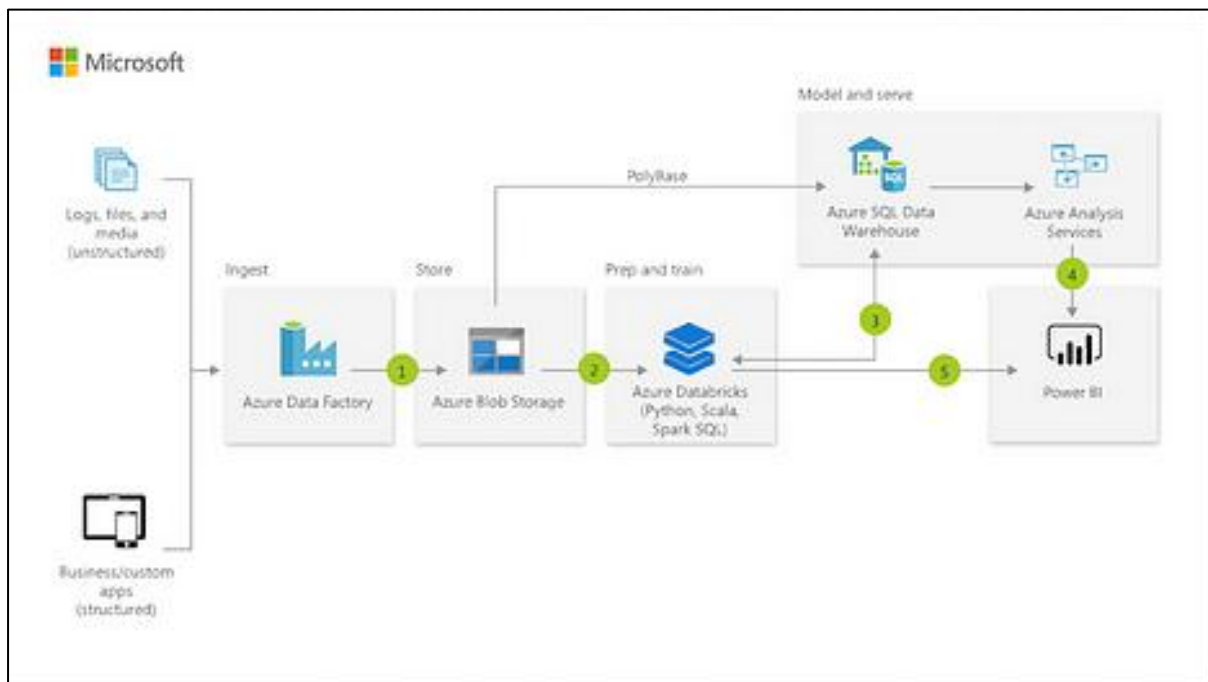


# Azure End-to-End Data Engineering Project



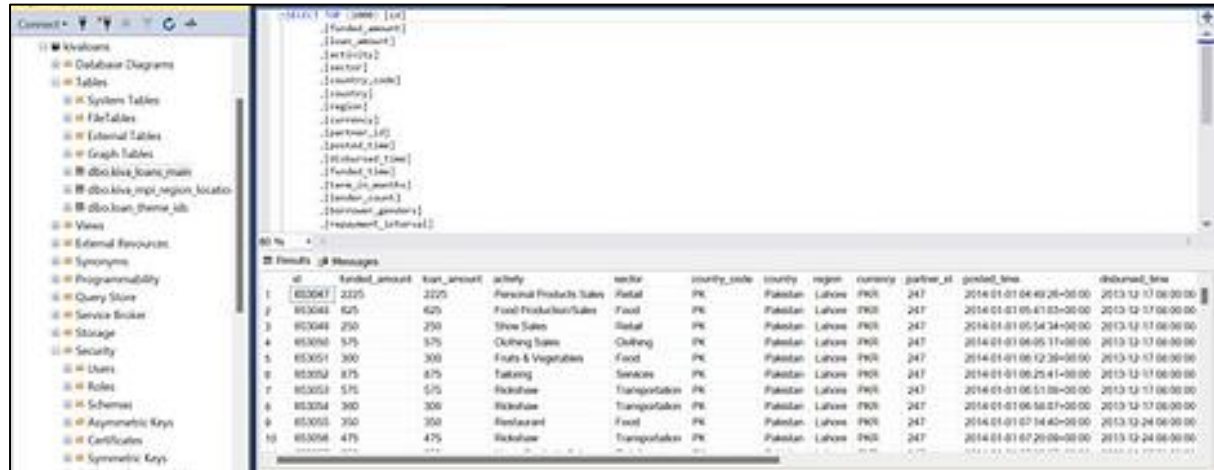
## What is Data Engineering?

Data engineering refers to designing and building systems that collect, process, store, and analyze data at different scales. This data usually comes from different sources at different times, hence the need to ensure proper cleaning and transformation before making it available to data scientists and analysts. The common task for data engineers involves creating data pipelines to allow streaming data for different use cases. The standard data pipeline phases include:

- 1. Ingestion:** This phase involves gathering data from various sources.
- 2. Processing:** This stage involves processing the data to desired data sets for different use cases.
- 3. Storing:** The processed data is then stored for easy retrieval from the database.
- 4. Access:** The stored data is accessible to users for different use cases.

## Azure End-to-End Data Engineering

I will use Microsoft Azure to demonstrate a real-world application of data engineering. I will also use an open-source dataset from the Kaggle competition “Data Science for Good: Kiva Crowdfunding”. This data is in CSV format; hence, you need to load the data into the database for storage. I assumed I would be receiving data daily to add to the different tables in the database. I used the SQL server database as the data source for the project.



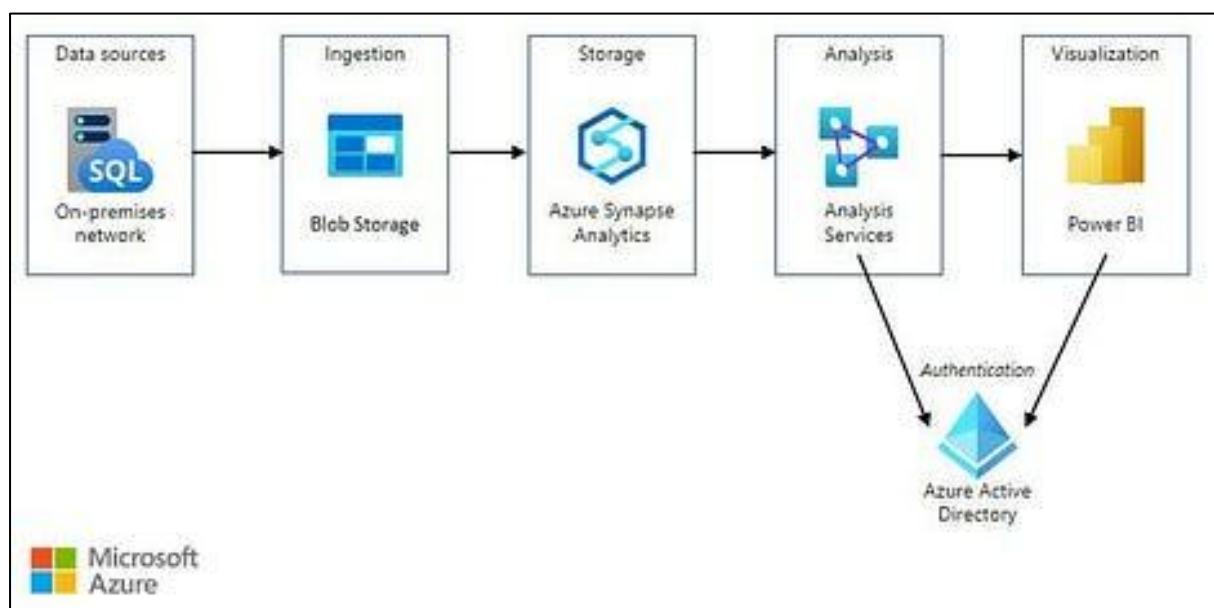
The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Server Enterprise Explorer' tree displays the database structure, including tables like 'dbo.kiva\_loan\_main', 'dbo.kiva\_mpi\_region\_location', and 'dbo.kiva\_theme\_id'. The main pane shows a query result set for a table named 'kiva\_loan\_main'. The query is a SELECT statement with columns: id, funded\_amount, loan\_amount, activity, sector, country\_code, country, region, currency, partner\_id, posted\_time, disbursed\_time, term\_in\_months, lender\_count, borrower\_gender, and repayment\_interval. The results table shows 10 rows of data, including loan details for various countries like Pakistan and India.

id	funded_amount	loan_amount	activity	sector	country_code	country	region	currency	partner_id	posted_time	disbursed_time	term_in_months	lender_count	borrower_gender	repayment_interval
1	600041	2225	2225	Personal Products Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 04:49:26+00:00	2013-12-17 00:00:00				
2	600044	625	625	Food Production/Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 05:47:53+00:00	2013-12-17 00:00:00				
3	600044	250	250	Shoe Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 05:54:34+00:00	2013-12-17 00:00:00				
4	600046	575	575	Clothing Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:05:11+00:00	2013-12-17 00:00:00				
5	600051	300	300	Fruits & Vegetables	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:12:39+00:00	2013-12-17 00:00:00				
6	600052	875	875	Farming	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:25:41+00:00	2013-12-17 00:00:00				
7	600053	575	575	Food/Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:51:09+00:00	2013-12-17 00:00:00				
8	600054	300	300	Food/Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:58:07+00:00	2013-12-17 00:00:00				
9	600055	350	350	Food/Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 07:14:40+00:00	2013-12-24 00:00:00				
10	600056	475	475	Food/Sales	PK	Pakistan	Lahore	PKR	247	2014-01-01 07:29:09+00:00	2013-12-24 00:00:00				

Kiva is an online crowdfunding organization that offers loan services to underserved and unbanked populations worldwide. Some of the use cases of these loans include starting a business, paying school fees, and investing in farming. The datasets available at Kiva show the loans issued to borrowers between January 2014 and July 2017. This information will be valuable in defining some Key Performance Indicators (KPIs) to evaluate the organization’s performance over time.

