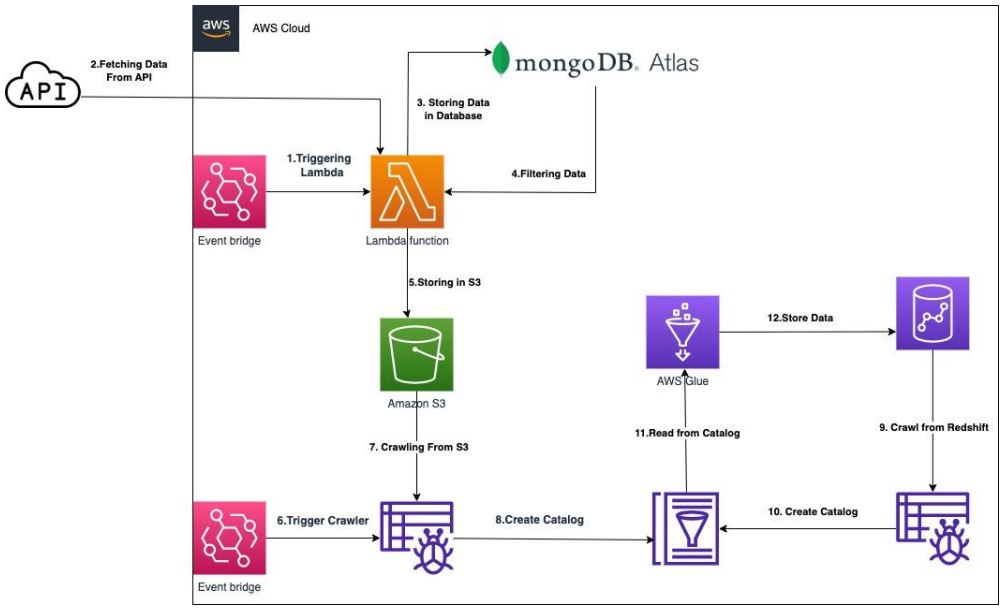
Creating ETL pipelines in AWS Cloud

Table of Contents

1 Introduction

The Aim of the project is to perform ETL Pipeline in AWS using different services of AWS.

1.1 Architecture



The above picture illustrates the architecture of the project. The numbering of each step explains the sequence of steps.

Explanation:

First, we will extract the data from the API ('https://house-stock-watcher-data.s3-us-west-2.amazonaws.com/data/all_transactions.json') and read the data in AWS lambda using the python Request Library.
The next step involves in storing the data in MongoDB atlas and querying the data using filters (i.e., using condition). The queried data is stored in AWS S3 bucket.
The data (i.e., which was stored in S3 bucket) is now crawled using AWS crawler and it creates a catalog (i.e., storing the schema of the file in metadata). Now we need to create a database in AWS redshift and create a table (the table should have equal columns and schema similar to the file in the AWS S3 bucket). We now need to use the crawler to crawl the schema of the table, which was created in AWS Redshift.

We need to create an AWS Glue job (the job reads the schemas from both catalogs) and store the data in the AWS Redshift table.
Note that the schemas of both (i.e., one from the S3 file and one from the AWS redshift table) must have equal columns and types to perform data mapping and to store data.

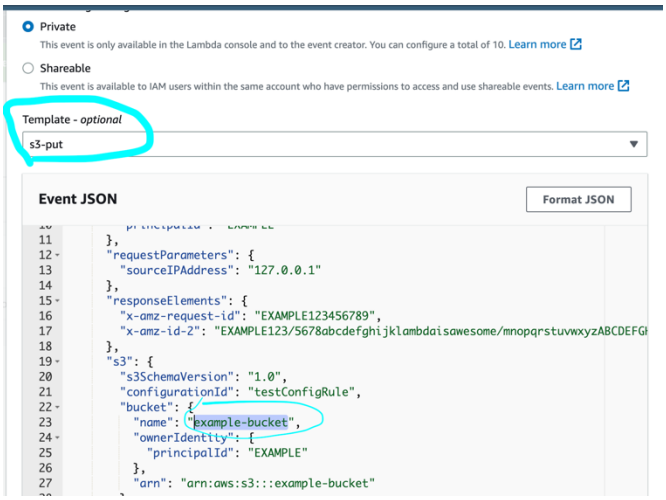
Finally, to automate create the workflow, we will use the AWS Event bridge service.

