DATA ENGINEERING PROJECT

AZURE MOVIE RECOMMENDATION PIPELINE

By Satish Gummadi

Email: satishgummadi4@gmail.com

Contents

1. INTRODUCTION:

In the present times, the Data is growing at an exponential rate. Data produced by the world each year is almost more than what it had produced till previous year. With increasing Data and its value, the companies are using methods and technologies to get the best of the data and add value to their business either by improving efficiency, reducing time, optimizing business models, etc. The use of Data to make business driven decisions didn't start now, it has started in earlier 2000's. But technology available then was not as fast as the technology which is available now. The Data which we used to store inbuilt servers are now being replaced by Giant Cloud service providers like Azure, AWS and GCP. We can now store and handle PetaBytes of data faster than GigaBytes of Data earlier with increased processing speed and storage space. Hence it is important for us to understand our business requirement, learn and implement best fit technology or technologies with changing time.

Along with improving technologies in processing speed and storage handling, there is also advancement in technology and services for automating these above tasks with least human intervention for different use cases across different sectors. Earlier we had to code using a specific programming language, connect it to a Database to access the data and use external or open source frameworks to connect them for performing ingestion, transformation, serving and storing the data (creating data pipeline) and manage them, which has comparatively lower availability and reliability under system downtime or if some other issues arise. But now the Cloud services have grown to such a extent that it can provide all the above services on a common platform with much easier way to integrate different services within or outside the given cloud service. These Cloud Service providers are also not only providing easier ways to scale the opted services as per changing loads or incoming data with time, but also providing pay-as-you-go subscriptions for individual services, which has improved the cost optimization for the business and rely more on these services. Also offering very high reliability, security and availability (with geo-redundancy) with least amount of down-time, all the small to large scale business are moving towards the cloud services rather than building everything from scratch in-house which will lead to a large capital investment for building and managing these services. Hence it is crucial for Software, IT, Data Science professionals, etc., to learn

these tools and services offered by various cloud service providers, and choosing best fit service based on our problem statement and how to optimize the resource or pipeline we created on these services to bring most value for a business with least amount of cost.

In this report we are going to understand and solve a real word problem movie recommendation system, where we can recommend a movie to a certain user based on his previous ratings and comparing his ratings to the ratings of other users, and providing the best possible recommendation to the user. With rise in online content and content creators, there is a rise in online platforms, one of them being streaming services like Netflix, Amazon Prime, Disney+Hotstar, Youtube, etc. These services use recommendation system to analyze the history of the user and recommend the similar content on user's homepage. The pipeline itself originates from collecting the data from the platform, cleaning, storing, using algorithms to provide the similar movies which the user may like the most, sending the data back to the platform and displaying it to the user. In our project we will build a similar pipeline where we will load the data in storage, build a pipeline to get the data from storage, check the schema format and perform ML algorithm, return the result to a service where we can send the data elsewhere. There are many cloud services providers emerged with time. Among them AWS (Amazon Web Services), Microsoft Azure and GCP (Google Cloud Platform Services) hold most of the market shares throughout the world. We will specifically use Microsoft Azure for our project and understand few services among many and how can we use them to solve our problem.

2. PROBLEM STATEMENT:

We are provided with two .csv files, one is movies.csv and the other one is ratings.csv file. The movies.csv file contains details such as **movieid**, **title** and **genre** of the movies. Whereas the ratings.csv file contains the details such as **userid**, **movieid**, **rating** and **timestamp** (the link of the files and other sources can be referred from the end of the report). Our task is to use **Microsoft Azure services** and create resources to **store data**, **analyze** the data and **provide recommendation** for given user, and **create pipeline to automate** the entire task. At the end, for a specific userid passed to the pipeline we should get the movie recommendation on our mail.

3. PROCEDURE (SOLUTION):

We are going to use **Microsoft Azure Cloud Services** as mentioned earlier. But even using the same service we can solve the same problem in multiple ways. The **Azure** provides more than 200 products and services which can be used to solve simple to complex business problems. In our case we will be using few services such as **Blob Storage**, **Azure Databricks**, **Azure Data Factory**, **Logic Apps**, etc., which we will try to understand in detailed. To simplify our problem we will break it down into 5 steps:

1) Loading and Storing the files using Blob Storage
2) Coding the Algorithm on Azure Databricks
3) Creating the pipeline on Azure Data Factory
4) Using Logic Apps to send the result to our mail
5) Running our entire pipeline

As we will be using multiple services, we also need to connect these services with each other by providing access, which we will understand under the above mentioned steps for each service. The Project flowchart will look like **Figure 1** below.

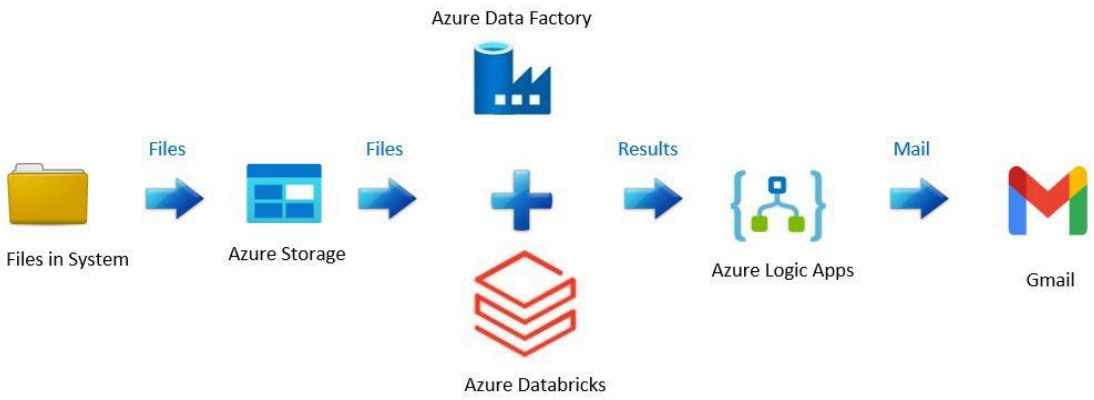

Figure 1: Project Flow Chart

To start with, we need to have an **Azure account**, which we can create one from the site https://azure.microsoft.com/en-in/free/ and start using it. We need an Azure subscription (which can be **Free Trial** or **Pay-as-you-go** or **Enterprise Agreement**) to use the services for the project. For new users Azure provides free trial for 1 month, which we can use if we do not have a subscription.

Once we logon to our Azure portal https://portal.azure.com/#home we will get a user interface similar to the **Figure 2** below. We can use global search bar to access different services in the portal.

