `table\_name` list. This is collecting the names of datasets stored at that location.

3. It then imports `SparkSession` from `pyspark.sql` and `col`, `regexp\_replace` from `pyspark.sql.functions`. However, `SparkSession` is not used in the snippet, and `regexp\_replace` is imported but not utilized in the code provided.

4. Another `for` loop iterates over each name in the `table\_name` list.

5. It constructs the path to each dataset based on the name and prints this path.

6. The DataFrame `df` is loaded from the constructed path using Spark's DataFrame API, which reads data in Delta format.

7. It gets the list of column names from `df`.

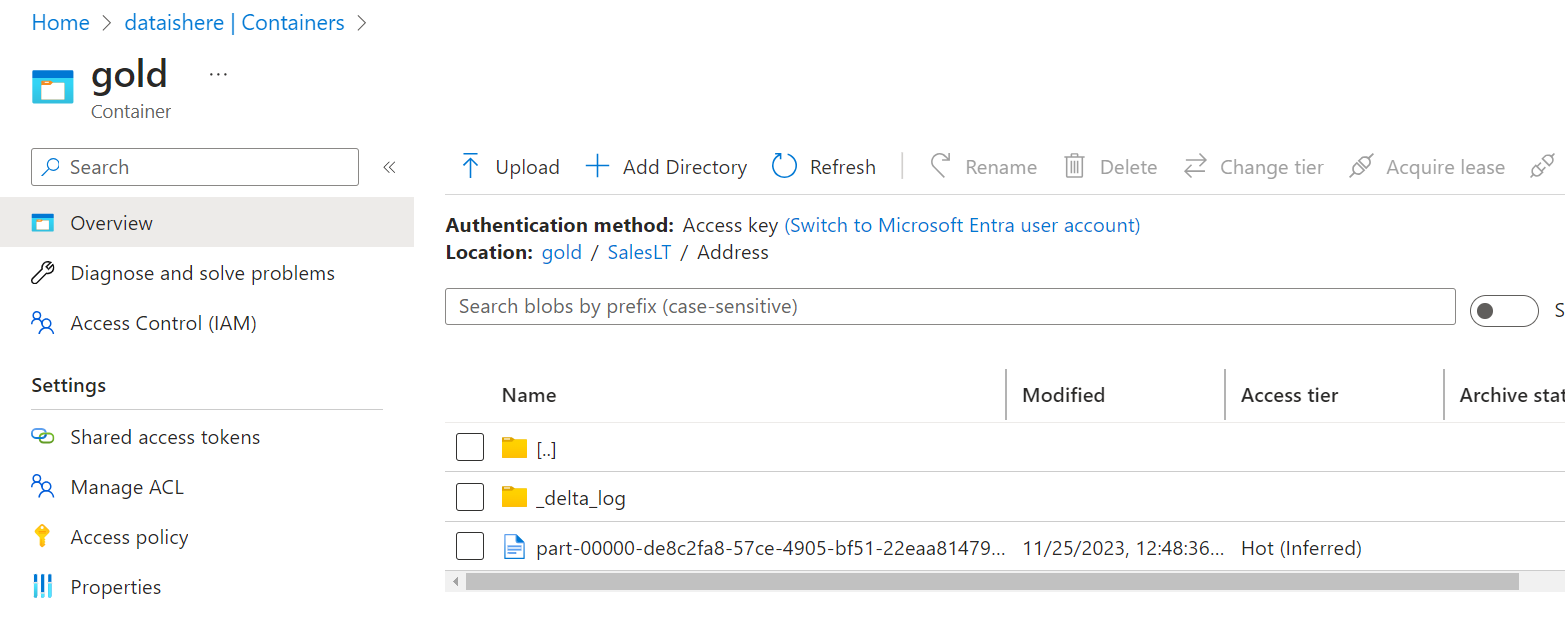
8. The next `for` loop iterates over each column name. It creates a new column name by inserting an underscore `\_` before any uppercase character that follows a lowercase character, effectively converting camelCase or PascalCase to snake\_case. The `.lstrip("\_")` is used to remove any leading underscore that might be added if the original column name started with an uppercase letter.