## The Data Architecture

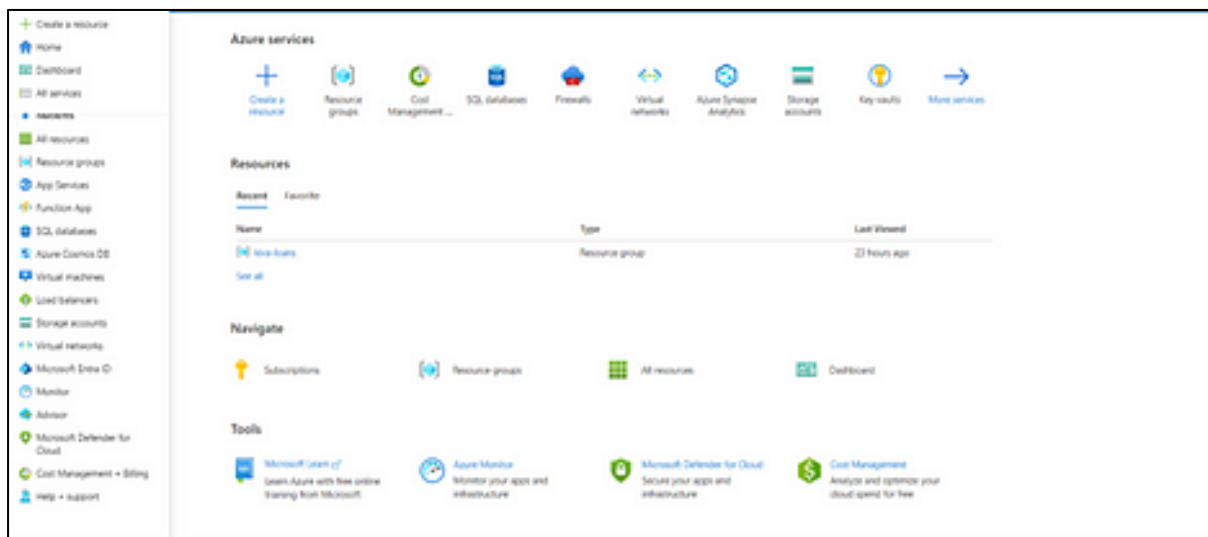
I used data stored on our On-prem SQL server database. The project aimed to create a data pipeline that migrated the data into the cloud from the SQL server. Therefore, I would trigger this pipeline to run daily when new data is added to the database.



- 1. On-prem SQL Server Database:** I used data stored in our local SQL database in the project.
- 2. Azure Data Factory:** ADF is the ETL tool used for data ingestion. I connected the ADF to the SQL server to copy the required tables to the cloud.
- 3. Azure Data Lake Gen 2:** This is where all the data are stored.
- 4. Azure Databricks:** Azure Databricks transformed the data into the required format.
- 5. Azure Synapse Analytics:** This tool will load the Azure SQL database data.
- 6. Power BI:** To load data from Azure Synapse Analytics and Create reports for business use cases.
- 7. Azure Active Directory and Azure Key Vault:** To enhance security access to the different data in the cloud.

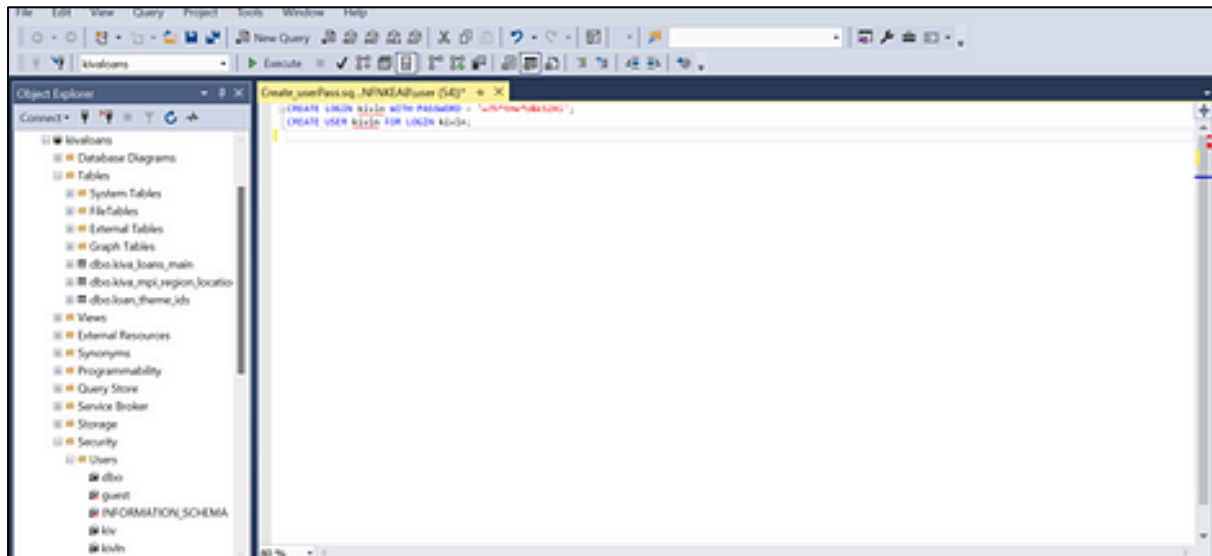
## Environment Set-up

One requires a Microsoft Azure account for this project. Microsoft offers beginners a free 30-day trial period to access the Azure portal. Inside the portal, you need to create a resource group with a unique name. Furthermore, you need to create different resources in the resource group to enable the execution of the different data pipeline runs.

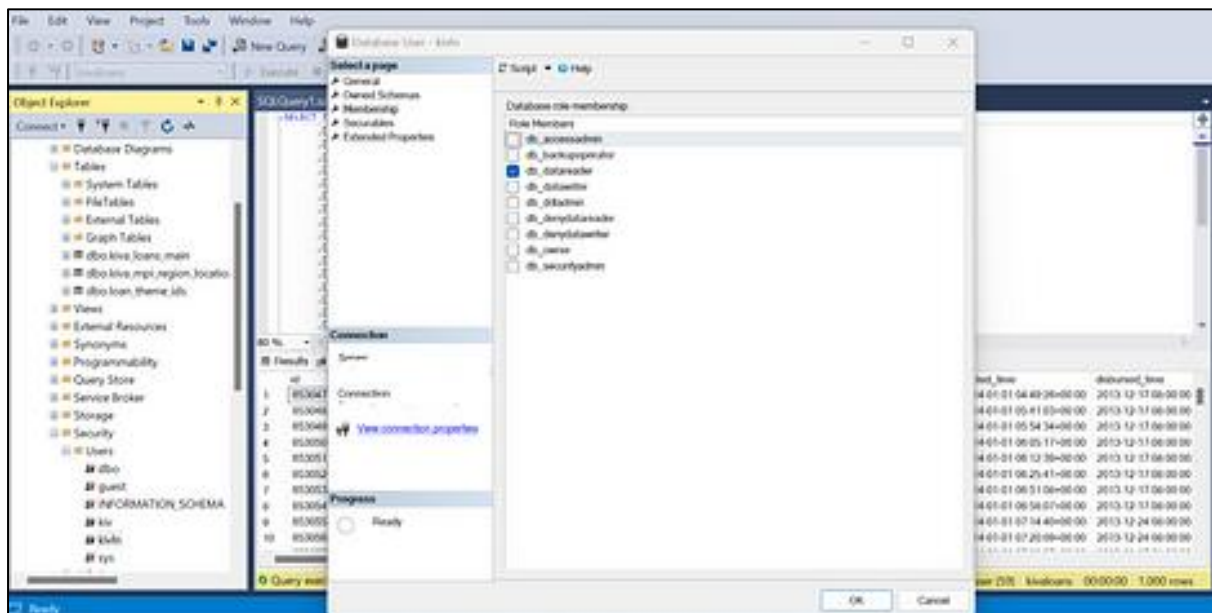


## Data Source

In this project, I created a username and password to allow access to the Azure database. Besides, I stored the username and password in the Azure Key vault for secure login to access the data.



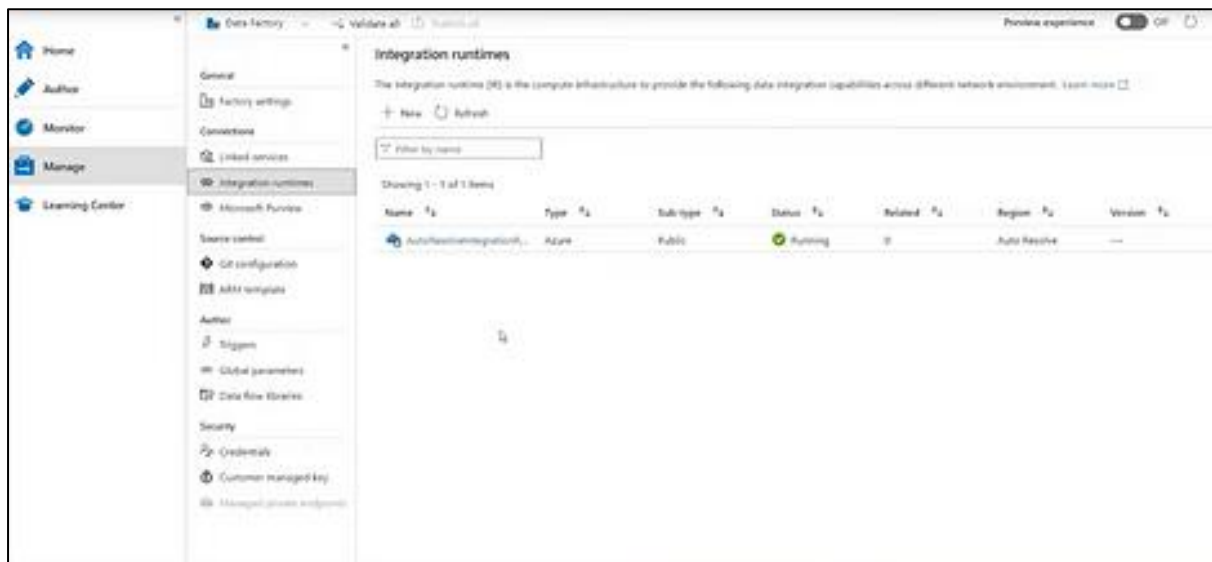
I also granted the user access to the database to read all the tables with the prefix 'dbo' since I will read the tables with the 'dbo' schema to Azure. To check the user permissions, expand the security panel on the left side under the particular database, expand the user, and refresh to show the new user created. Right-click on the user name, under the properties, grant the 'db\_datareader' for the database role membership.