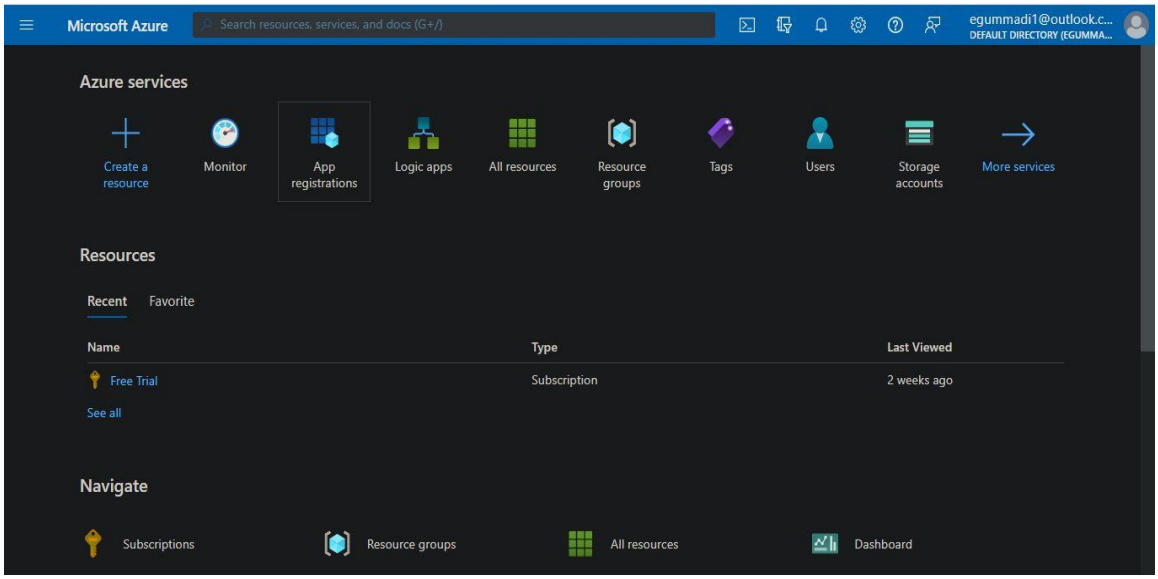

Figure 2: Azure Portal Home page

Before getting into project we need to understand few terminologies such as **Resources**, **Resource groups**, etc. As per Azure's definition, Resource is a manageable item that is available through Azure. **Virtual machines**, **storage accounts**, **web apps**, **databases**, and **virtual networks** are examples of **resources**. **Resource groups**, **subscriptions**, **management groups**, and tags are also examples of **resources**.

Resource Groups is a container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. For this project we will create a resource group with name let's say **movie_rg**. Now we will use this resource group for every resource we create moving forward.

For keeping the report more readable, I will divide the report into Sections and Appendix. Section 3: Procedure (Solution) will focus on the explanation and procedure to solve the problem while Appendix will have additional codes and resources.

3.1 Loading and Storing Files:

For Loading and storing the files we will be using **Azure Blob Storage service**. **Azure Blob Storage** helps us to create data lakes for our analytics needs, and provides storage to build powerful cloud-native and mobile apps. It also helps us to optimize costs with tiered storage for our long-term data, and flexibly scale up for high-performance computing and machine learning workloads.

We will create a **Blob Storage** resource from **Storage Account resource** (which we can find from global search bar). We will provide all the inputs to create a **Blob Storage**. One thing to note while creating **Blob storage** is to enable **hierarchical namespace**. This will allow the collection of objects/ files within an account to be organized into a hierarchy of directories and nested subdirectories in the same way that the file system on our computer is organized. Since we will be using multiple folders to validate our incoming files, we will be enabling this feature.

Once created, we will open our **Blob storage** resource and go to **Containers** section. This is similar to our file manager on our PC. We can create multiple folders as containers, containers within containers and store our data in them. Now this is how we will manage our storage: for our project we will create four folders named **rawdata**, **validateddata**, **rejecteddata** and **referenceschema** folders as shown in the **Figure 3**.

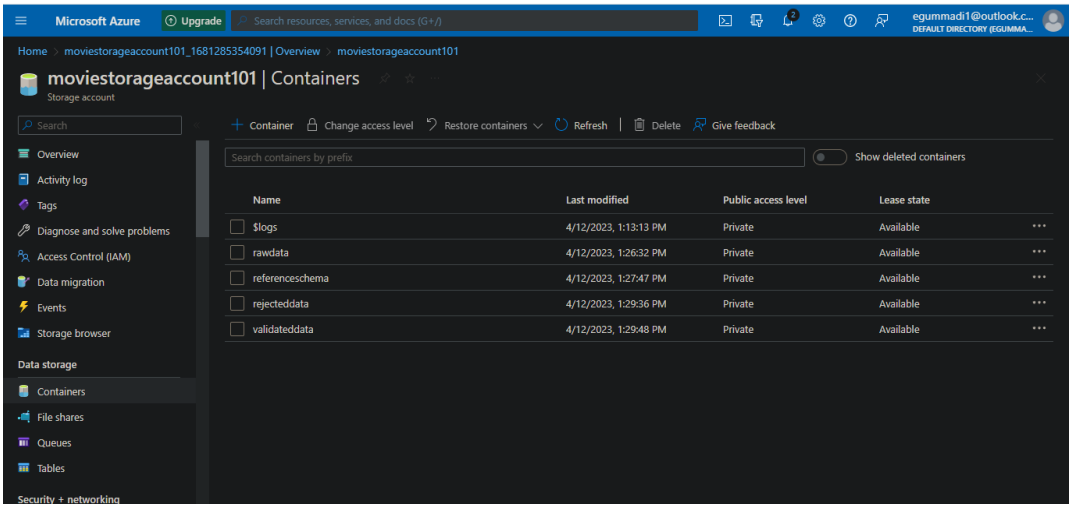

Figure 3: Azure Portal Home page

What we will do is we will use **rawdata** folder to upload and store our files. Once we create our pipeline in later section, we will check the schema of files in **rawdata** with our pre-defined schema in **referenceschema**. If they match, we will copy the files from **rawdata** folder to **validateddata** folder, which will further be used in the downstream. Else we will copy the files in **rejecteddata** folder.

With this our storage resource is ready. We will now go and add our **movies.csv** and ratings.csv file inside **rawdata** folder using Upload. We will upload the data from our PC. Once we have uploaded the files from our system, we have to connect our storage to different services (with **Databricks Notebook**, with **linked service** in **Azure Data Factory**), which we will discuss after creating other resources.

3.2 Coding the Algorithm on Azure Databricks:

We now need a platform or framework where we can write some code, pass on the files and inputs to get recommendation for specific input as output. For this we will use **Azure Databricks**. **Databricks** is a unified set of tools for building, deploying, sharing, and maintaining enterprise-grade data solutions at scale. We can use **Databricks** to process, store, clean, share, analyze, model, and monetize their datasets with solutions from BI to machine learning. With **Azure Databricks** we can set up our **Apache Spark environment** in minutes, autoscale, allow data processing with **Apache Spark cluster**, which helps in parallel processing of larger dataset and collaborate on shared projects in an interactive workspace. **Azure Databricks** supports Python, Scala, R and SQL, as well as data science frameworks and libraries including TensorFlow, PyTorch, and scikit-learn. **Azure Databricks** enables an open data lakehouse in Azure. With a lakehouse built on top of an open data lake, quickly light up a variety of analytical workloads while allowing for common governance across your entire data estate.

We have created **Blob storage** resource earlier. Now we will be using **Databricks** to run our code and obtain the recommendation as end result. For this we will create **Databricks** resource on **Azure**. Once we have created the resource, we open the **overview** page of our **Databricks** resource and click on **Launch**

to launch the **Azure Databricks**. This will open a new tab with different URL from our current Azure portal. The interface of **ADB** (**Azure Databricks**) can be seen in **Figure 4**.

After we create the resource and launch it, we have to create notebook in **ADB**. **Notebooks** are the primary tools for data science and machine learning workflows. In our project we will be using two notebooks:

1) For mounting the ADB to our storage and
2) To write our mounting code and movie prediction code in notebook and running it for output.