9. It then renames the columns from the original name (`old\_col\_name`) to the new snake\_case name (`new\_col\_name`).

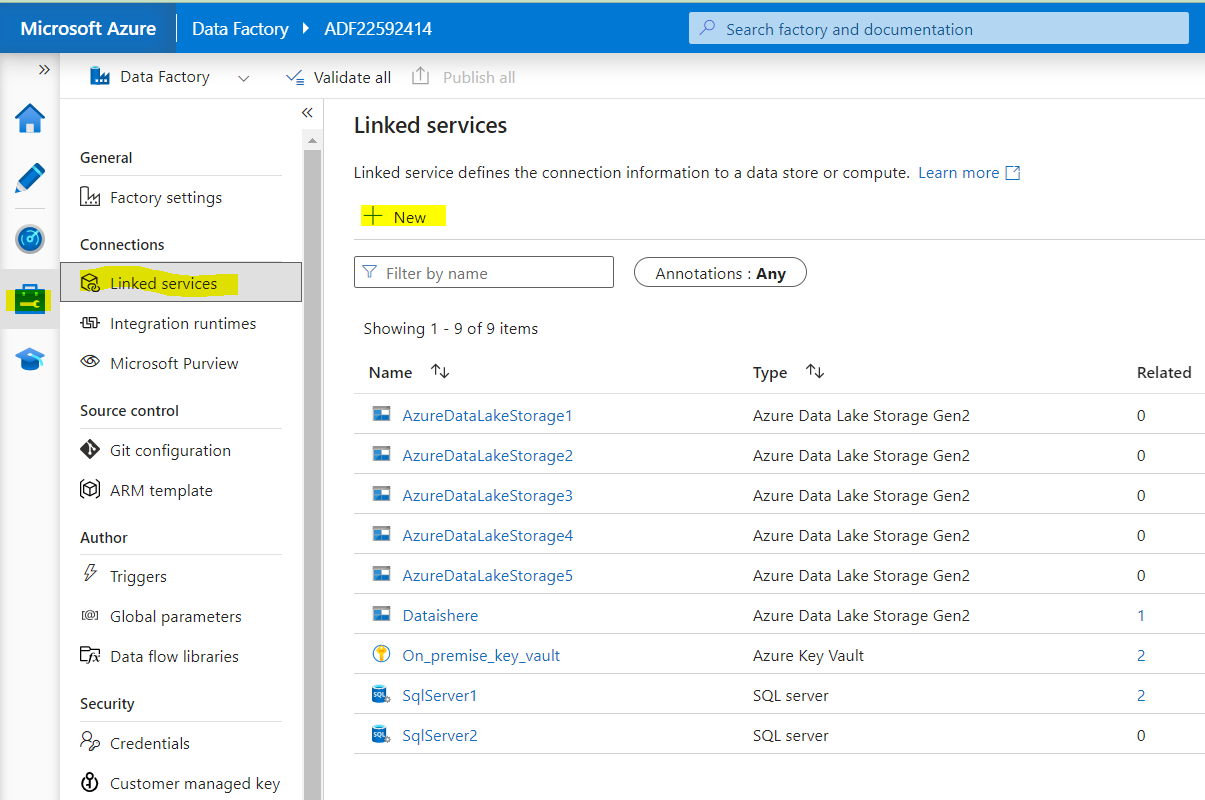
10. After renaming the columns for the DataFrame, it defines an output path using the name of the dataset. This path points to a different directory in DBFS (`/mnt/gold/SalesLT/`).

11. Finally, the DataFrame is written to the specified output path in the Delta format with the write mode set to "overwrite". This will replace any existing data at that location with the newly transformed DataFrame.