The project is divided into two parts for understanding the flow of the process.
Part -1
1.Event_bridge ➔ AWS Lambda Function-1 ➔ MongoDB Atlas ➔ dumping data in S3

Part-2
2.Event_bridge ➔ Workflow ➔ crawler ➔ AWS Glue_Job ➔ AWS Redshift

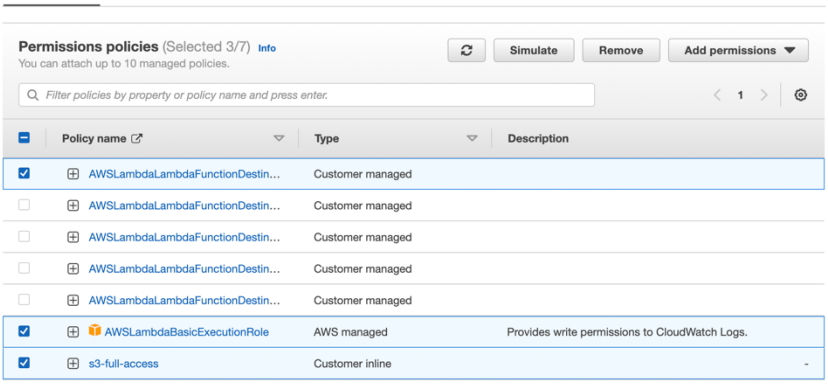
2 Implementation

The implementation starts with creating the **AWS lambda function** and locating the S3 bucket to store extracted data.
The below picture shows how one can map an S3 bucket to AWS lambda to store extracted data from the API



- Select the S3-put option in the sample template option
- Change the “example-bucket” name to the S3 bucket name (i.e., your S3 bucket name)
- Save the changes

Now we need to navigate to the **AWS IAM** service and we need to attach **S3 full access** policy and **AWSlambdaBasicExecutionRole** to the **AWS function role.**



The above changes will map the S3 bucket to AWS lambda and provide full access to S3 (i.e., one can read, write and perform other operations of S3 giving full access).

In order to Read and download the data from API, we need the following packages

- Requests (used to download the data)
- Pymongo[srv] (used to connect with MongoDB)
- Pandas (used to perform transformations on Data)

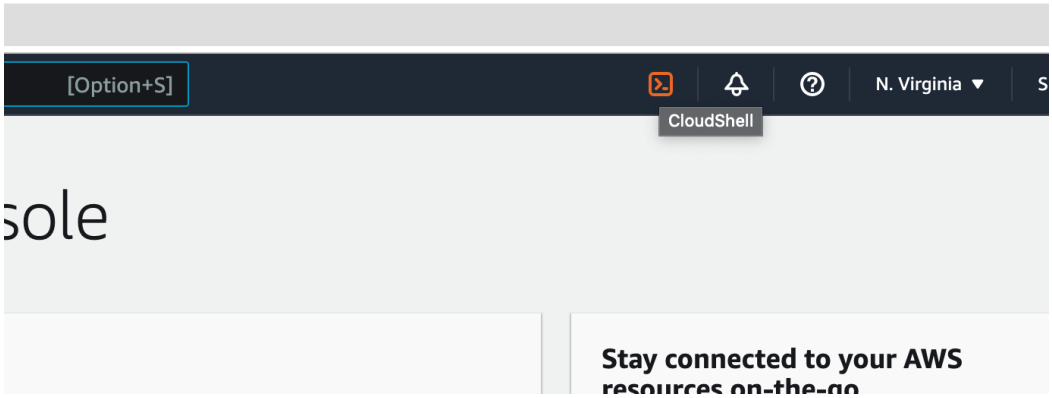
Now we need to manually upload the required packages to the AWS lambda since AWS lambda has only a few libraries.

There are many ways to upload the libraries to AWS lambda, but we believe the below instructions will be the most simplistic one.