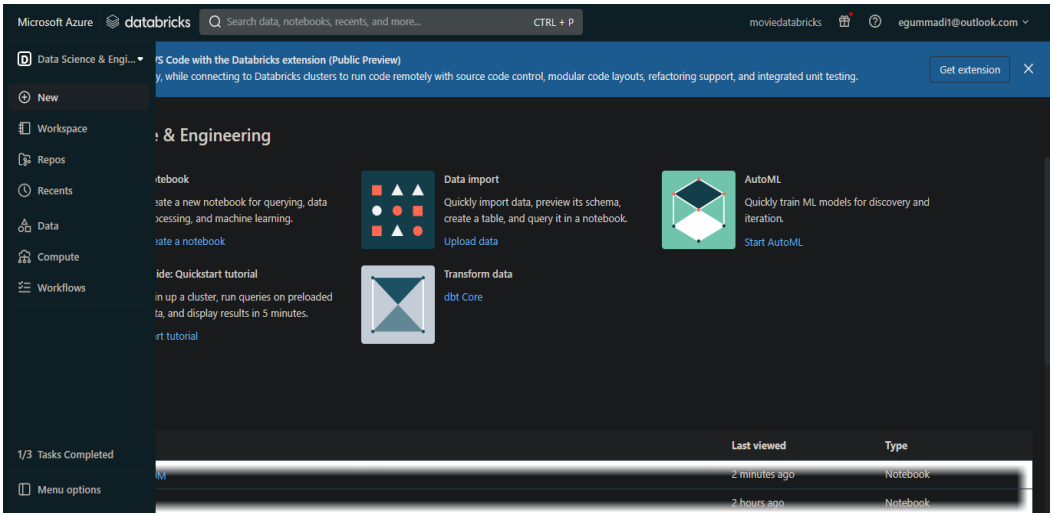

Figure 4: Azure Databricks

3.2.1 Mounting ADB to Blob Storage:

Azure Blob Storage and **Azure Databricks**, though offered by the same service provider on same portal, are two completely different resources. In **Azure**, in order to access a resource externally we need to provide access to the resource or service through which we are accessing. Here in our case we have our CSV files in **Blob storage** resource we have created earlier. To access these files from **Blob storage**, we first need to mount the storage with **ADB**. We can connect **ADB** to our **Storage** in multiple ways. We can either connect using the **ADLs Gen2** storage account **access key** directly (which is not recommended) or we can use services such as **Azure App registration** (Azure

Active Directory-AAD) and **Key-vault** to make connections more securely. We will go with using the second method.

We will be using **Azure App registration service** and **Azure Key-vault** service to connect our **ADLs Gen 2 account** to **ADB**. **Azure App registration** with Azure Active Directory enables a user with Power Apps user account to connect to their Microsoft Dataverse environment from external client applications using OAuth authentication. Registering an app provides us with Application ID and Redirect URI values that ISVs (Independent Software Vendors) can use in their client application's authentication code. On the other hand **Azure Key-vault** is used to securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets. It also makes it easy to create and control the encryption keys used to encrypt your data.

We will follow the following steps in order to connect these services with one another and to make a secure connection:

I. From **Global search bar**, go to **App registration service** and register a **new app** named **MovieApp** (**Supported account types is set to Single tenant type**). Then will go to our **Blob storage** and provide **Blob Storage Contributor access** to **MovieApp** in our **Blob storage IAM roles** section.

II. Create a **new Client Secret** in **MovieApp**. We will get a Value for our **client secret**. Copy the **client secret** value as it will disappear after the page is refreshed.

III. To save our **client secret** we will use **Key vault** service by **Azure**. Go to **Azure Key vault** resource and create a **new key vault**. Once we have created a **new vault**, we will go to **Secrets** section and **create a secret**. We will create three secrets: one for **client secret** which we copied in step II, and name it as **clientsecret**. We will create two more secrets named **clientid** and **tenantid**, for which we will add values from **MovieApp overview** page.

IV. Once we have created the **MovieApp** and **Key vault** with secrets added to it, we need to create a **Scope** in **ADB**, to link our **key vault** to it. For this we will copy our **ADB** URL from browser, paste in new tab and change the last section

URL as /secrets/createScope. The new URL will look like https://adb-4135278976388805.5.azuredatabricks.net/?o=4135278976388805#secrets/createScope

V. We can refer a simple flow chart to understand the overall process in **Figure 5**. Once the scope is created the connection is established for mounting.

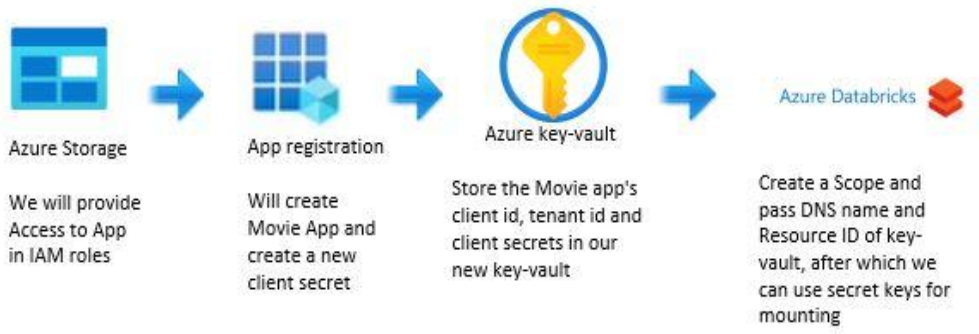

Figure 5: Flow chart for creating connection of Storage with ADB

3.2.2 Writing our mounting code, movie prediction code in notebook and running it for output:

Now to mount the storage we will use the mount code and assign keys of different secrets we saved in key vault under different sections. The Mounting code can be referred from the file **mounting.ipynb** file.

Before we can run the code, one important thing is to create the **cluster**. **Databricks** allows the user to create the code on **notebook** using various programming language, but to run them we need to create a cluster in order to allocate a compute engine for executing the code. As **Databricks** can setup **spark environment**, we can create **multi-node cluster** as per our project file size and requirement. As our data is not too large, a **single node** or **2 node clusters** will be sufficient. To create a cluster we will go to **Compute** section and click on **create cluster**. Here we will specify our cluster specifications and click on create. Once created and active, we need to go to our notebook and attach our **cluster** to our **notebook**, which we will see on top right of the notebook. Once we have attached and run the notebook, our storage will be successfully mounted to the location we have specified in our code.

We can check the files in our mount folder using following command:

```
ls dbfs:/mnt/Files/Validateddata
```

Now we will create a second **notebook** (which I have named it as **Movie_Lens_20M_final1**) for our movie prediction. The movie code will be attached with this report as .ipynb file. The whole code is written using **Python** language. Each section of code is written with explanation in the **Movie_Lens_20M_final1.ipynb** file, hence going through file from the top we can understand the code sequence. At the end we have used **Databricks widgets** to receive the **userid**(also specified a pre-defined value in case if no value is passed) and to pass the movie prediction as output. For this project, our main focus will be on understand the creation of pipeline for a given problem statement.

We are ready with our main coding section. Now we are good to start creating pipeline and automate it.

3.3 Creating Pipeline on Azure Data Factory:

We have most of our resources ready, now we need to create a pipeline, sequence them in right order for executing everything at once. For this we will use **Azure Data Factory** (ADF). **Azure Data Factory** is a managed cloud service that's built for complex hybrid extract-transform-load (ETL), extract-load-transform (ELT), and data integration projects.

To create ADF account we will go to **Data Factories** resource in **Azure portal** and create a new ADF resource under **movie_rg** resource group. Once created, go to **overview** section and **launch** the Data Factory. This will launch the Data Factory in a new tab, with a different URL from Azure portal, where we can start working on our pipeline.