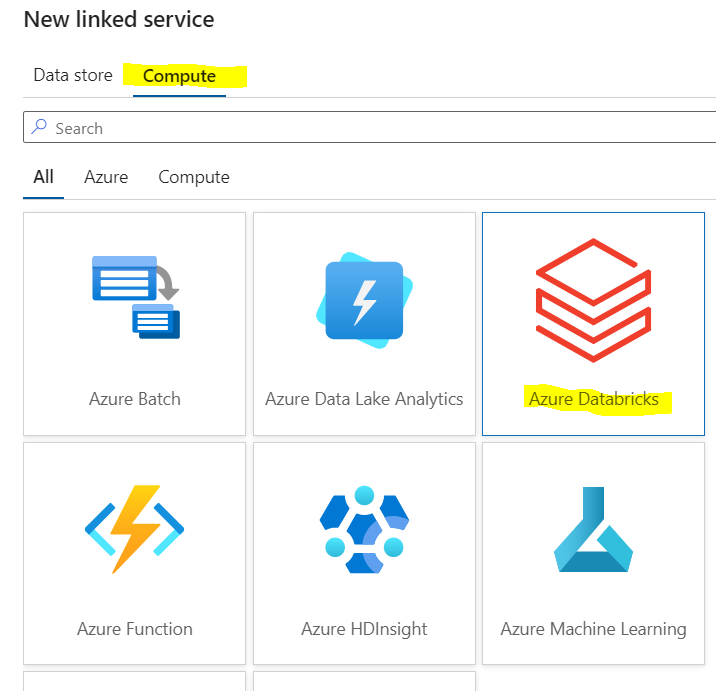
* The code is part of a data pipeline that promotes datasets from the 'silver' layer to the 'gold' layer in a data lake hierarchy, which typically signifies a progression from raw/processed data to refined/consumable data. The Delta format is used for its advanced capabilities like ACID transactions and time travel, which are beneficial for data reliability and consistency.
* The gold container now consists of the further transformed data in the datalake



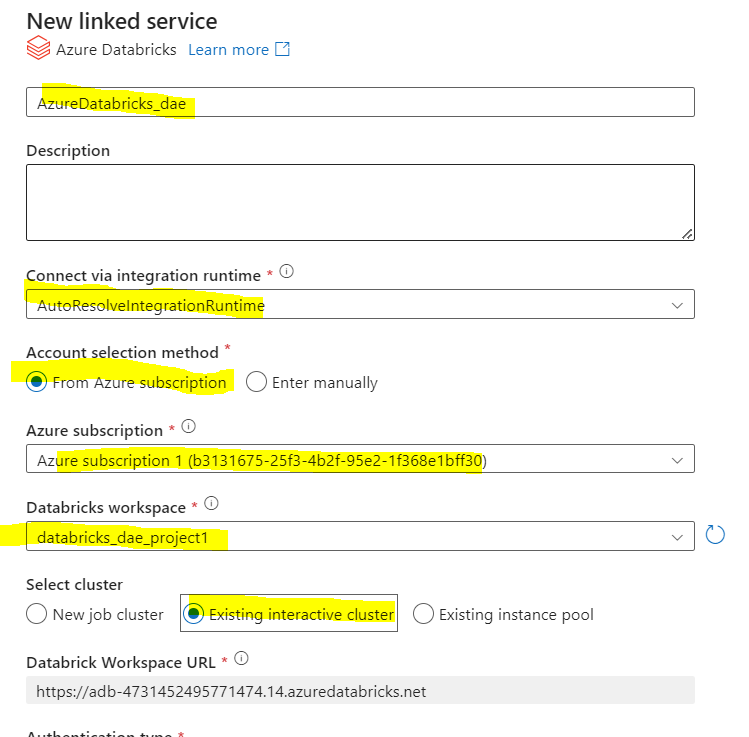
* Now all the transformation is done and the notebooks are ready to be added to the Azure data factory pipeline. The notebooks bronze to silver and silver to gold need to be added to the pipeline created in Azure Data factory.
* To do this first go to the ADF and create a new linked service connection to Databricks
* In the ADF manage tab, select linked services and click on new



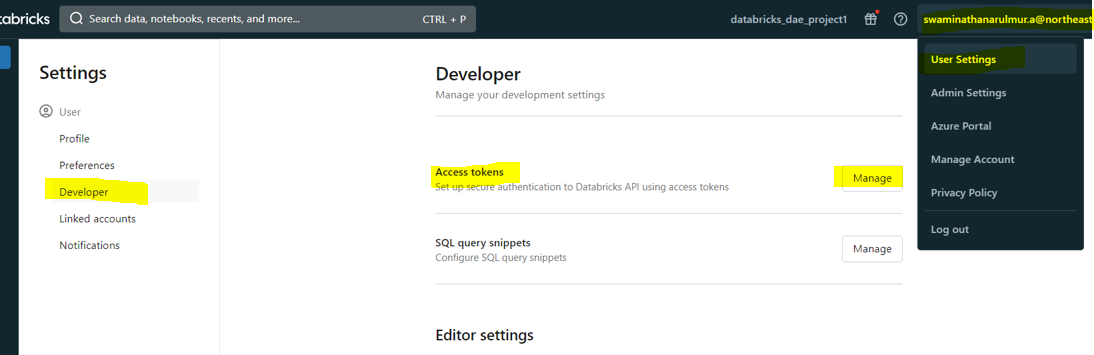
* In the compute section select Azure data bricks