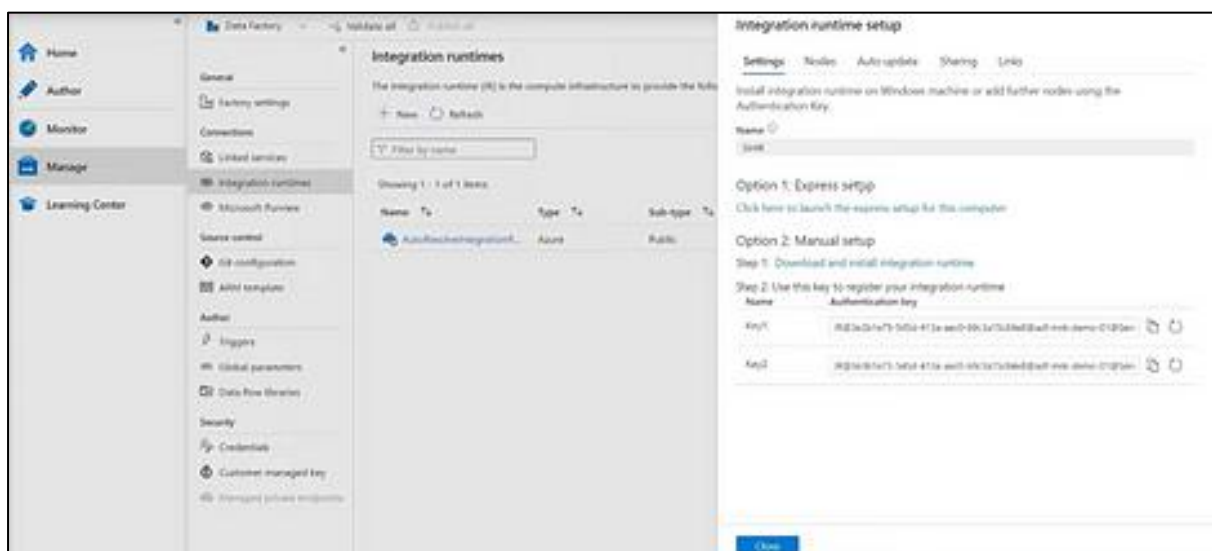
## Data Ingestion Using Azure Data Factory

### Creating a Connection Between SQL Server and Azure Data Factory

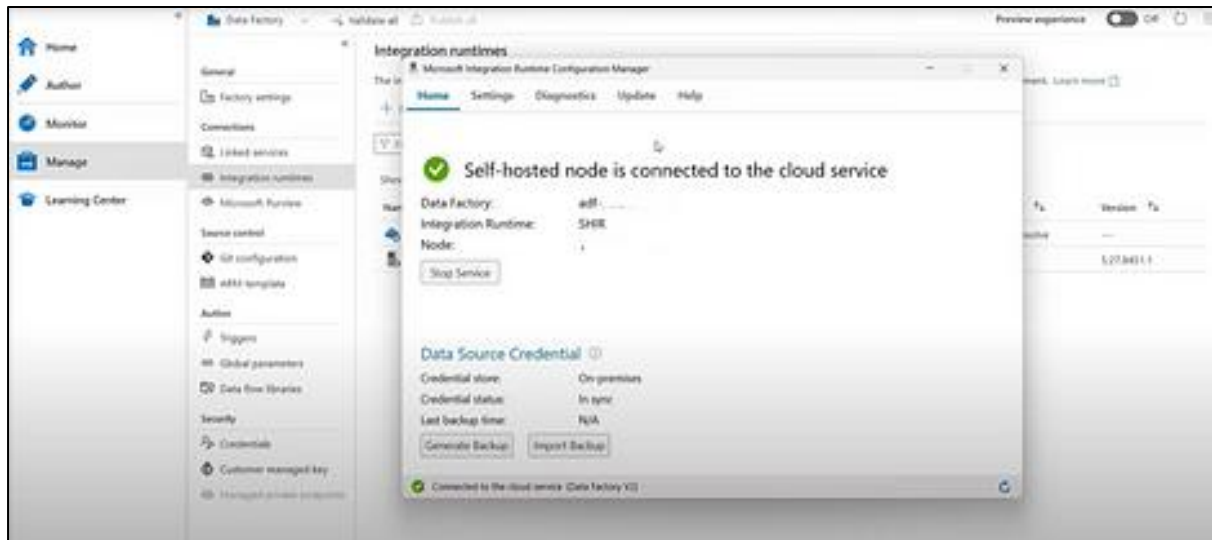
Since there was no relationship between Azure Data Factory and the SQL server database, I needed to install a Self-Hosted Integration Runtime to create the link. Under the manage tab in the ADF, select integration runtime. Under this option, there will be an AutoResolveIntegrationRuntime already running. This AutoResolveIntegrationRuntime is usually used to connect to cloud-based resources. However, I needed to create an integration runtime to connect to the on-prem SQL database.



To create the SHIR, select the new button and select Azure, Self-Hosted setup, give a descriptive name, and create the runtime. For this project, I used the express setup option since I was installing it on my machine only. However, one can use the manual setup option when installing the SHIR on multiple machines for the same project.

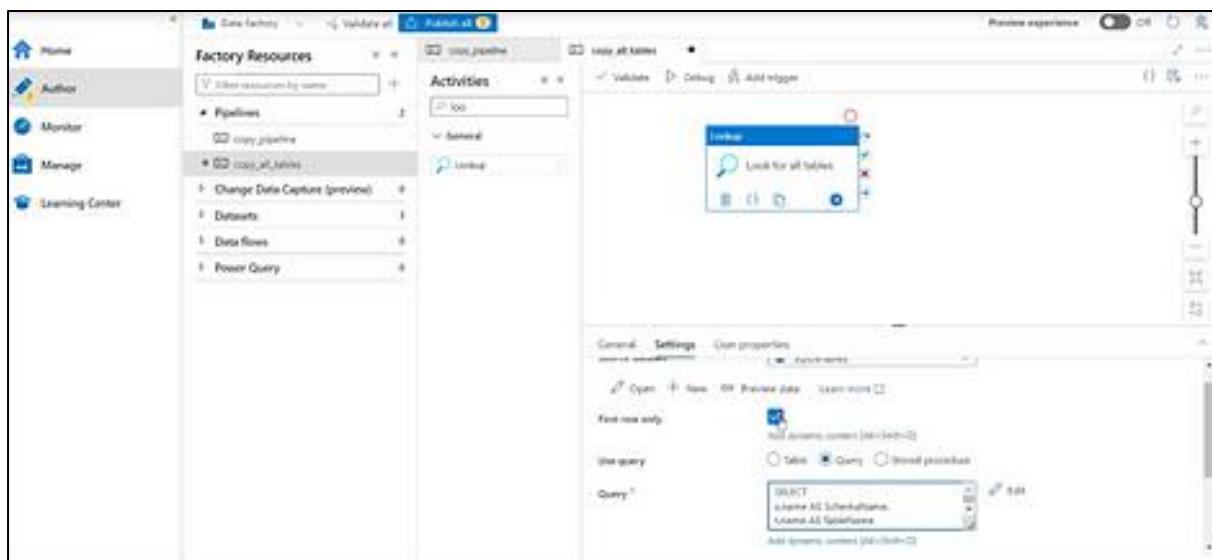


On successful installation, run the newly installed Microsoft Integration Runtime Configuration Manager to confirm if the connection is established. A successful installation should show the connection as:

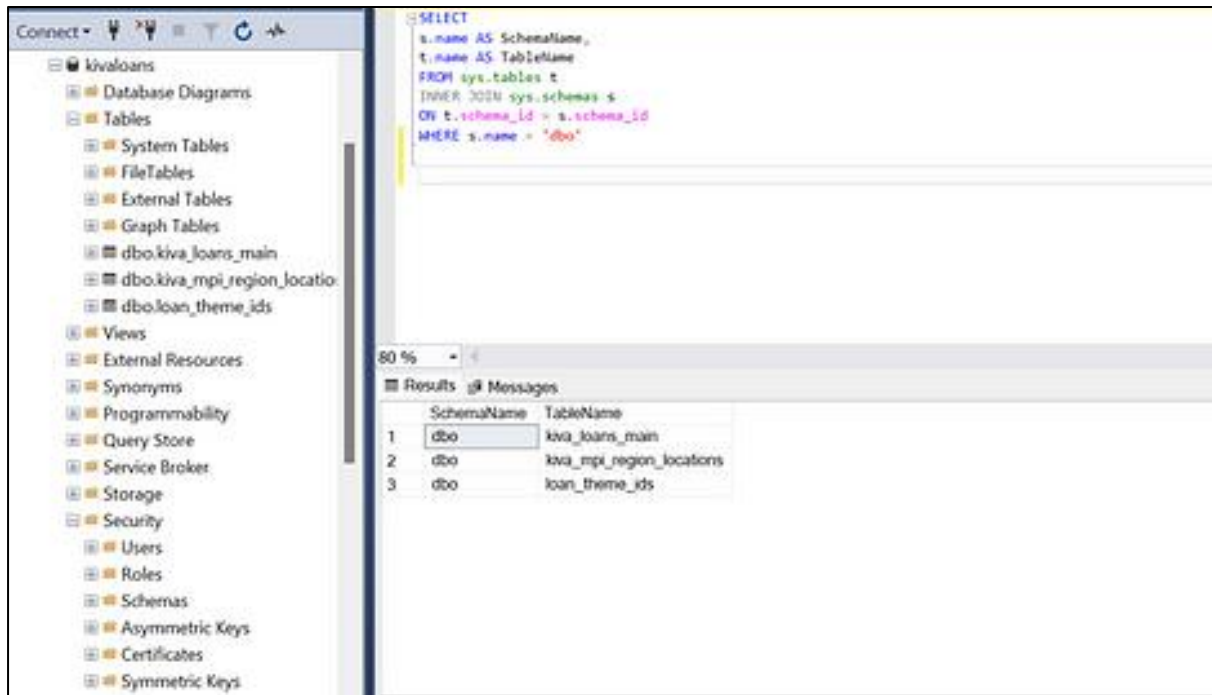


## Data Ingestion

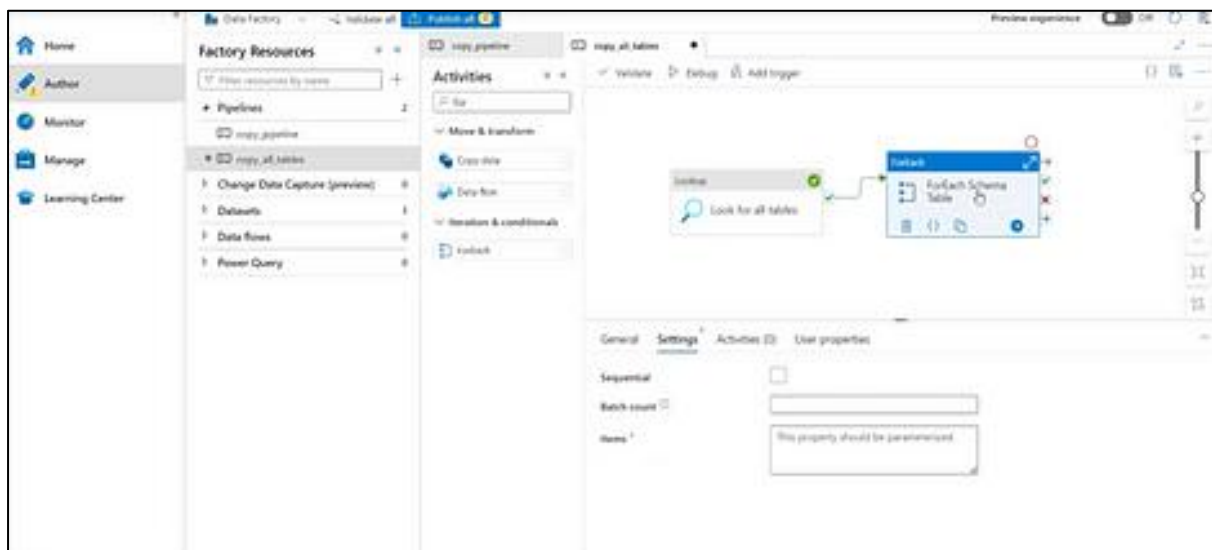
For this project, I was interested in creating a pipeline that would read all the tables in the SQL server and load them in Azure Data Lake Gen 2. I created three storage containers, (gold, silver, and bronze) where the bronze container will store the raw data as it was from the source. The silver container would contain data from level 1 transformation from the bronze container while the gold container would contain the cleaned and transformed data from the silver container. To create a pipeline to read all the tables in the database, I used the lookup activity to look up all the tables from the SQL database. In this lookup activity, I use a query to list the tables to ingest in Azure Data Lake Gen 2.



This query would list the tables available in the database with the given schema name.

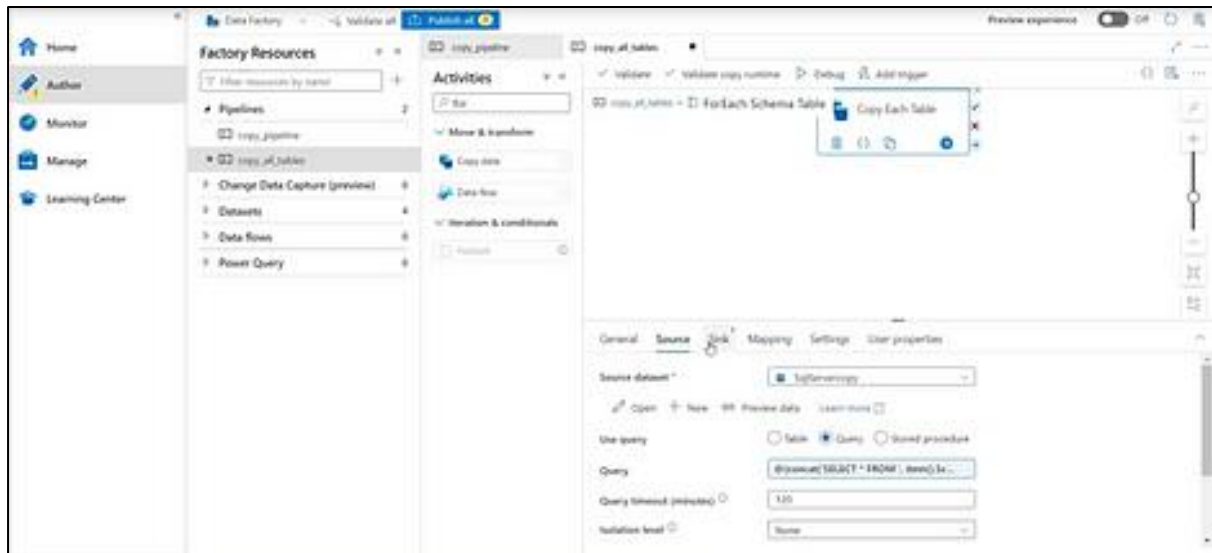


Thereafter, I created a ForEach item in the pipeline to iterate from the Lookup activity to copy all the tables. The ForEach item would loop through all the tables and copy the items listed from the lookup table.



Under the ForEach item, I created the Copy data item to copy data from the source. I configured the source of the data and the sink to save the data as CSV files. I also specified the path of the datasets so that each copied table would be stored separately in a folder and a path with the file name. Furthermore, the data ingestion pipeline was used to read and store the data from the SQL server to the bronze container, where the data was in the raw format.



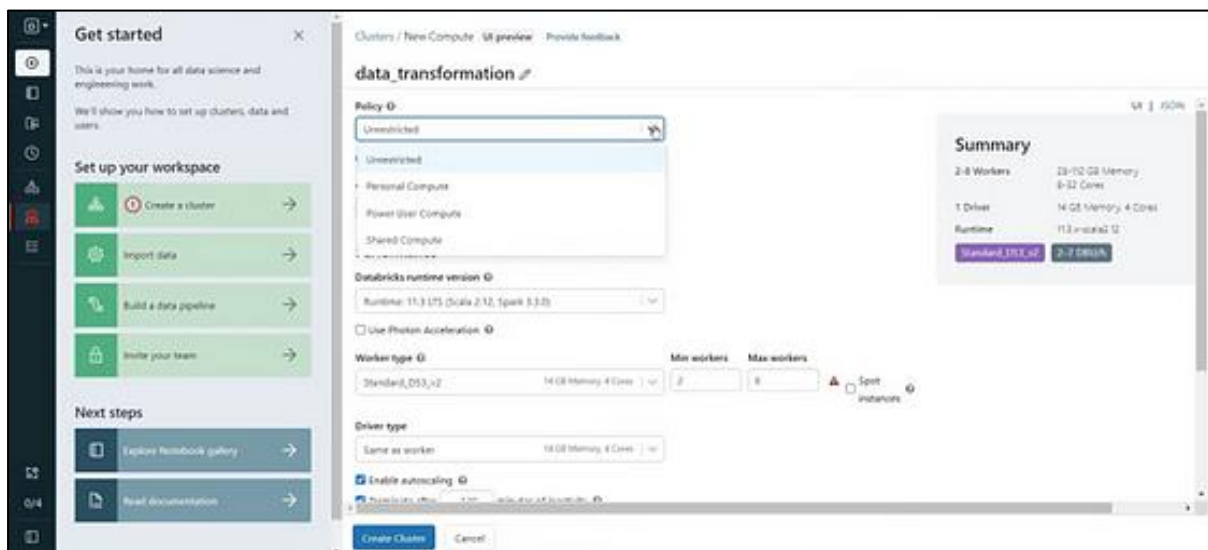


After the successful configuration of the pipeline, I ran using the debug option to test whether it was working without any errors. Besides, I used the add trigger option to run the pipeline and tested with the current time. Since the pipeline was meant to fetch the tables from the on-prem SQL server every time it runs, every successful run would overwrite the existing folder in the bronze container. At the end of the data ingestion, the data was now stored in the bronze container ready for transformation.

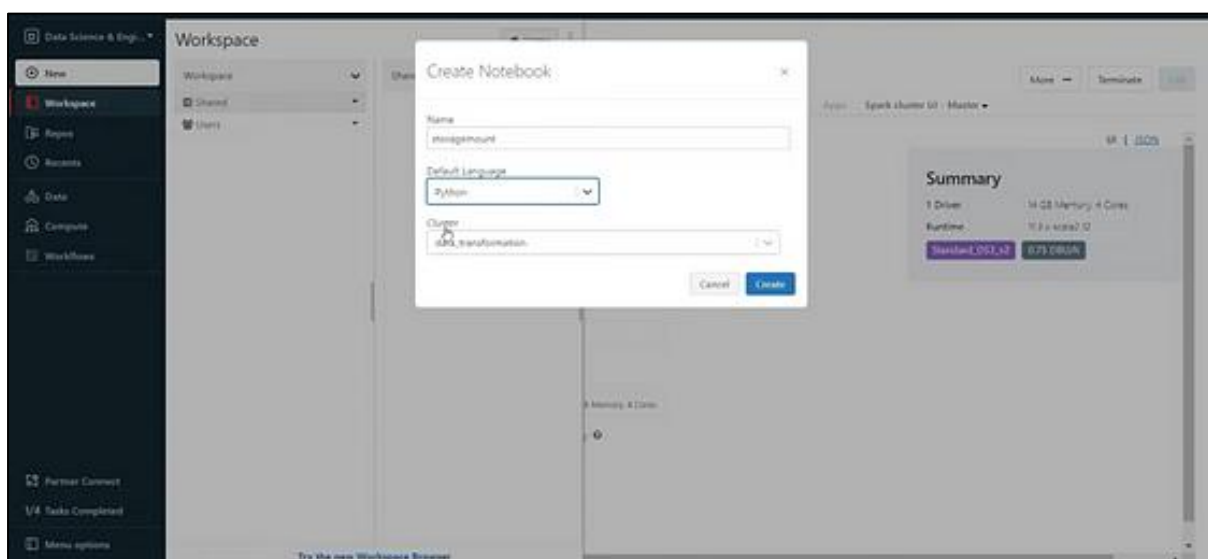
## Data Transformation Using Azure Databricks

### Storage Mount

After loading data from the SQL server to the bronze container, I performed two levels of transformations of the datasets into cleaner versions. However, I needed to create a compute cluster to allow different jobs to run in the notebooks. To create the compute cluster, launch the Azure Databricks studio, click on the compute tab, and configure the compute instance according to your needs. I selected a single user since I was the only one using the cluster in the resource and selected the other default options. It is also important to note that I enabled 'credential passthrough for user-level data access' to allow Azure Databricks to connect to Azure Data Lake Gen 2.