To start building pipeline we will navigate to **Author** >> **Activities** >> **pipelines** >> **Create pipeline**. To have an overview of what we are about to create, this is how our end pipeline looks like in **Figure 6**.

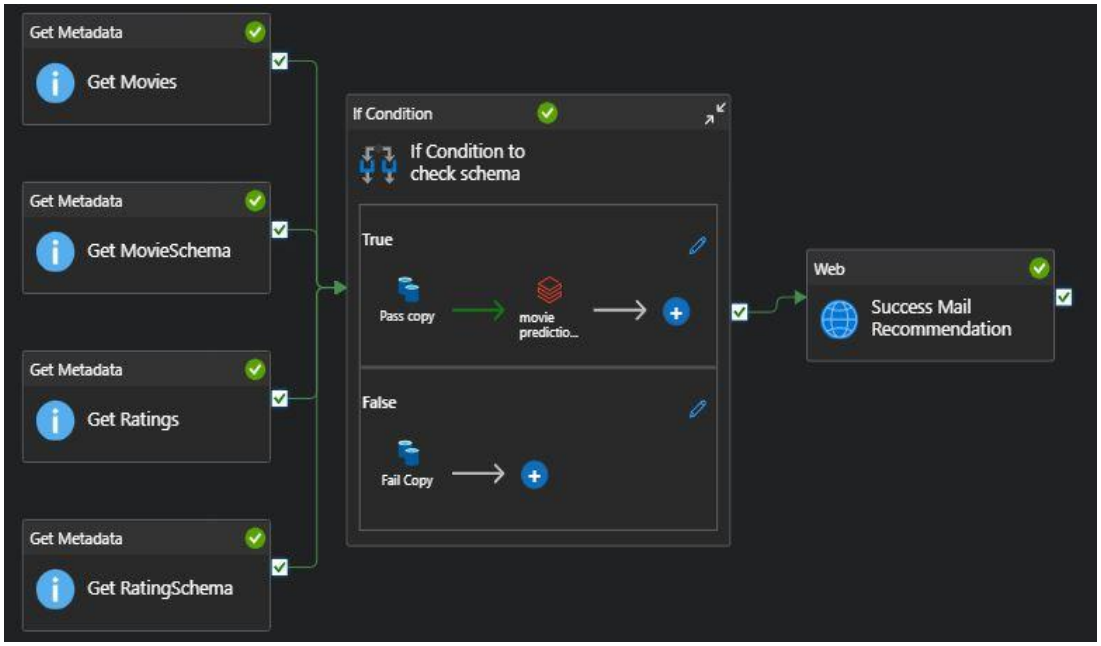

Figure 6: Pipeline Diagram

In ADF we have many **activities**, which are like functions to perform specific tasks. We have Activities **Get Metadata** to get information of specific files, **Copy Data** Activity to copy the file from one location to another, **Lookup** Activity can be used to read or look up a record/ table name/ value from any external source, etc. We will be using few of them as per our requirement to check the data, run the code and send the results. For this we will be using following Activities:

i. **Get MetaData**: Get Metadata activity can be used to retrieve metadata of any data in a Data Factory or Synapse pipeline.

ii. **If Condition**: The If Condition can be used to branch based on condition that evaluates to true or false

iii. **Copy Data**: Copy Activity in Data Factory copies data from a source data store to a sink data store

iv. **Notebook**: Notebook is used to process the notebook present in ADB.

v. **Web**: Web Activity can be used to call a custom REST endpoint from a pipeline.

Let’s divide the whole pipeline into three parts to understand each part of the pipeline.

Part 1: Getting the Meta Data from each file:

We have our storage where we will receive the CSV files of movies and ratings. But it is not necessary that every time we receive a right file with right schema structure. There are chances that we may receive a file with a missing column or we may receive a completely different file with same name. This will end up giving errors in the downstream of the pipeline. Hence creating a condition to check at the beginning of the pipeline helps to identify the issue in earlier stages and take necessary actions.

For this there is an activity called **MetaData** which we can use to extract the metadata of any file, which can include **itemName**, **itemType**, **size**, **lastModified**, **structure**, etc., which can be further used to analyze and make decisions based on file’s metadata. In our case we will use **structure MetaData type** in order to get the column names and column types of the file we are using. To compare the raw folder files, we will create a pre-defined schema (similar to the actual schema structure expected in the pipeline) by adding the names of the column in two new CSV files in the first row, one for both movies and ratings files respectively, and save those files in reference schema folder of our storage.

To use the **MetaData** activity, we will drag and drop this activity on the pipeline page. Once we click on **MetaData** activity we will find Settings like **General**, **Dataset** and **User Properties**. We will navigate to **Dataset** section and **create a new dataset**. First we will select a new Data store, which in our case is **ADLS Gen 2** (the place from where we are going to retrieve our file). Next it will ask to enter the **file format**, which we will specify as **DelimitedText**. Next we need to set the property and create a **linked service** where we will add our storage details and path details. Once done we will click on create to create the dataset. Here we will set this dataset for movie.csv. We will similarly create three more Get MetaData for ratings.csv, movieschema.csv and ratingschems.csv. If we **Publish all** and

Debug our MetaData, we can see the output something like the **Figure 7**.

Figure 7: Output of Get MetaData for movie.csv file

We will get a similar output for all the MetaData we created for different files. Our next task is to compare the **structure** of these files which we will cover in Part 2.

Part 2: Comparing MetaData Using If Condition Activity and Validating Data if True and Rejecting Data if False:

We have the metadata of all the files. Now next thing to do is to validate these files. We have two CSV files (movie.csv and ratings.csv) which we get from external source and another two files (movieschema.csv and ratingschema.csv) which we created for comparison. Now we need to compare movies.csv with movieschema.csv and ratings.csv with ratingschema.csv, and only when these two conditions satisfy we will proceed with using these files for our prediction. So for this we will use an Activity called **If Condition**. If Condition has a **pipeline expression builder** in its **Activities** section which allows user to write conditions, which when gets True will perform certain activity/ pipeline, and will execute some other activity/ pipeline if the condition gets False (we can see this in **Figure 6**, where If Condition is holding two pipelines, one for **True** condition and one for **False** condition).

To build the condition we will first drag and drop the **If Condition** activity onto our pipeline next to **Get MetaData** activity. We will connect these just by pulling the tick marks from each **Get MetaData** onto the **If Condition**. This allows the output of one activity to be used by the next activity. Now if we click on our Activity we need to set three things. One is the condition for comparing the metadata. The Second is the pipeline for True condition and the third is the pipeline for False condition.

First for setting the condition we will click on our **If Condition** activity and navigate to **Activities** section. Here we will find **Expression** tab. This is where we will write our condition. Click on **Add dynamic content**. This will open a new window where we can select **dynamic variables** used across pipelines, **functions**, etc., which we will use to build the condition. Our final condition will look something like this:

```
@and(equals(activity('Get
Movie').output.structure,activity('Get
Movieschema').output.structure),equals(activity('
Get Ratings').output.structure,activity('Get
Ratingschema').output.structure))
```

We use '@' to start our expression for having variables. There are multiple variables which we can use such as **Activity outputs**, **Parameters**, **System variables**, **Functions** and **variables**. In our case we will use Activity outputs of our MetaData whose syntax will look similar to: `@activity('Activity Name').output.structure`. To compare we will use **@equals()** function which will take two activities in it and compares them. We will also use another function **@and()** which will take two conditions and returns True only when both the conditions satisfy. The syntaxes of both the functions individually are as follows:

```
@equals('Activity 1','Activity 2')
@and('Condition 1','Condition 2')
```

When we combine all our activities with conditions, we will get the final expression as mentioned earlier. With this our condition part is done.