2.1 Uploading the packages from AWS cloud shell to AWS Lambda

Navigate to s3 console and create an s3 bucket and give it a unique name (This is where you will store the zip package created)

Now open aws cloud shell. It can be found at the top right of the console. Looks like the image below



Once the terminal is loaded, create a directory named "packages", then change into the directory by running the commands below one after the other

```
mkdir packages
cd packages
```

Next, create a virtual environment named venv and activate the virtual environment by running the commands below one after the other.

```
python3 -m venv venv
source venv/bin/activate
```

Once the virtual environment is created. Create a folder named as “python” in the same directory (Make sure you create the python folder in the same directory, where the environment was activated)

```
mkdir python
cd python
```

Now we will download the required packages in the python folder using the PIP package manager.

- pip install requests -t .
- pip install pymongo[srv] -t .

once the packages are downloaded, we have to check the files by listing them (i.e., using “**ls**” command). This will show all the files related to the packages in the python folder.

In-order to save some space we will delete objects with "**.dis-info**" extension from the python folder (i.e., we don’t need these files to run our project). The best way to do this is by running the command below

No need to zip the python folder where they live but before we do that, let's save space and delete objects with "**.dis-info**" extension from the folder. They are not needed. The best way to do this is by running the command below

```
rm -rf *dist-info
```

After running the above, we should be left with only the relevant packages needed.

Now change the directory by using the command “**cd..**” and make sure we are in the packages folder.

Now, we will zip the python directory and give it a name called "my-first-lambda-package.zip". Run the command below to achieve this.

```
zip -r my-first-lambda-package.zip python
```

Once done, you should have a zip file named my-first-lambda-package.zip in the current directory.

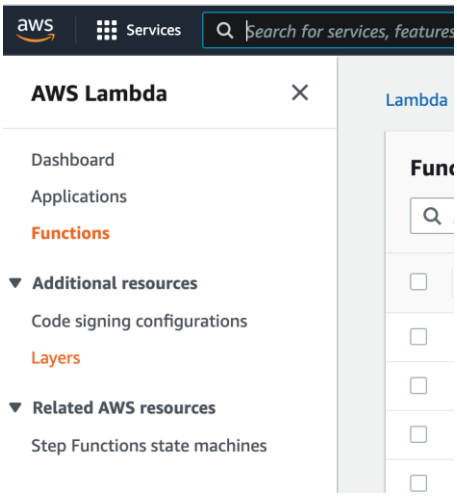
Next, upload the packaged zip into the s3 bucket you created earlier. My command looks like the below

```
aws s3 cp my-first-lambda-package.zip s3://your-s3-bucket-name/
```

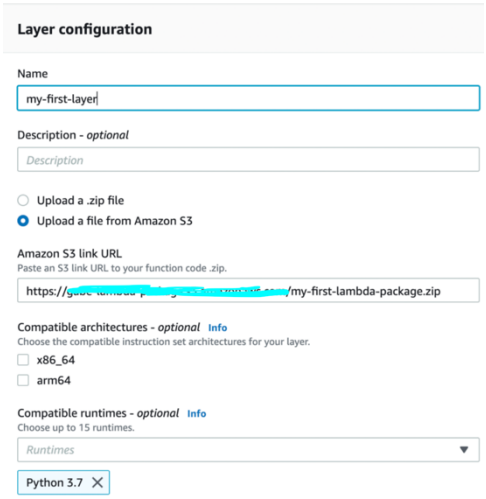
Now navigate to s3 bucket that we have gave in the above command and check if the file exists