In the Databricks workspace, create a notebook called 'storagemount' to mount the storage account in Azure Databricks. This option will allow accessing the files in the storage account.



To mount the storage using credential passthrough, there are codes from this webpage (<https://learn.microsoft.com/en-us/azure/databricks/archive/credential-passthrough/adls-passthrough>), and made some changes according to my configuration.

```
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://bronze@kivaloansfolder.dfs.core.windows.net/",  
  mount_point = "/mnt/bronze",  
  extra_configs = configs)
```

For this code, update the container and storage account names to match what is on the resource group. Also, change the mount point name with that of the container.

Similarly, I mounted both the silver and gold containers. This option allowed accessing data from these different containers. Furthermore, I could access the data by giving the full path for the containers since I had allowed credential passthrough.

```
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://silver@kivaloansfolder.dfs.core.windows.net/",  
  mount_point = "/mnt/silver",  
  extra_configs = configs)
```

```
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://gold@kivaloansfolder.dfs.core.windows.net/",  
  mount_point = "/mnt/gold",  
  extra_configs = configs)
```

## Level 1 Data Transformation

Since the data looked relatively clean in the database, I focused the first level of transformation on changing the date format from date-time to date type. At this level, I wanted the transformation to apply to all the tables and the columns of the date data type.

Results Messages												
	id	funded_amount	loan_amount	activity	sector	country_code	country	region	currency	partner_id	posted_time	disbursed_time
1	653047	2225	2225	Personal Products Sales	Retail	PK	Pakistan	Lahore	PKR	247	2014-01-01 04:49:26+00:00	2013-12-17 08:00:00
2	653048	625	625	Food Production/Sales	Food	PK	Pakistan	Lahore	PKR	247	2014-01-01 05:41:03+00:00	2013-12-17 08:00:00
3	653049	250	250	Shoe Sales	Retail	PK	Pakistan	Lahore	PKR	247	2014-01-01 05:54:34+00:00	2013-12-17 08:00:00
4	653050	575	575	Clothing Sales	Clothing	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:05:17+00:00	2013-12-17 08:00:00
5	653051	300	300	Fruits & Vegetables	Food	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:12:39+00:00	2013-12-17 08:00:00
6	653052	875	875	Tailoring	Services	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:25:41+00:00	2013-12-17 08:00:00
7	653053	575	575	Rickshaw	Transportation	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:51:08+00:00	2013-12-17 08:00:00
8	653054	300	300	Rickshaw	Transportation	PK	Pakistan	Lahore	PKR	247	2014-01-01 06:58:07+00:00	2013-12-17 08:00:00
9	653055	350	350	Restaurant	Food	PK	Pakistan	Lahore	PKR	247	2014-01-01 07:14:40+00:00	2013-12-24 08:00:00
10	653056	475	475	Rickshaw	Transportation	PK	Pakistan	Lahore	PKR	247	2014-01-01 07:20:09+00:00	2013-12-24 08:00:00

First, I created another notebook named 'bronze\_to\_silver' for this first transformation. I created an array to list the table names in the bronze container. This code iterates through the bronze container to get the directory name and append the directory name to the table name array.

```
table_name = []
for i in dbutils.fs.ls('mnt/bronze/dbo/'):
    table_name.append(i.name.split('/')[0])
```

The code below generates the input path for all the tables iteratively and then loads as a data frame. The code declares the variable column, which gets the column names from the data frame as a list. I ensured that the file type was specified as CSV and the header was set as true to return the tables with header names. The next loop in the code checks if any column name has the name 'date' or 'time' and then converts it to the date format only. The last part of the code generates the output path in the silver container using the table name and writing the transformed data in the data lake. I specified the file type as CSV and used the overwrite option to rewrite any data in the silver container with the new version.

```
from pyspark.sql.functions import from_utc_timestamp, date_format
from pyspark.sql.types import StructType, StructField, StringType, TimestampType

for i in table_name:
    input_path = '/mnt/bronze/dbo/' + i + '/' + i + '.csv'
    df = spark.read.format('csv').option('header', 'true').load(input_path)

    for col in df.columns:
        if "date" in col or "time" in col:
            df = df.withColumn(col, date_format(from_utc_timestamp(df[col].cast(TimestampType()),
"UTC"), "yyyy-MM-dd"))

    output_path = '/mnt/silver/dbo/' + i + '/'
    df.write.format('csv').mode("overwrite").option('header', 'true').save(output_path)
```

## Level 2 Data Transformation

After performing the first level of transformation to the data, the data in the silver container appeared cleaner but required further transformation to be ready for different use cases. Thus, I performed another transformation from data in the silver container by changing the column names to lowercase. This transformation was to ensure there is a uniform naming convention for the data across the different tables. The transformed data was then stored in different folders in the gold container with respective table names.

I created another notebook and named it 'silver\_to\_gold.' Similarly, I created an array listing table names in the silver container. This code iterates through the silver container to get the directory name and append the directory name to the table name array.

```
table_name = []
for i in dbutils.fs.ls('mnt/silver/dbo/'):
    table_name.append(i.name.split('/')[0])
```

The code below generates the input path for all the tables iteratively and then loads as a data frame. The code declares the variable `column_names`, which gets the column names from the data frame as a list. I ensured that the file type was specified as CSV and the header was set as true to return the tables with header names. The next loop in the code iterates through the column names and changes to lowercase. The last part of the code generates the output path in the gold container using the table name and writing the transformed data in the data lake. I specified the file type as CSV and used the overwrite option to rewrite any data in the gold container with the new version.

```
for name in table_name:
    input_path = '/mnt/silver/dbo/' + name # Fix the input_path variable
    print(input_path)
    df = spark.read.format('csv').option('header', 'true').load(input_path)

    # Get the list of column names
    column_names = df.columns

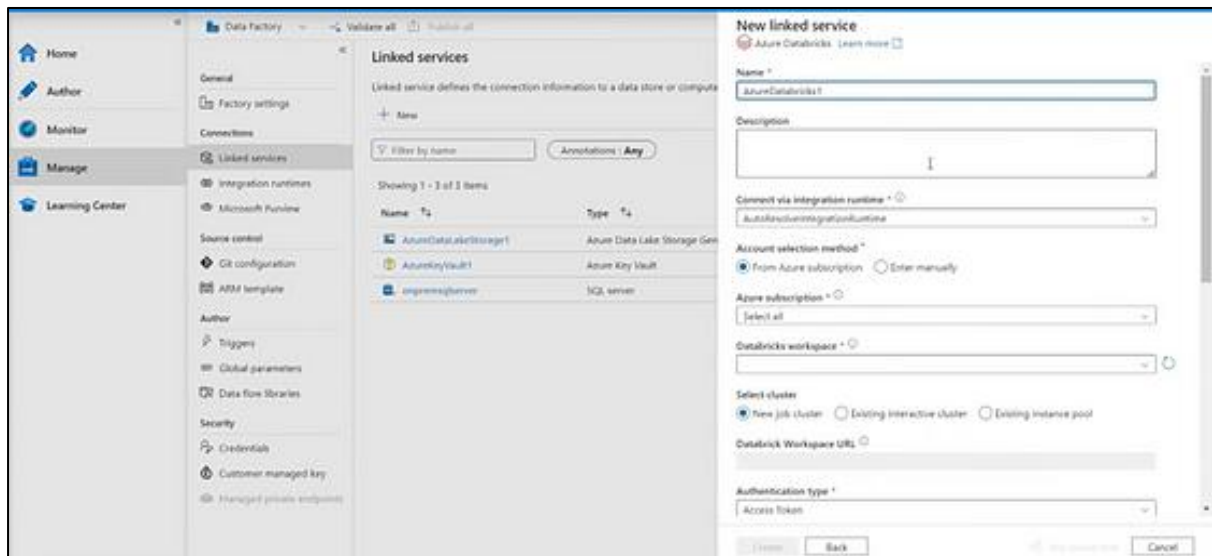
    for old_col_name in column_names:
        # Change the column names to lowercase
        new_col_name = old_col_name.lower()

        # Change the column name using withColumnRenamed
        df = df.withColumnRenamed(old_col_name, new_col_name)

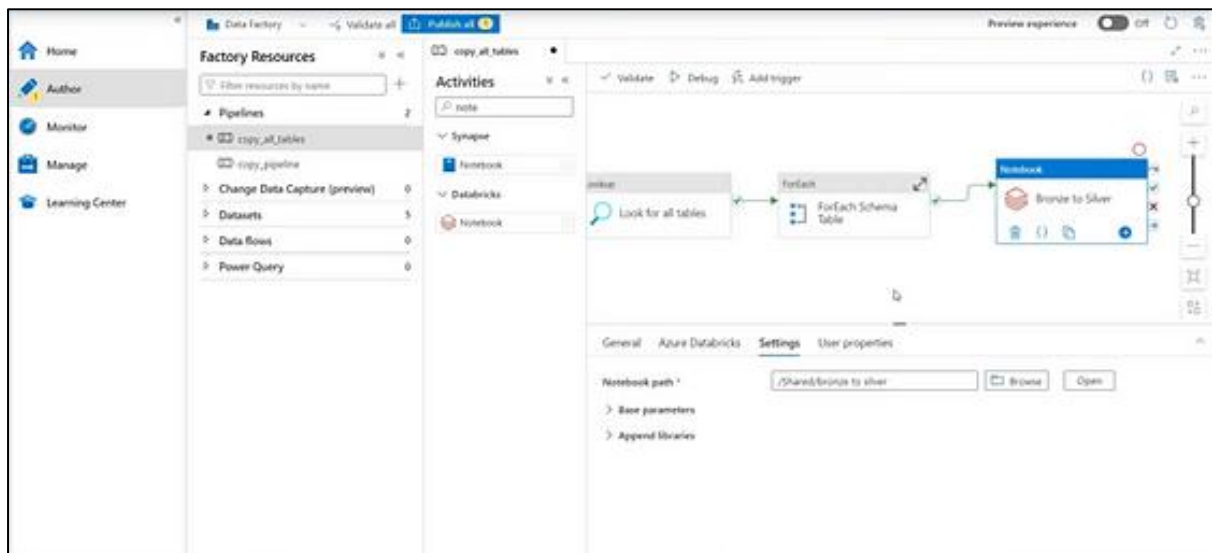
    output_path = '/mnt/gold/dbo/' + name + '/'
    df.write.format('csv').mode("overwrite").option('header', 'true').save(output_path)
```