Secondly we need to add the pipeline if conditions gets **True**. For this we will navigate to same Activities section in **If Condition** Activity, and under **Expression** we will find **Case True** and **False** as shown in **Figure 8**.

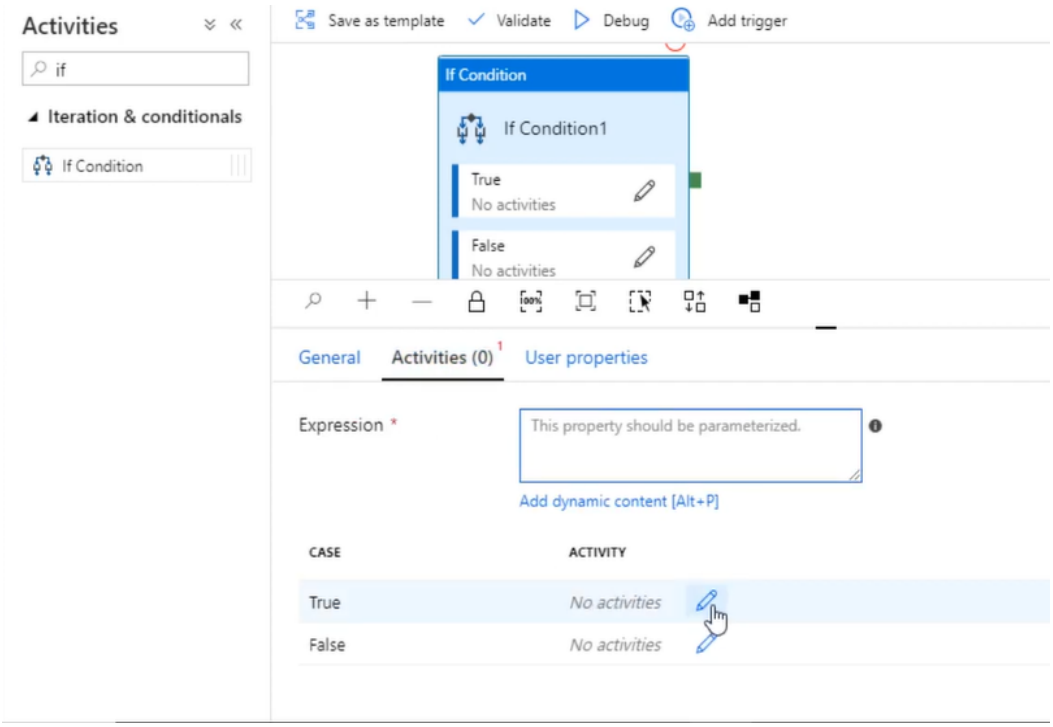

Figure 8: If Condition Activity

Now click on the Edit pointer near **No activities** to write new Activity/ pipeline under True condition. At this point of pipeline, our input files must be as per our set schema, so files are good for consumption. As discussed in Databricks section we will use these files for movie recommendation from **validateddata** folder in our storage resource. But the files we receive are first sent to **rawdata** folder. Hence we will use **Copy** activity to copy movies and ratings files into **validateddata** folder from **rawdata** folder, which we will then use **Notebook** activity to run our **Databricks notebook** (this can be seen in **Figure 5: Pipeline Diagram**). To use **Copy Data** Activity we will drag and drop the same inside the **True** case of our **If Condition**. Then we will click on it to assign the **Datasets** and **linked services** as done for **Get MetaData**. In case of **Copy Data** we have to mention **Source** and **Sink** details both to take the file from source and copy it to a destination. Once given the details we can **publish all** to save the changes. Now we will have our data in **validateddata** folder once we run

the pipeline. Next thing to add is our **Databricks Notebook**. For this we have a **Notebook** activity, which we will drag and drop next to our **copy data** activity and connect them. To edit the **Notebook**'s properties we will click on it and navigate to **Azure Databricks** >> **Databricks linked service** and click on **New**. Here we need to create a **linked service** to connect our **Notebook** in **ADB** with our **Notebook** activity in pipeline. We will enter all the details as usual, but to provide the access from the ADB we will go to **ADB** and go to **settings** >> **User settings** >> **Generate New Token**. This will give us a unique token which we will use in our **Notebook** activity to add it in our **linked service** section to access the **ADB notebook**. Once we add the **token key**, we can see our **ADB notebook** which we created at the start in our **Data Factory Notebook** activity. We will select it and click on **create**. To save the changes we will **publish all** the changes.

Thirdly we should copy all the files whose schema does not match with our reference schema, i.e., the files which fail the condition in **If Condition** activity. For this we have already created a **rejecteddata** folder in our storage which we will use to send the unmatched files and store it here. We will now navigate to **Case False**, click on edit and add a **Copy Data** activity in it. We will add **source** and **sink** details in this activity with file destination location changed to **rejecteddata** folder and **publish all** the changes. With this our Part 2 is complete. If we **debug** our pipeline with a pre-defined value of user, we will see the recommendation in our **Activity runs** section under the **Status** column, in front of our **Databricks Notebook** activity.

Part 3: Sending the Data to the End point

After **If Condition** Activity we will have our recommendation result with us in the form of JSON. Now we must use some Activity with which we can send the data to some app which is connected to some External Device. For this we will use **Web** Activity. As explained earlier, a **Web** Activity can pass datasets and linked services to be accessed and consumed by custom endpoint. To use it we will drag and drop the web activity next to **If Condition** and connect them. Now we will navigate to **Settings** of our **Web** activity (As shown in **Figure 9**).

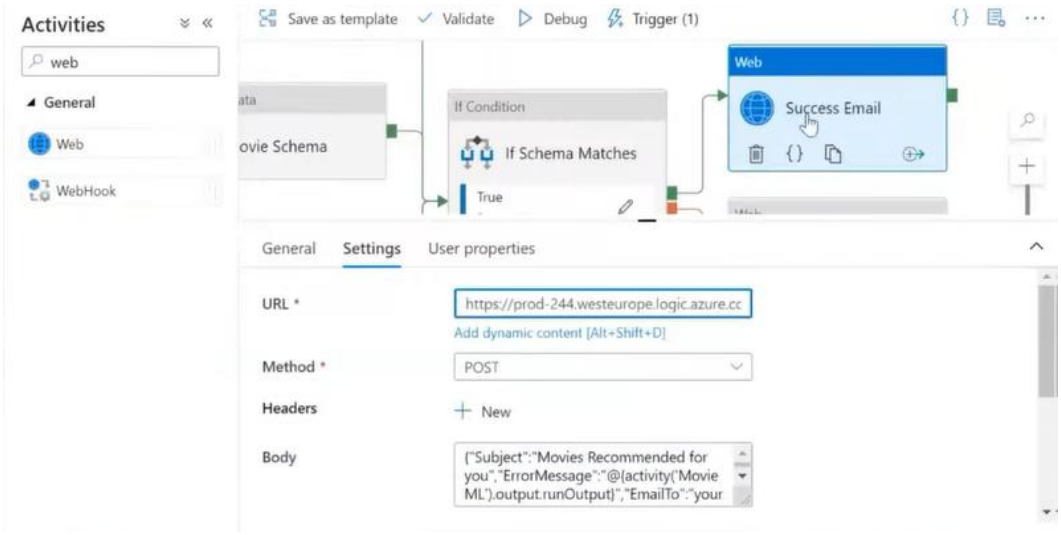

Figure 9: Web Activity properties

In **Settings** we have to edit few things. The first one is the **URL*** (The URL to which we will be sending our data, in this case it will be the URL generated by our **Logic Apps**, which we will cover in the next section). The second one is the **Method**, which we will set to **POST**. And third is the Body (in JSON format) in which we will add all the keys and dynamic values of our results from our **Notebook** Activity, which we need to send to our **Logic App**. The **Body** which we will be using is as follows:

Body:

```
{
      "DataFactoryName": @{pipeline().DataFactory},
      "EmailTo": "satishgummadi8@students.vnit.ac.in",
      "ResultOutput": @{activity('movie recom
      notebook').output.runOutput},
      "PipelineName": @{pipeline().Pipeline},
      "Subject": "Movie recommendation mail"
}
```

As we can see we have added sections such as Data Factory Name, Email, etc., we have also added our result JSON from Notebook Activity corresponding to “ResultOutput” key. Our final task is to add the URL which will be generated once we create a Logic App resource.