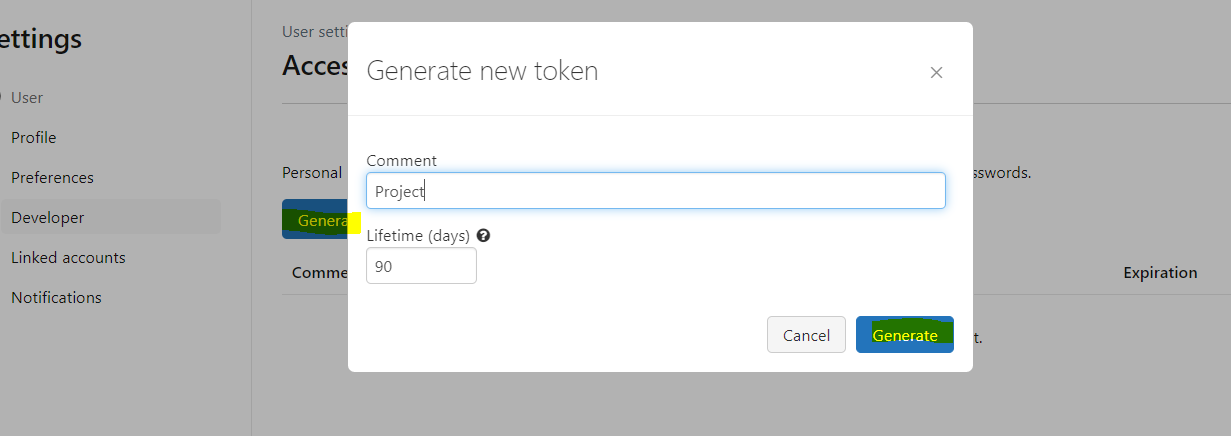
* Mention the below details for the initial setup of the linked service



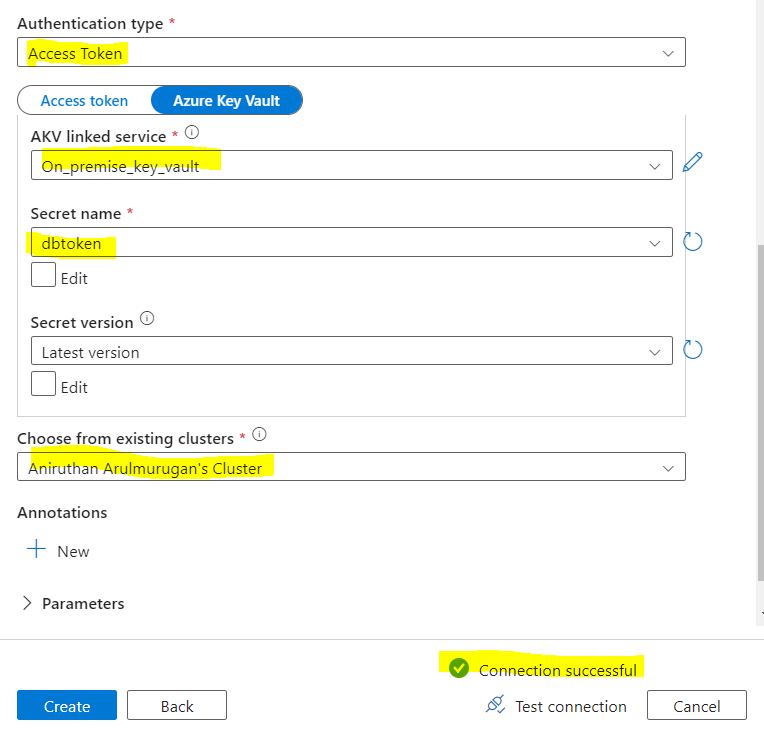
* For the authentication type, we have to create a new access token from databricks and store it in keyvault and use that secret to authenticate through Azure keyvault linked servive
* To generate a token in databricks, go to user settings->Developer->Access Tokens-> Manage