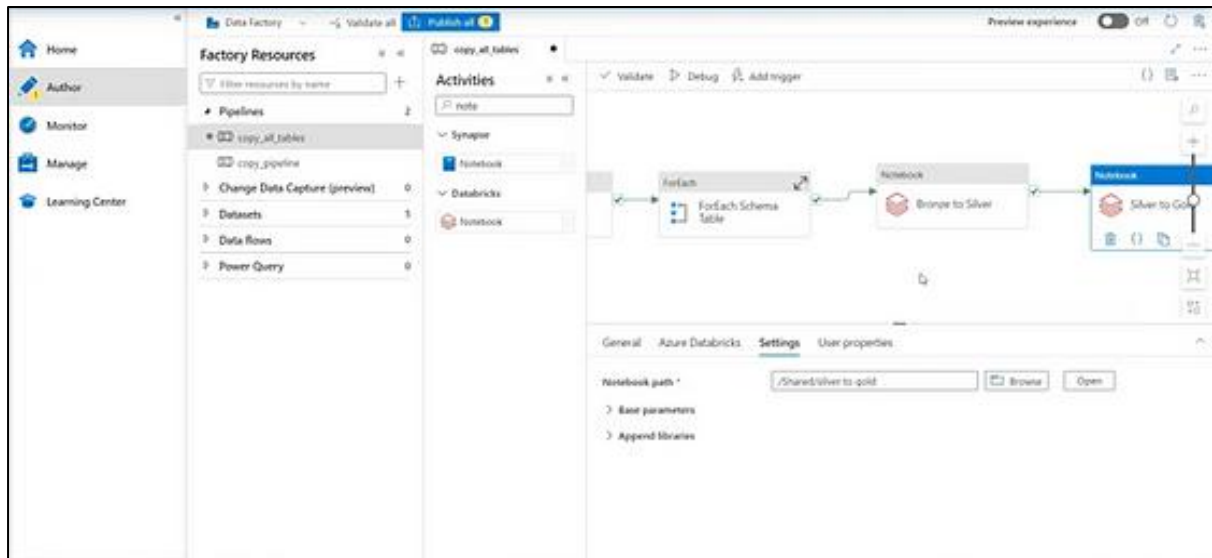
## Data Transformation Pipeline

Since the two notebooks perform the necessary transformation for data, I needed to add them to the Azure Data Factory pipeline to transform the data when triggered with daily additional data. To achieve the connection between Azure Databricks and Azure Data Factory, I created a new linked service and used an access token to authenticate the linked service.



After creating the linked service, I selected the author tab and looked for the pipeline I had created earlier to add the two notebooks. I added the Azure Databricks notebook item from this tab to add the new activity for the bronze to the silver notebook. I specified the path of the notebook and connected the output of the ForEach activity to the bronze-to-silver notebook. This link ensured the notebook would only run if the ForEach activity were successful. Similarly, I added another notebook item for the silver-to-gold activity to run after the successful run of the bronze-to-silver notebook.

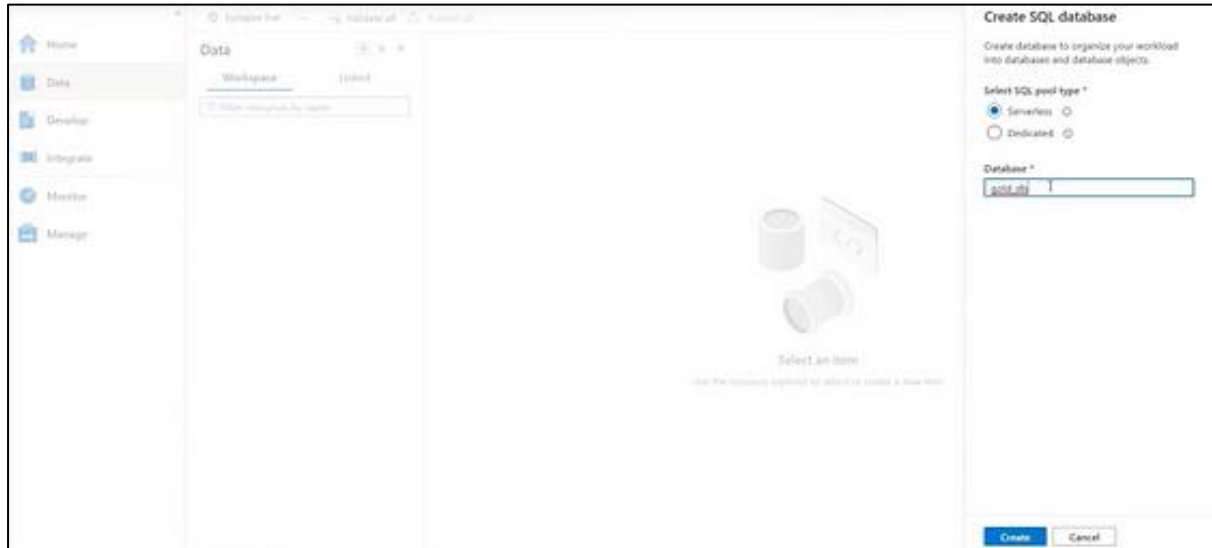




I published all the changes and used the debug option to run the pipeline. Similarly, I added a new trigger to run the pipeline to confirm if it properly worked, where I confirmed each activity ran successfully from data ingestion to processing.

## Data Loading Using Azure Synapse

With the transformed data in the gold container, I needed to load the data in the Azure SQL database for access and reporting. Therefore, I created a serverless database to store the data and named the database 'gold\_db.' The Azure Synapse Analytics workspace is linked to the Azure Data Lake Gen 2; hence, one can find data from the different containers in the linked datasets.



Since there were different tables in the gold container, I needed to create views for the Azure SQL database. Therefore, I created a stored procedure using a new SQL script to create views of the different tables.

```
USE gold_db
GO
```

```
CREATE OR ALTER PROC CreateSQLServerlessView_gold @Viewname nvarchar(100)
AS
BEGIN
```

```
DECLARE @statement VARCHAR(MAX)
```

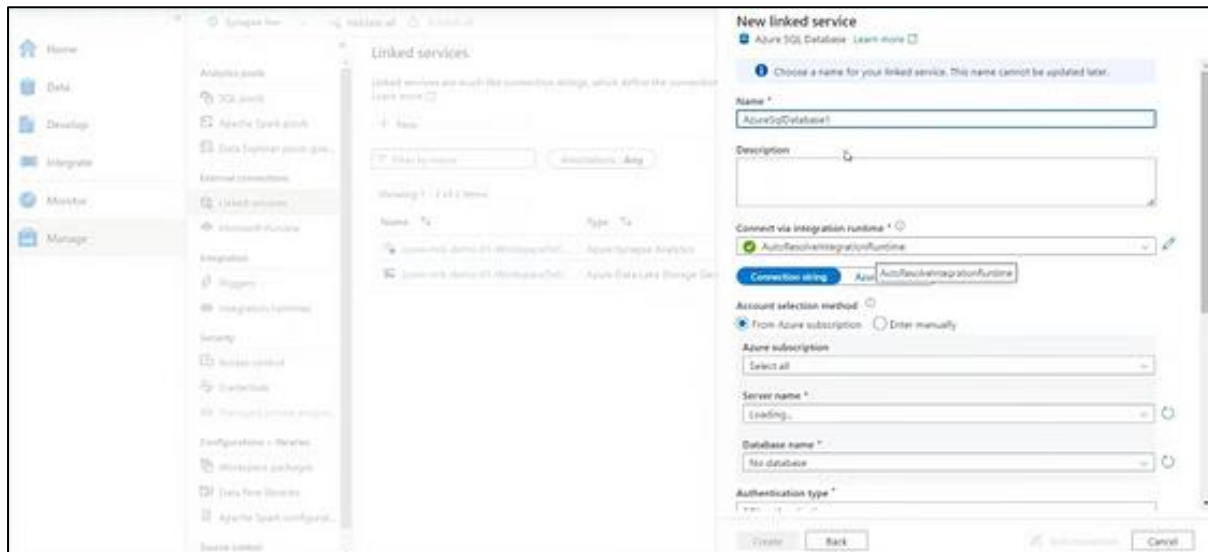
```
SET @statement = N'CREATE OR ALTER VIEW ' + @Viewname + ' AS
SELECT *
FROM
OPENROWSET(
BULK "https://kivaloansstg.dfs.core.windows.net/gold/dbo/' + @Viewname + '/',
FORMAT = "CSV"
) AS [result]
'
```

```
EXEC (@statement)
```

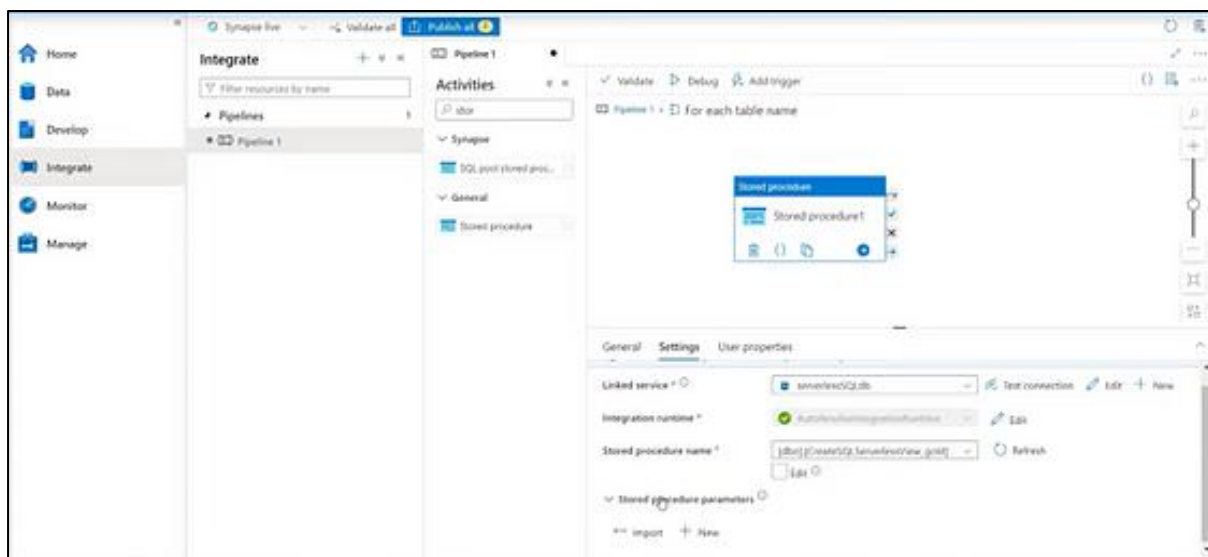
```
END
GO
```



I also needed to create a pipeline to use the stored procedure (SQL query) to create views in the SQL serverless database. Under the manage tab, I made a new linked service for the 'Azure SQL Database' to access the serverless SQL database. Note the integration runtime is automatically selected as 'AutoResolveIntegrationRuntime' since it is a cloud-based resource. Configure the account selection manually and use the 'System Assigned Managed Identity' authentication type.