If required we can add another **Web** Activity to send the mail in case our file gets rejected or there is any issue in the upstream of the pipeline.

3.4 Using Logic Apps to send the results to our Mail:

Azure Logic Apps is a cloud-based Platform-as-a-Service (PaaS) that is used to automate tasks, workflows, etc. It helps in creating and designing automated workflows that are capable of integrating services, systems, and applications. In our case we will be using **Logic Apps** to create a workflow where we will receive our result from **Web** Activity of **ADF** pipeline and send this result to our **Gmail**. First go to **Logic Apps** service in **Azure portal** and create a **Logic Apps** resource. Once created go to **overview** section and click on **Edit**. This will navigate us to **Logic Apps Designer page**. Here on top end we will add **When a HTTP request is received** and in the bottom we will add **Send an email(V2)**.

We will edit first HTTPs function. We have to add the JSON schema here similar to the schema we added in our **Web** Activity in **ADF** pipeline. Our JSON schema will look as follows:

Request Body JSON Schema:

```
{
    "type": "object",
    "properties": {
        "DataFactoryName": {
            "type": "string"
        },
        "EmailTo": {
            "type": "string"
        },
        "ErrorMessage": {
            "type": "string"
        },
        "PipelineName": {
            "type": "string"
        },
        "Subject": {
            "type": "string"
        }
    }
}
```

Once we complete editing JSON schema we will get a **HTTP POST URL** as shown in the figure. This URL we will copy and paste in **URL** section of our **Web** Activity. This means Web activity will use this URL to send the data to this Logic App.

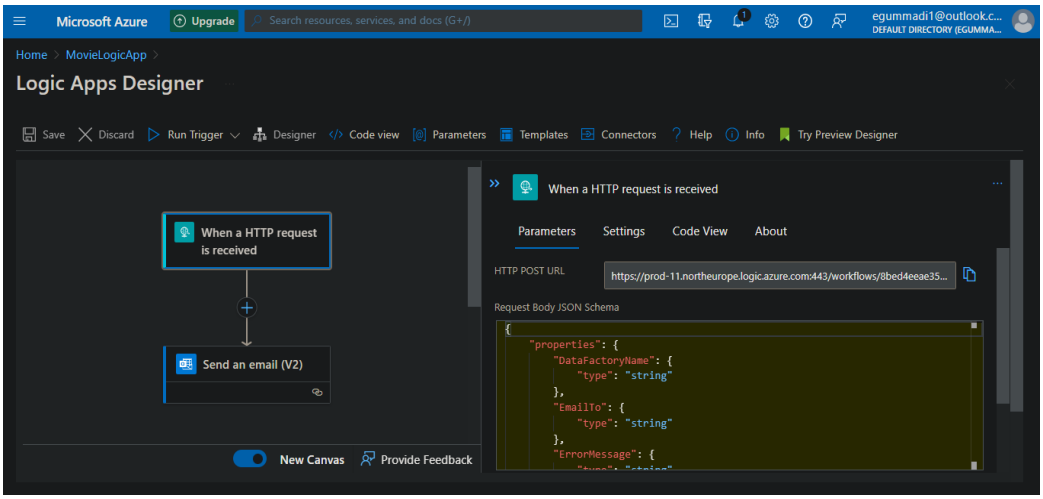

Figure 10: Azure Logic Apps, When a HTTP request is received properties

Now we need to edit the **Send an email(V2)** section. Click on it and we will find interface similar to our general email interface (refer **Figure 11**).

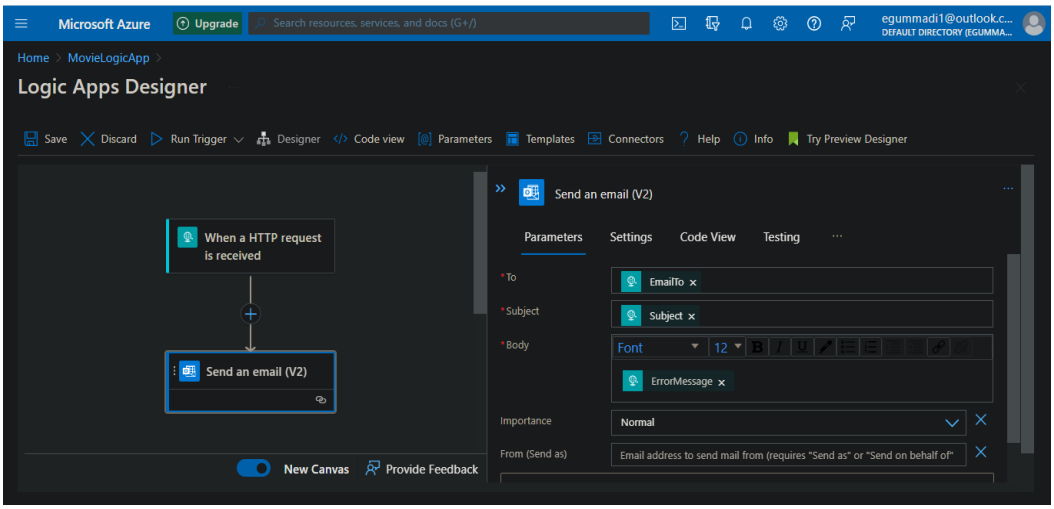

Figure 11: Azure Logic Apps, Send an email (V2) properties

Once we have published our pipeline code after updating the **URL** in **Web** Activity, we will be able to update our email with dynamic content which we receive from the **Web** activity. We will be using given keys in our result at specified locations like in **To**

section we will add our **EmailTo** key, and so on. Once we have updated these details we will save our design.

3.5 Running our entire pipeline:

Since we have created all the resources, pipeline and made all the connections, it's time to **Publish all** and **Trigger** them. We will go back to our pipeline and first **Publish all**. Once we have published all the changes we will now use **Add Trigger**. In **Add Trigger** we have different types of triggers as shown in the Figure below.