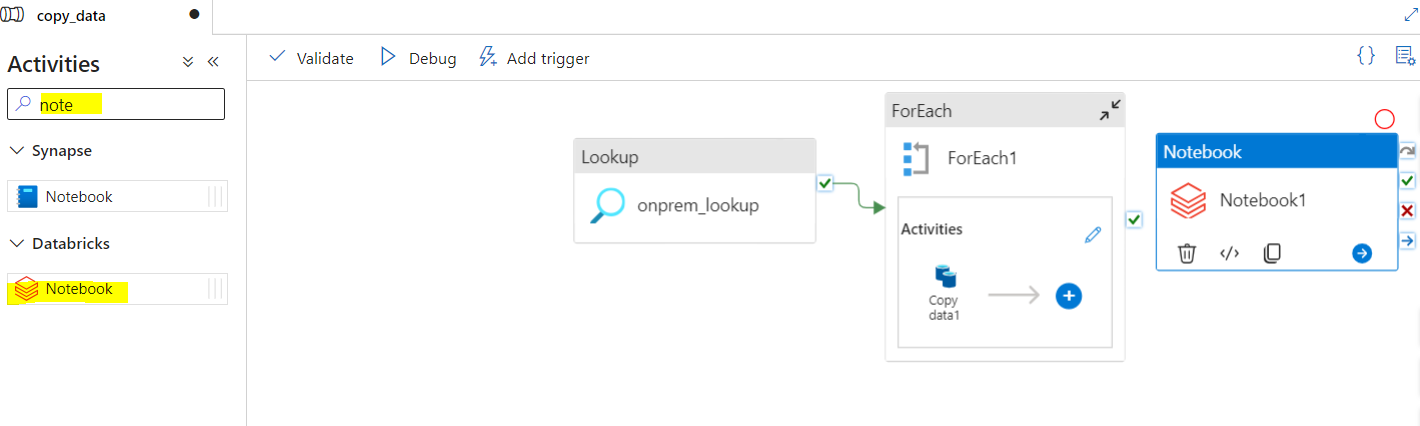
* Click on generate token-> enter the comment and click on generate



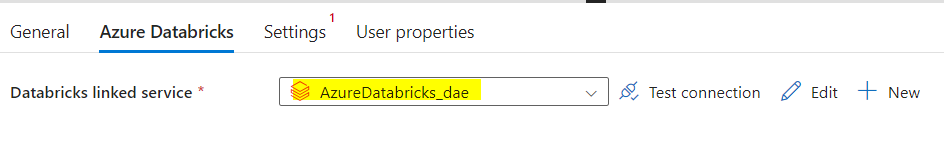
* Make sure to copy the token as you won’t be able to view it again- dapi69411a275677b0dd2fd9c0eec572faa7-3
* Now create a secret in vault for the access token and use it for authentication using the vault linked service



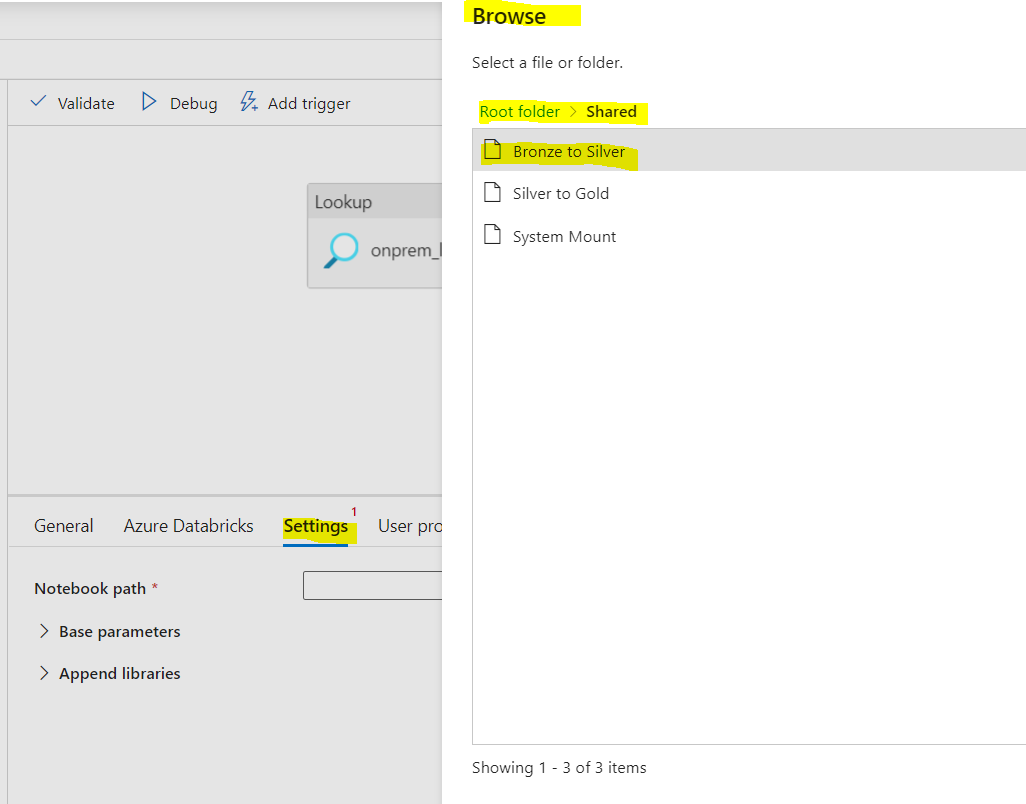
* Once the linked service is established we can add the notebooks to the ADF pipeline
* Go to the author tab in ADF to open the created pipeline and create two notebook activities for bronze to silver and then silver to Gold.
* Select Azure notebook activity