2.1.1 Creating lambda layer and uploading packages

Navigate to the AWS Lambda panel and select layers

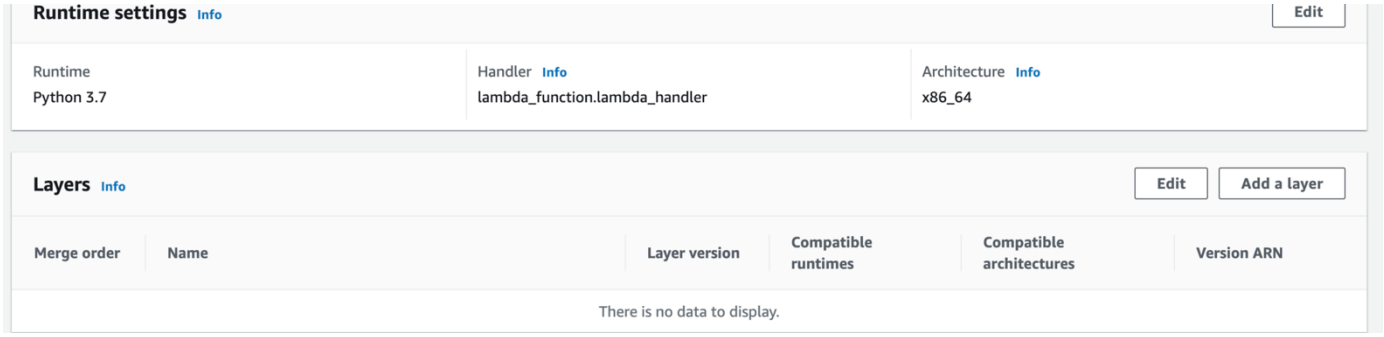


Click on "Create layer" and fill in the needed configuration. See the below image. We will select the "upload a file from Amazon S3" option and paste the URL of the S3 file (i.e., the zip file link). Select the python3.7 runtime and click create