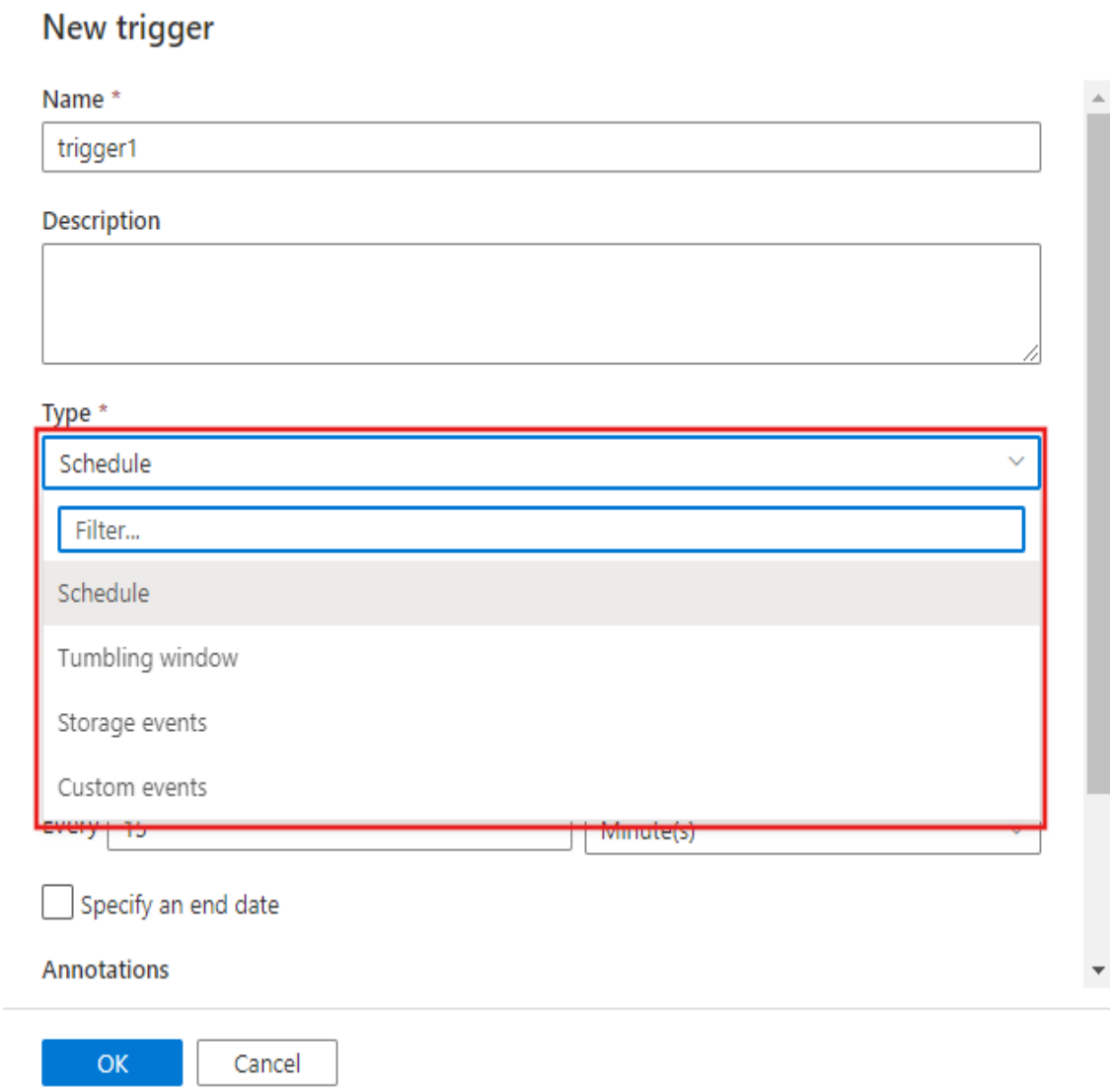

Figure 12: Add Trigger in Azure Data Factory

The four types of triggers are **Schedule Trigger**, **Tumbling window trigger**, **Storage event trigger** and **Custom event trigger**. We use these different triggers under different circumstances like **Schedule trigger** to invoke a pipeline on a wall-clock schedule. **Tumbling window trigger** operates on a periodic interval. **Event based trigger** responds to an event. We can also use **trigger now** to trigger the pipeline immediately without any condition. In our case we will use **Storage events** which will allow us to trigger our pipeline once a file is either copied or replaced into our file space. Once we create **storage events trigger** after specifying details like location, etc., this will immediately trigger our entire pipeline since we already have our movies and ratings file in our file storage. We can observe our pipeline under the **Output** section of our pipeline page as shown in the figure below.

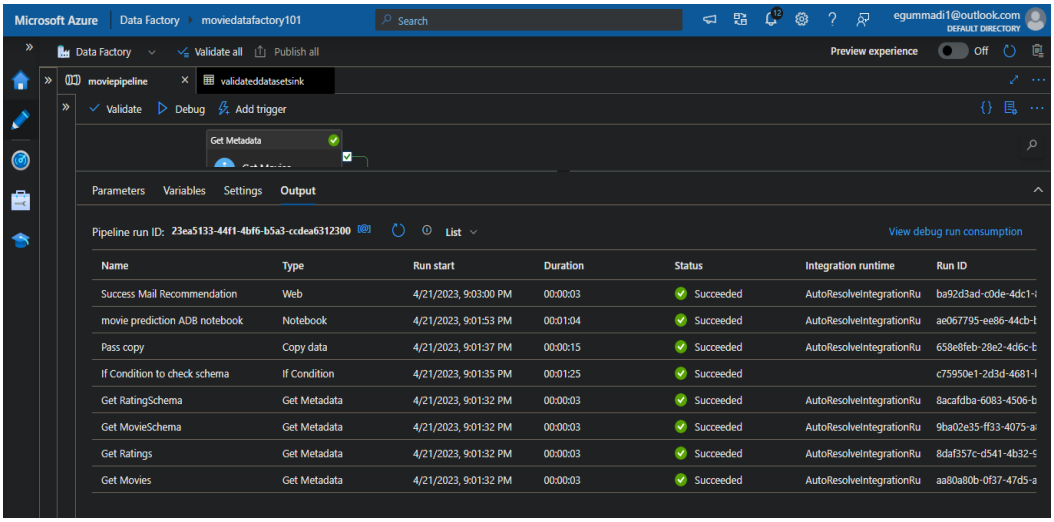

Figure 13: Output of Pipeline run

We can also monitor our pipeline in detailed in **Monitor** section of Data factory. If there is any error in pipeline we can see the issue in the **status** column, which will help us to rectify the issue immediately.

We can also check the **status** of pipeline in the form of **timeline** from **monitor** section which will look as shown in the figure below.

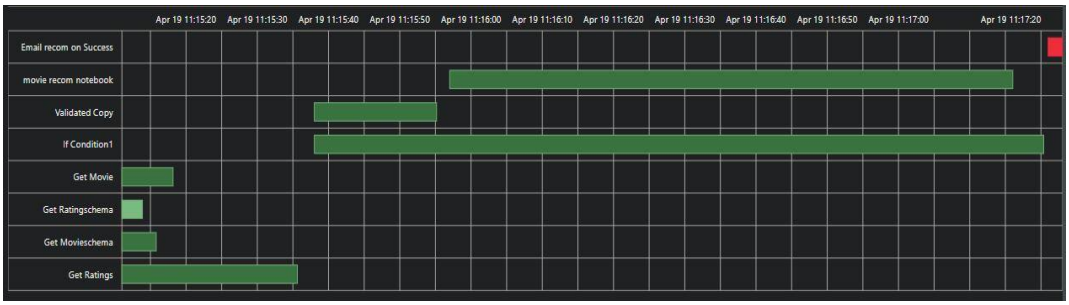

Figure 14: Timeline of pipeline

If we observe Figure 14, I initially made a small error in the **Web** Activity of the pipeline, which lead to the red indication in the timeline, hence we can detect such issues on time from **output** and **monitor** sections and resolve it.

After the successful pipeline run and email, we can see the green check marks in our **Data Factory pipeline** and **Logic Apps** resource which will give an indication whether our email was successfully sent or not, as shown in the figure below.