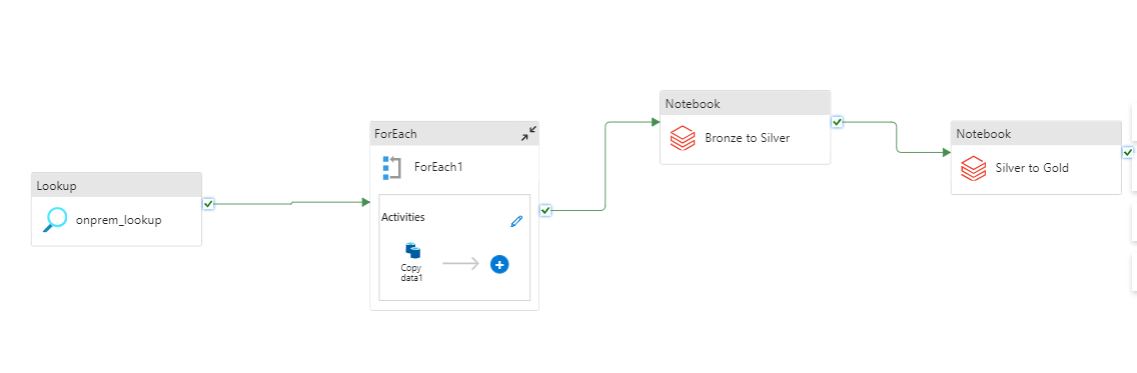
* Connect the for each on success to first notebook
* In the Azure databricks tab for Notebook1, add the created linked service