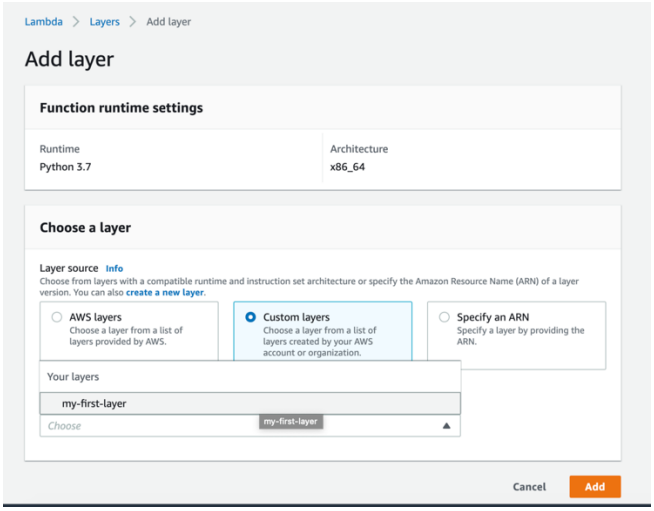


2.1.2 Attaching layer to lambda

Open the lambda function and scroll all the way down. You will see an option to add a layer on the downward right.



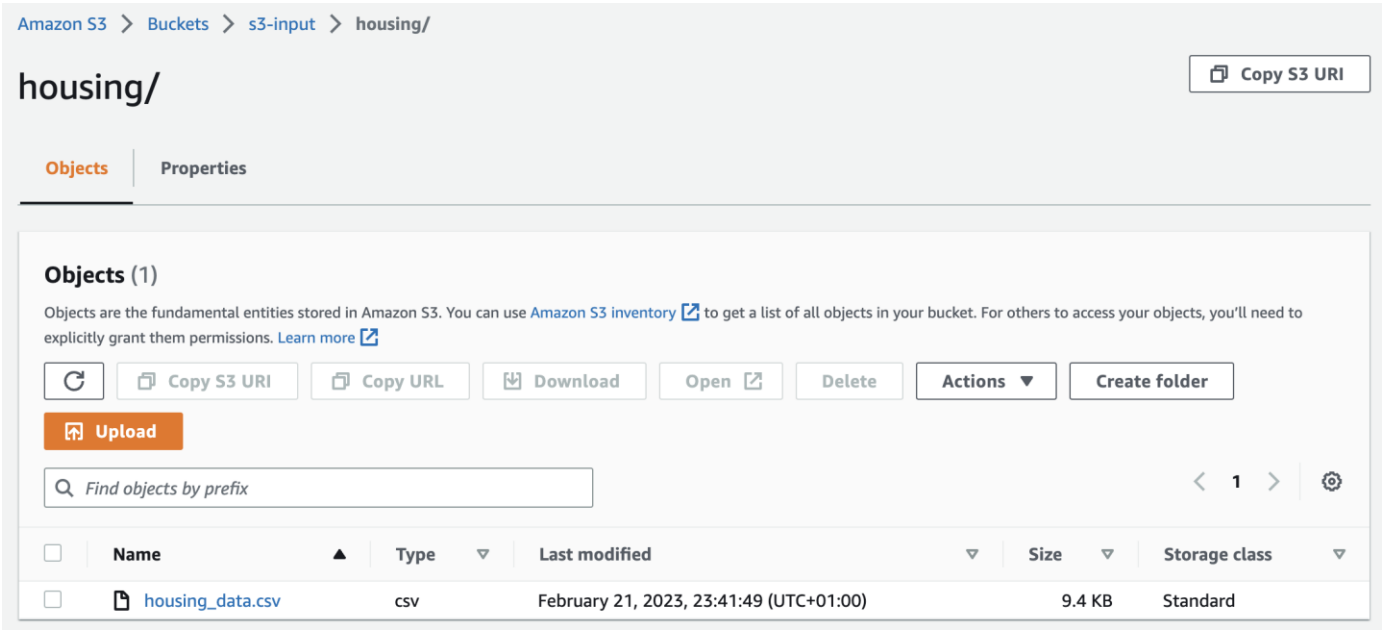
Click "Add a layer" and a page should appear for configuration. Select "Custom layers" and from the dropdown, select "my-first-layer" in previous section. If it asks for version, select version 1.