After creating the linked service, I made a new pipeline and added the 'Get Metadata' item to get the table names from the gold container. Similarly, I added a 'ForEach' item to get each item from the metadata and selected the stored procedure name from the existing list.

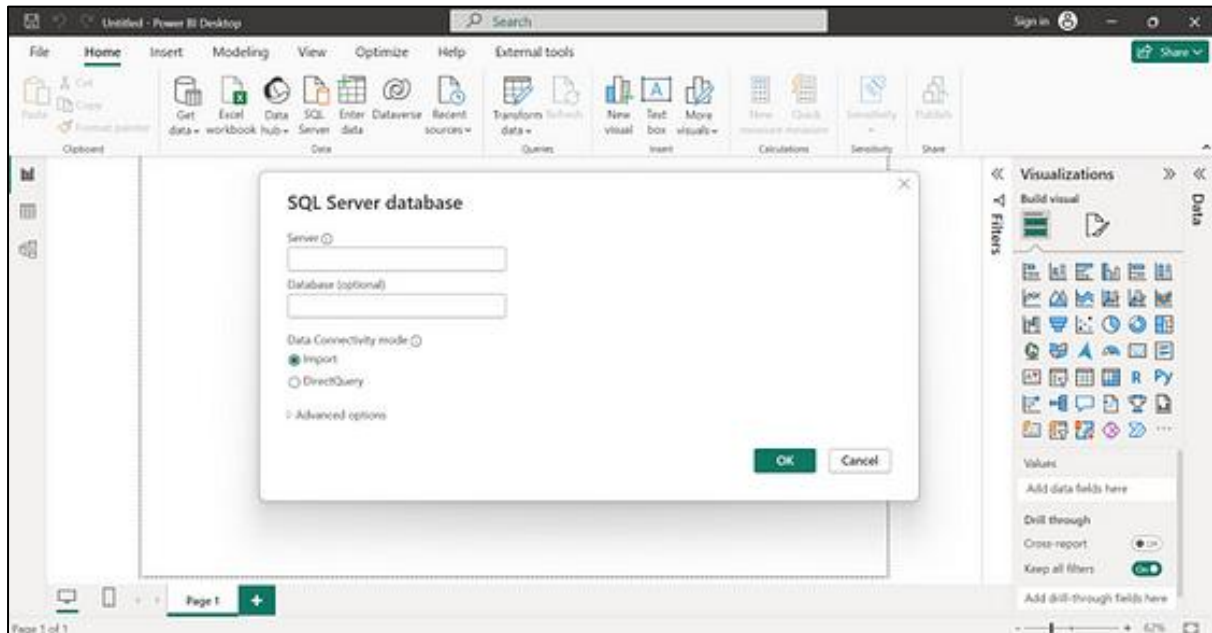


After that, I published all the changes and ran the pipeline using the debug option. This option added the different views from the tables in the gold container to the Azure SQL database.

## Data Reporting using Power BI

In this last section of the project, I explain how I used Power BI to connect to the Azure Synapse SQL serverless database, fetched the views in the database, and created an interactive dashboard to report some of the KPIs for Kiva.

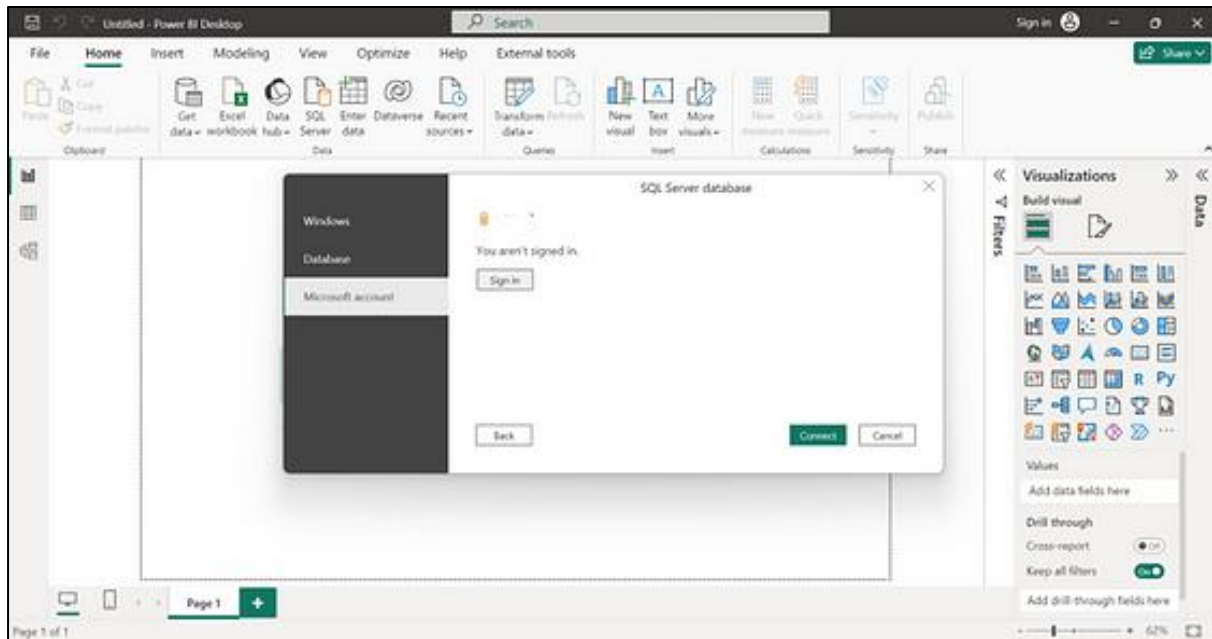
I used the 'Get data' option on the Power BI ribbon for this scenario and selected Azure. Under the Azure data source, I selected 'Azure Synapse Analytics SQL' and hit the connect button. This action prompts one to input the server name and database.



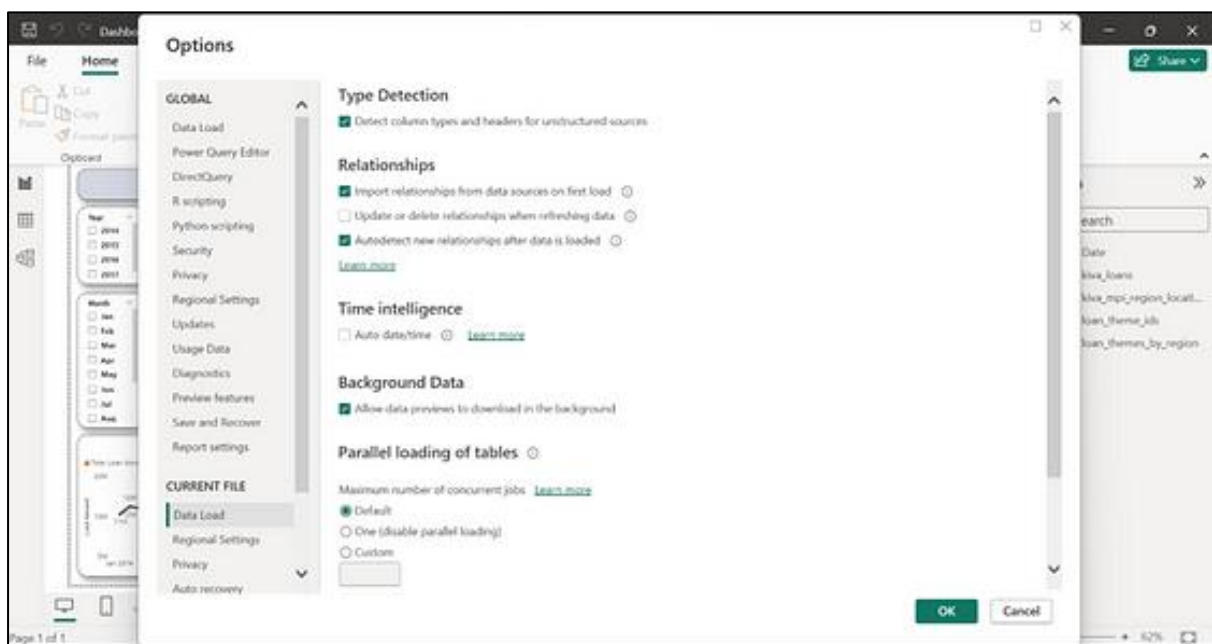
One can get the server's name by navigating to the Azure Synapse Analytics workspace, under the properties tab, and copying the 'Serverless SQL endpoint' as the server name. I also maintained the import data connectivity mode to load the data/tables in Power BI.



I used my Microsoft account to sign in and connect to the database.