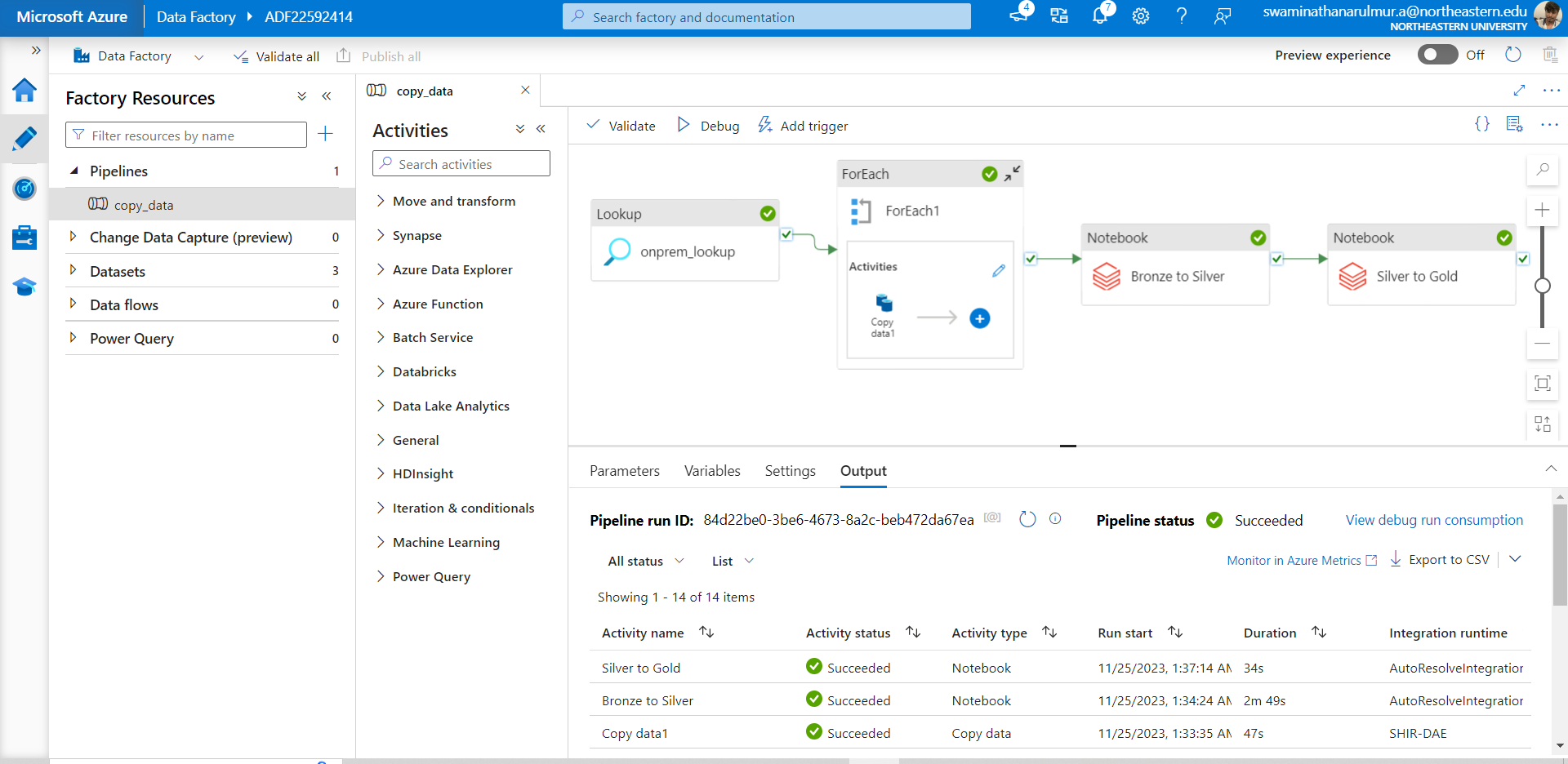


* In the notebook1 settings tab, click on browse->root-> shared->Bronze to Silver



* Similarly create an another notebook, and use the same linked service but in the settings choose the Silver to Gold in root-> shared folder.
* Connect notebook1 on success to notebook2
* Rename the notebook 1 to bronze to silver and notebook 2 to silver to gold.
* Publish all the changes and trigger now the pipeline