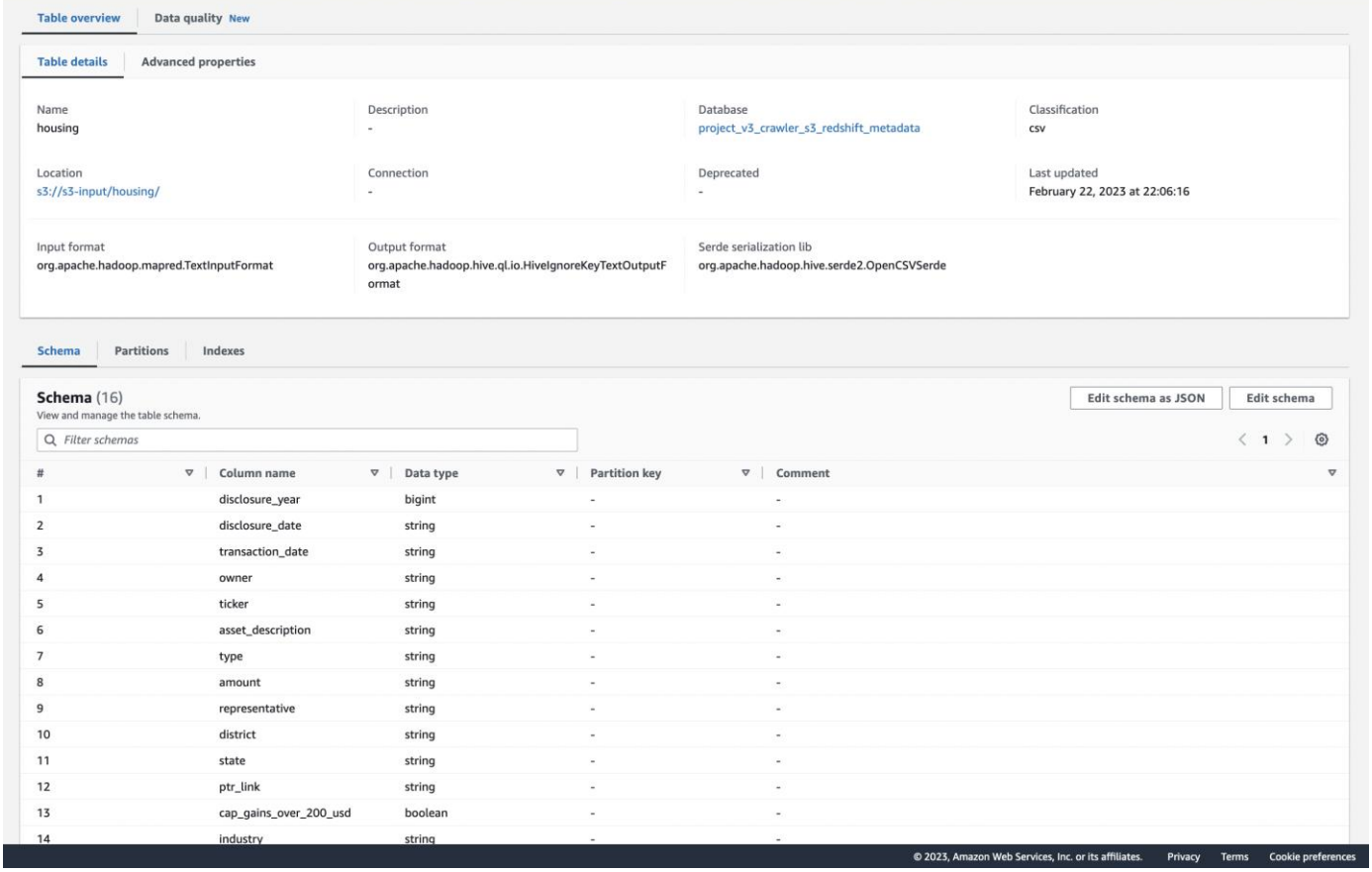
3 AWS lambda Function

```
1.  import json
2.  import requests
3.  import pymongo
4.  from pymongo import MongoClient
5.  import pandas as pd
6.  from pandas import DataFrame
7.  import boto3
8.  from io import StringIO
9.
10. response_API = requests.get('https://house-stock-watcher-data.s3-us-west-
    2.amazonaws.com/data/all_transactions.json')
11. cluster =
    MongoClient("mongodb+srv://<mongoID>:<password>@cluster0.juv3uy5.mongodb.net/?retryWrites=true&w=majorit
    y")
12.
13. def lambda_handler(event, context):
14.     # TODO implement
15.     try:
16.
17.         data = response_API.text
18.         print(" check if reponse code is equal to 200, then you have good conenction: \nreponse code =
    ",response_API.status_code)
19.         parse_json = json.loads(data) #loading API data to Json object
20.         housing_DB = cluster.housing_db #creating database in mango_atlas
21.         house_Collection = housing_DB.house_table #creating collection in mango_atlas
22.         house_Collection.insert_many(parse_json) #inserting records in mangodb
23.         print("API records inserted in MongoDB_atlas")
24.         query = {"transaction_date": {"$gt": "2021-12-01" , "$lt":"2021-12-05"}} #filtering dates based
    on schdule
25.         collections_without_mango_ID = house_Collection.find(query,{'_id':0}).sort("transaction_date")
    #sorting records based on transaction_date
26.         list_cur = list(collections_without_mango_ID)
27.         df = DataFrame(list_cur) #converting cursur to dataframe
28.         print("printing dataframe")
29.         print(df.head(3))
30.
31.         bucket = 's3-input' # bucket name that we want to store in S3
32.         csv_buffer = StringIO()
33.         df.to_csv(csv_buffer,index = False, header=True)
34.         s3_resource = boto3.resource('s3')
35.         s3_resource.Object(bucket, 'housing_data.csv').put(Body=csv_buffer.getvalue())
36.         print("Data Write Successfull in S3")
37.
38.
39.     except Exception as err:
40.         print(err)
41.
```