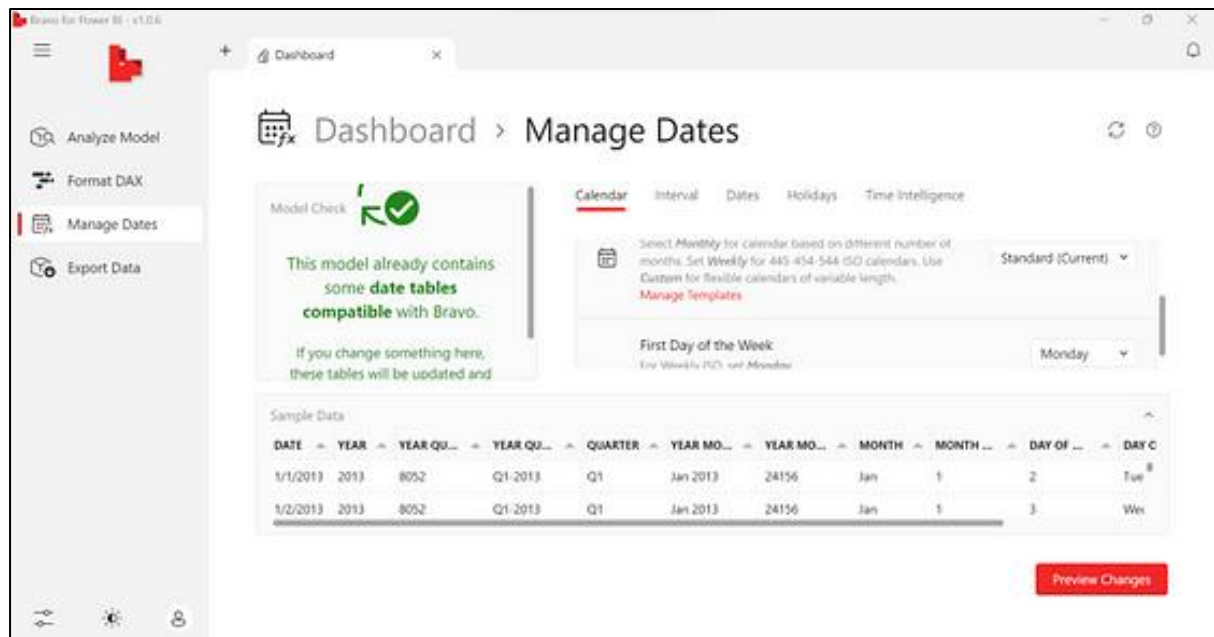
This action connected to the database and fetched the different views in the Azure SQL database as tables. I noticed an error in importing the files due to a conflict between the date format in the tables and my settings in Power BI. Therefore, I disabled the 'Time Intelligence' option under the 'Options' menu for the current file.



I performed some transformations on the tables using Power Query to ensure each column was of the right data type. I also converted the date column of the main table to the appropriate data type.

Since I was to use the dates to filter the data for some insights, I created the date table separately using Bravo, an external tool for Power BI. Bravo generates the dates automatically and creates the date table.

Refer to this video for using Bravo: (<https://www.youtube.com/watch?v=WybnTHDL-AM&t=326s>).

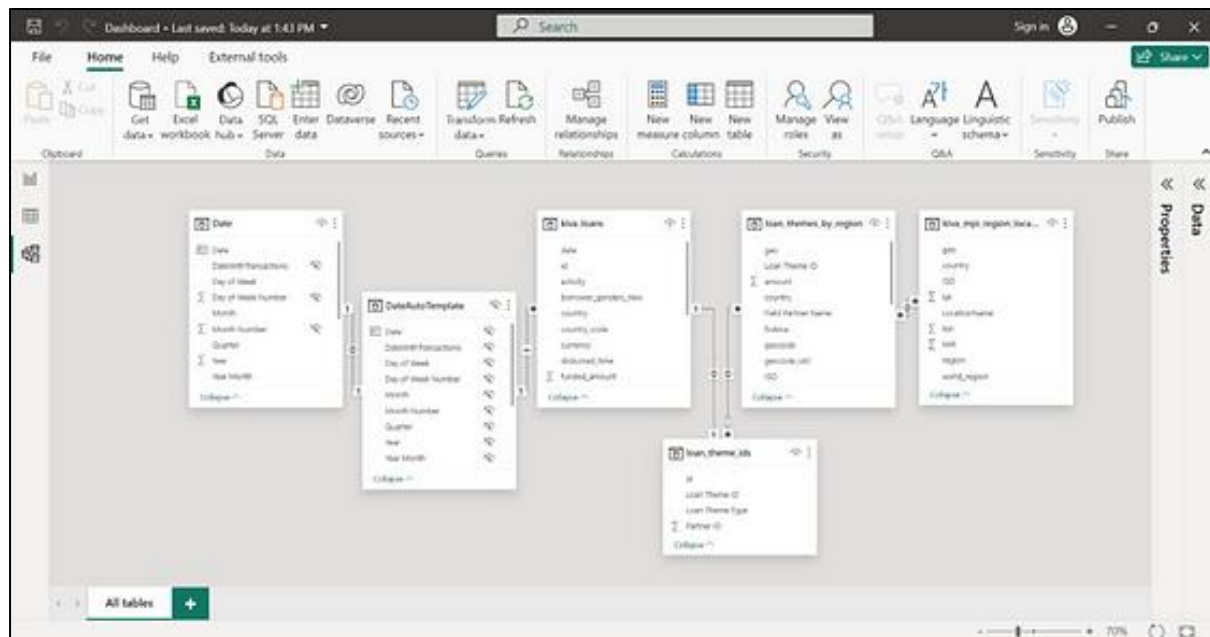


The resulting data table is shown below. The other tables remained the same.

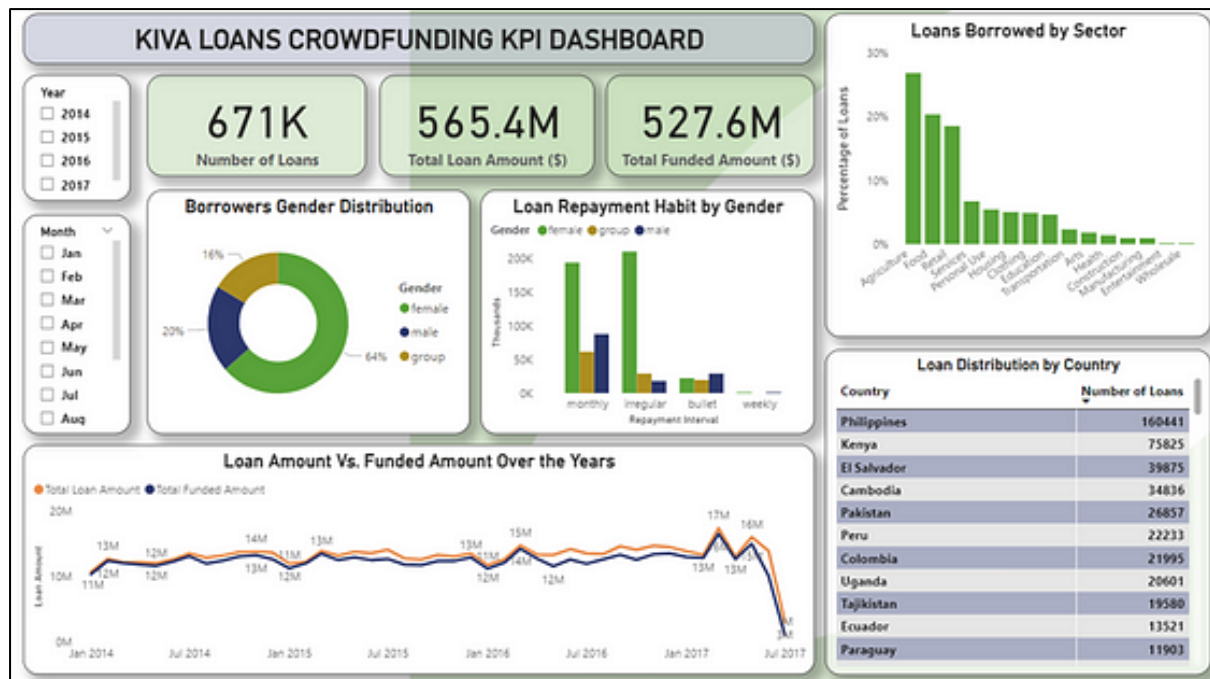
Table: Date (1,826 rows)

Date	Year	Year Quarter Number	Year Quarter	Quarter	Year Month	Year Month Number	Month	Month Number	Day of Week
7/1/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/2/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/3/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/4/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/5/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/6/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/7/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/8/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/9/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/10/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/11/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/12/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/13/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/14/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/15/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/16/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/17/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/18/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/19/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?
7/20/2016	2016	8066	Q3-2016	Q3	Jul 2016	24198	Jul	7	?

Before analyzing and building the visuals, I checked the modeling tab to ensure the tables had proper relations. One must manually create the relationship between the new date table and the date column of the 'kiva\_loans' table.



After confirming the tables had the appropriate relationships, I created the dashboard below for reporting.



According to this dashboard, I used three cards to display the important metrics. Using the unique ID from the 'kiva\_loans' table, I got the count of the total number of loans disbursed by Kiva. Besides, the total loan and total funded amounts are \$565.4 million and \$527.6 million, respectively. I also included a bar chart to show the percentage of loans borrowed by sector. This chart shows the sector where the loan borrowers intended to use the loans. Additionally, I included a doughnut chart to show the gender distribution of the borrowers. To achieve this chart, applied conditional formatting to the column borrowers' gender and categorized the 'male' as male, 'female' as 'female', and any other combination as a group. Therefore, the chart shows females were the most borrowers while people in group borrowers were the least.

I also included a clustered column chart to show the loan repayment habits of the borrowers based on gender. The chart indicates most of the borrowers' monthly repayments. However, some borrowers made irregular repayments making it the second-most repayment routine. Still, the bullet and weekly repayment intervals were the least preferred repayment intervals by the borrowers. I also included a table to show the loan distribution by country. The table is in descending order and shows the top two countries, the Philippines and Kenya. The line chart shows the total loan amount vs the funded amount from January 2014 to July 2017. There is a significant trend and difference between these two metrics over the years and different months.

From the data table, I included the 'Year' and 'Month' slicers to help filter the dashboard according to specific periods. This approach would allow users to check the different metrics and compare the trends throughout the various months.

### **Pipeline Testing**

I tested this pipeline by adding 10 random rows to the main table and triggering the pipeline to run. I also created a scheduled trigger to run once daily as I added ten rows of data in the original table for five days. After the successful run of each trigger, I refreshed the Power BI report, which changed to reflect the new data according to the number of loans added. This approach showed the data engineering pipeline worked as scheduled.

### **Conclusion**

According to this project, I have showcased how I used Microsoft Azure to create an end-to-end data engineering pipeline. This pipeline used data from the on-premise SQL server database and loaded the data in Azure Data Lake Gen 2 storage. I performed the necessary transformations using Azure Databricks and loaded the data in Azure SQL Database. Using two levels of transformations, I ensured the data was clean and ready for different use cases. Thus, I extracted the data using Power BI and created a report for presentation to relevant stakeholders. This report represents a practical use case of Azure in data engineering in gathering data and making it ready and reliable for use.