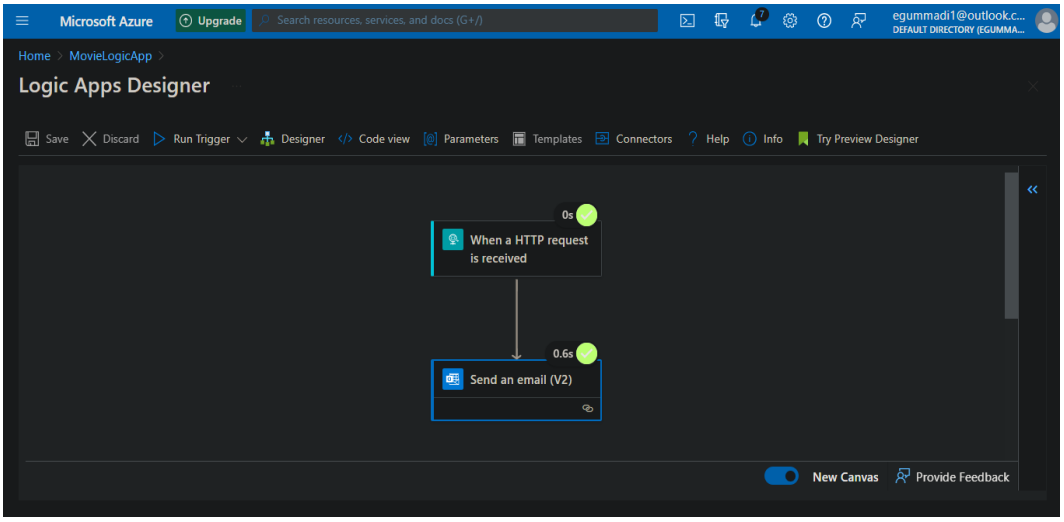

Figure 15: Logic Apps resource page after successful email

As we have encountered no errors and everything went well, we must have received an email of recommendation for a pre-defined user id passed in our **ADB**. We can see the email in the figure below.

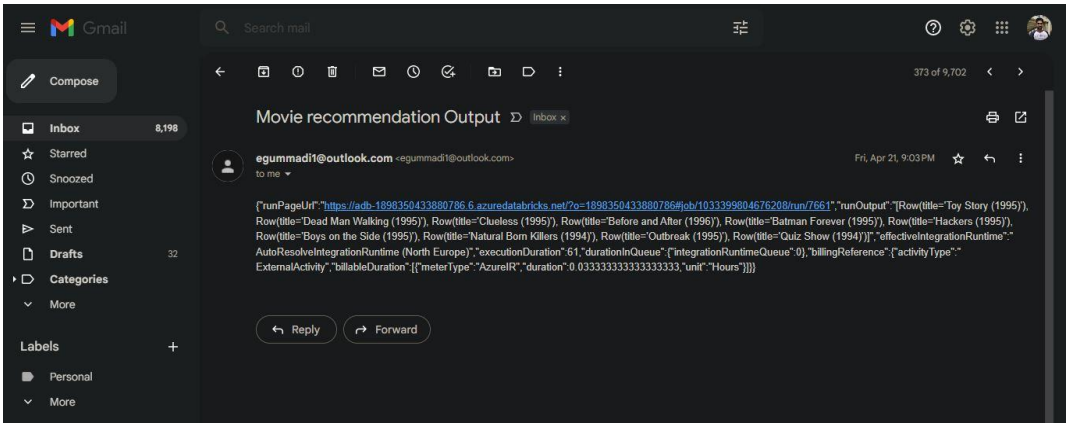

Figure 16: Our email recommendation on Gmail

Once the project is complete and purpose is served, we will delete the entire **resource group** in order to avoid additional costing.

4 CONCLUSION:

In this project we have created an End-to-End pipeline starting from loading the data from external source to sending the result to Gmail. We can summarize the entire project in following steps:

1) We created a **Microsoft Azure account** to use the services. We also created a **resource group** in order to group all the project files together.

2) We created a **Blob storage** with **hierarchical namespace** enabled and created different folders for different purpose of storing files.

3) We created **Azure Databricks** resource, **mounted** it to our **Storage** using **App registration** and **Key vaults**. We created a **cluster** in **ADB** and added our mounting code and movie recommendation code in different **notebooks**. We can run the codes by attaching the cluster to the notebooks and running them.

4) We then created **Azure Data Factory resource**, created our pipelines using different activities such as **Get MetaData**, **If Condition**, **Copy Data**, **Notebooks** and **Web Activity**. We also connected them, set their properties and published them.

5) We created a **Logic App** resource and designed a **workflow** to receive data from pipeline and send result to our email. After using **URL** to connect our pipeline with **Logic App**, we created a **Storage event trigger** and triggered the entire pipeline.

6) We checked the entire pipeline run on **Output** section and in **monitor**, and we checked the movie recommendation email we received on our **Gmail**.

With this we have completed the movie recommendation pipeline using Microsoft Azure services. Now we should remember that we can complete this project in multiple ways either using different services in Azure (like Synapse, etc.) or we can use different cloud provider like AWS or GCP. In any way our main goal should not only focused on completing the project, but also to use the right technology by finding the balance between the speed, cost and resources and optimizing it for worst case scenarios as much as possible, with least possible maintenance for pipeline.

APPENDIX

APPENDIX- 1:

Mounting Code

In **Azure Databricks (ADB)** we use mounting to mount the storage with our ADB notebook. The mounting details has been discussed in Section 3.2. The code for mounting the storage is as follows:

```
adlsAccountName = "movieprojectstorageacc"

adlsContainerName = "validateddata"

adlsFolderName="Data"

mountPoint="/mnt/Files/Validateddata"

# Application (Client) ID

applicationId=dbutils.secrets.get(scope="MovieSec",key="clientid")

# Application (Client) Secret Key

authenticationKey=dbutils.secrets.get(scope="MovieSec",key="clientsecre
   t")

# Directory (Tenant) ID

tenantId=dbutils.secrets.get(scope="MovieSec",key="tenantid")

endpoint="https://login.microsoftonline.com/"+tenantId+"/oauth2/token"

source="abfss://"+adlsContainerName+"@"+adlsAccountName+".dfs.core.wind
   ows.net/"+adlsFolderName

# Connecting using Service Principal secrets and OAuth

configs={"fs.azure.account.auth.type": "OAuth",

"fs.azure.account.oauth.provider.type":
   "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",

"fs.azure.account.oauth2.client.id": applicationId,

"fs.azure.account.oauth2.client.secret": authenticationKey,

"fs.azure.account.oauth2.client.endpoint": endpoint}

# Mounting ADLS Storage to DBFS

# Mount only if the directory is not already mounted

if not any(mount.mountPoint==mountPoint for mount in
   dbutils.fs.mounts()):

   dbutils.fs.mount(source=source,mount_point=mountPoint,extra_configs=
   configs)
```

APPENDIX- 2:

Movie Prediction Code

Since the movie prediction .ipynb file is big with multiple cells, I will be attaching the .ipynb file along with this report for reference.

REFERENCES:

This project has been referred from multiple sources to complete. Few of them are as follows:

A. The Major concept of the project is referred form a YouTube channel “Data Engineering for Everyone”. Link: https://www.youtube.com/@dataengineeringforeveryone
B. The use of various Azure services is referred from Microsoft Azure portal. Link: https://azure.microsoft.com/en-us/products/
C. We have used the Datasets from the website grouplens.org which provides many real world datasets to be used for various Analysis.
   Link: https://grouplens.org/datasets/movielens/25m/