the above code extracts the data from API with the python request package (look at like 10) and dump the data in mango DB atlas and finally stores data in S3. The below image shows the file in the S3 bucket.

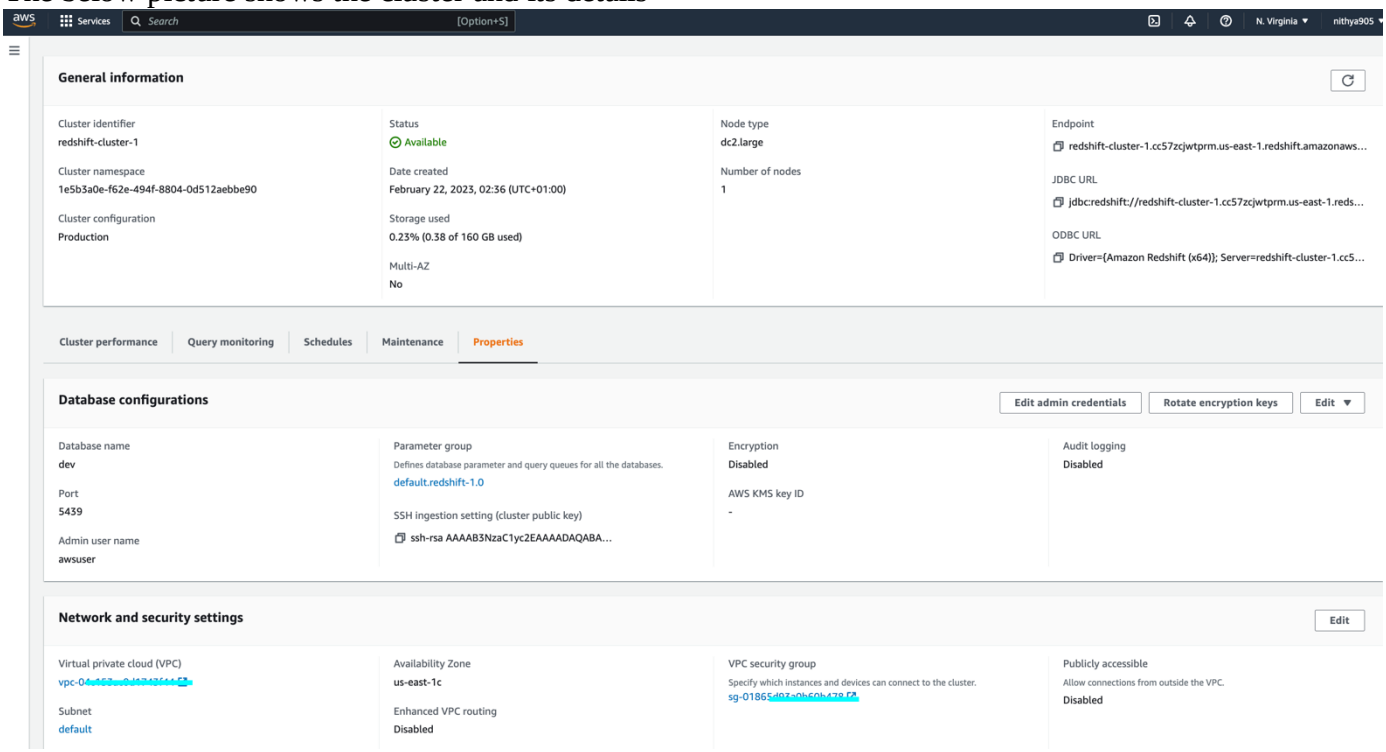


We will crawl the data using a crawler, from S3, and store the metadata in the database (catalog). The below picture shows the schema stored in the catalog



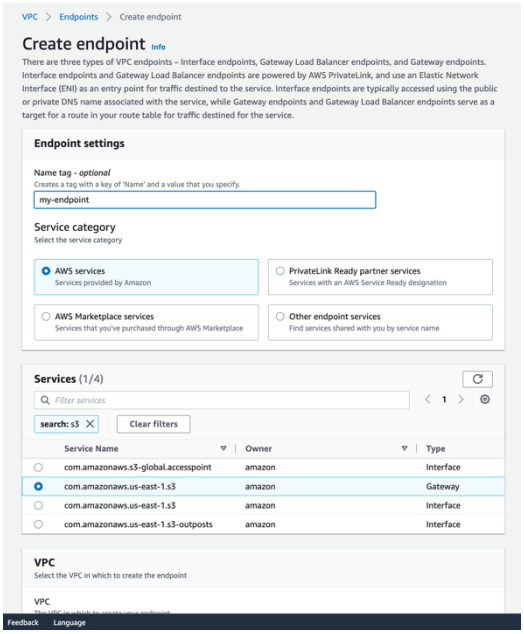
4 Creating Database and table in AWS Redshift cluster

Before creating the database and table in redshift, we must create the cluster in AWS redshift.
The below picture shows the cluster and its details