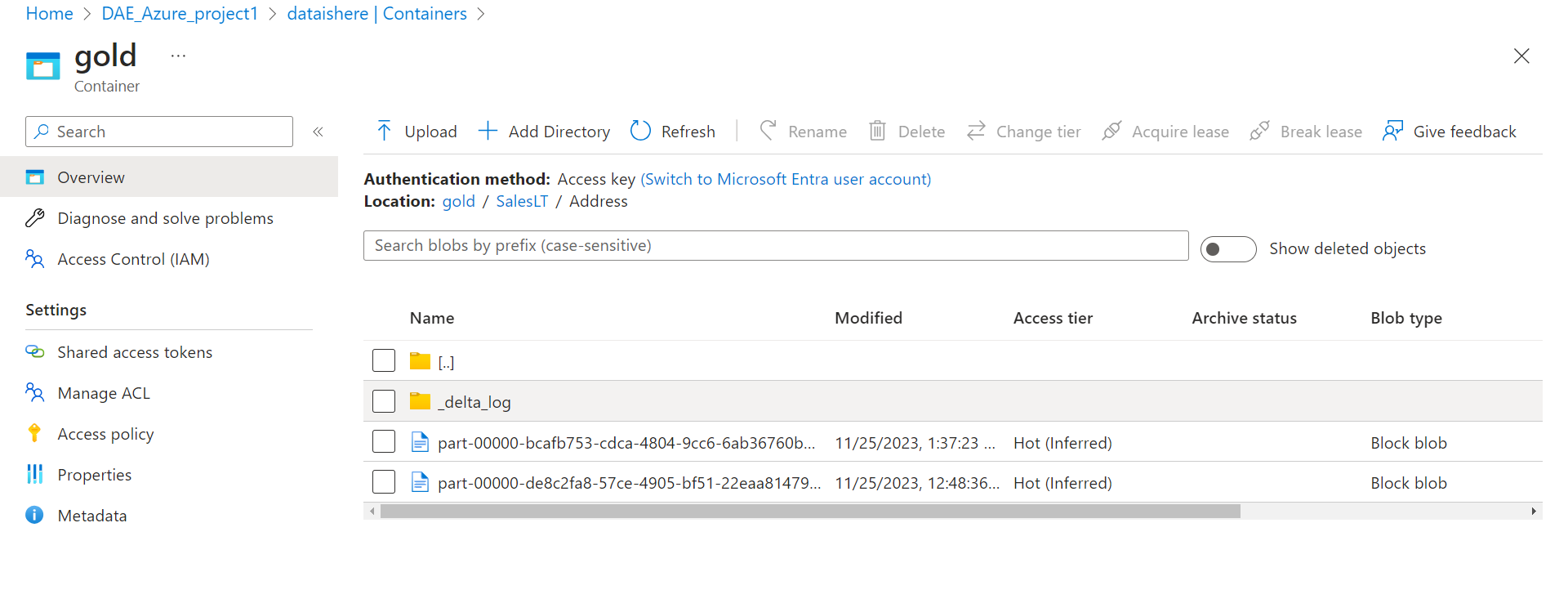




* The pipeline was executed successfully.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Activity name | Activity status | Error | Activity type | Run start |
| Silver to Gold | Succeeded | DatabricksNotebook | 11/25/2023, 1:37:14 AM |
| Bronze to Silver | Succeeded | DatabricksNotebook | 11/25/2023, 1:34:24 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:35 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| Copy data1 | Succeeded | Copy | 11/25/2023, 1:33:34 AM |
| ForEach1 | Succeeded | ForEach | 11/25/2023, 1:33:33 AM |
| onprem\_lookup | Succeeded | Lookup | 11/25/2023, 1:33:20 AM |

* The same is processed twice one in the data bricks without the pipeline and once through the pipeline. You will see two identical delta files inside all the folders of silver and gold folders. However, the delta file is capable of handling the latest file that’s why this format is preferred



**Things to know:**

1. **What is .dfs?**

The `.dfs` in Databricks refers to Databricks File System (DBFS), which is a layer over a cloud object storage (like Amazon S3, Azure Blob Storage, etc.) that allows you to treat cloud storage like a file system. DBFS provides various benefits and features in the context of Databricks:

1. \*\*Unified Data Access\*\*: DBFS offers a unified view of data stored across various cloud storage services, simplifying data access and management. It allows you to access data using directory and file-like semantics, similar to a traditional file system.

2. \*\*Compatibility and Integration\*\*: It provides compatibility with traditional file-based APIs. This means you can use standard file I/O APIs in your programming languages (like Python, Scala, or R) to interact with data stored in DBFS.

3. \*\*Mount Points\*\*: DBFS allows you to mount cloud storage (like S3 buckets or Azure Blob containers) into the DBFS namespace. This means you can access data stored in these services as if they were directories in a file system.

4. \*\*Performance Optimization\*\*: DBFS is optimized for the performance characteristics of cloud storage and big data workloads, offering improved read/write performance compared to accessing cloud storage directly.

5. \*\*Databricks Notebooks and Jobs Integration\*\*: DBFS is seamlessly integrated into Databricks notebooks and jobs, making it easy to read and write data for your analytics and machine learning tasks.

6. \*\*Security\*\*: DBFS integrates with Databricks' security model, allowing for fine-grained access control to data stored in DBFS.

In practical terms, when you see a file path starting with `/dbfs/` or use the `%fs` magic command in a Databricks notebook, you are interacting with the Databricks File System. For example, reading a file from `/dbfs/mnt/my-data/file.csv` would access the file stored in the mounted cloud storage, abstracting away the underlying cloud storage details.