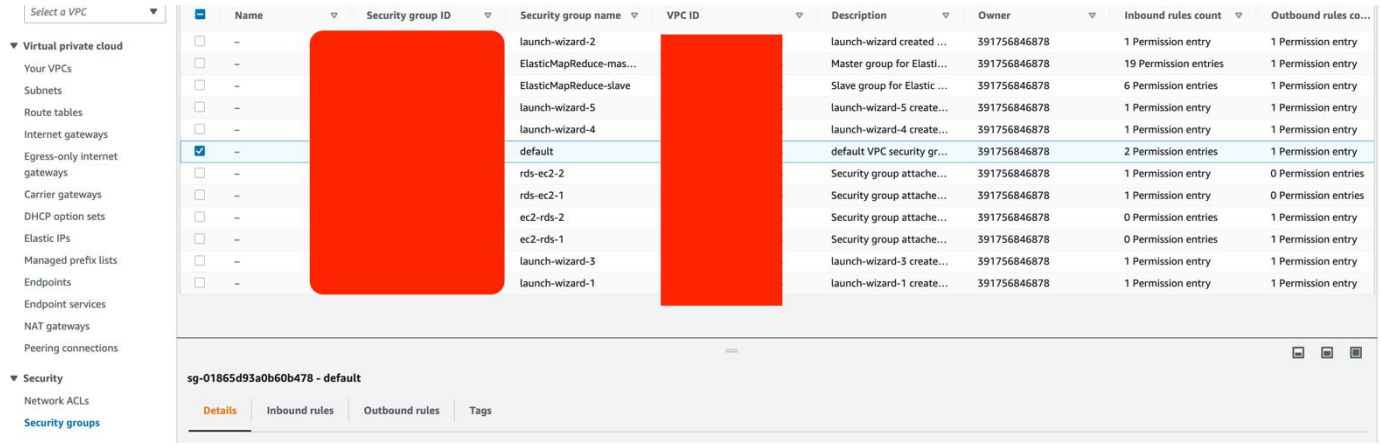


Notice the VPC security group in the network and security settings. We must make some changes in AWS VPC manager service.

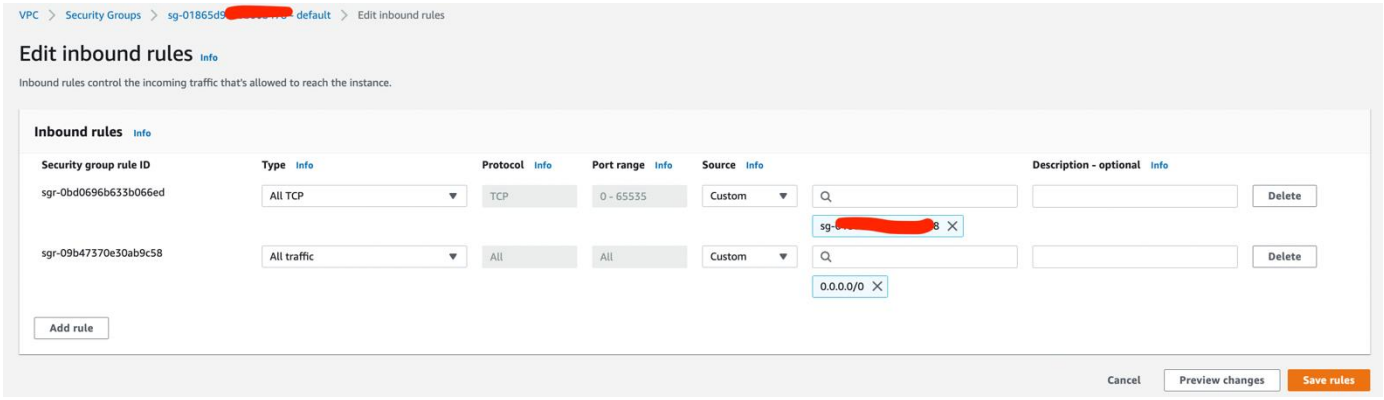
- Navigate to the AWS VPC manager and create an end point using S3 (make sure the type is the gateway)



- Now navigate to the security group and add in bound (make sure you select the default one)

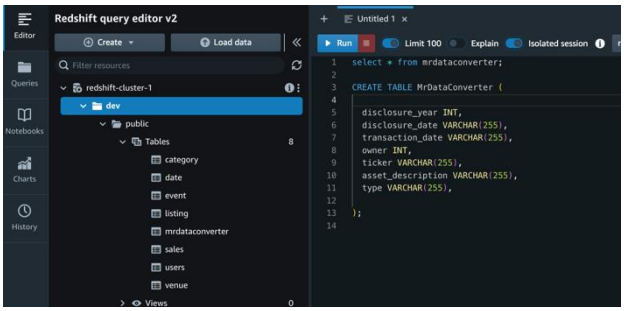


- Click on the edit inbound rule and make sure the type should be ALL TCP and save the changes.



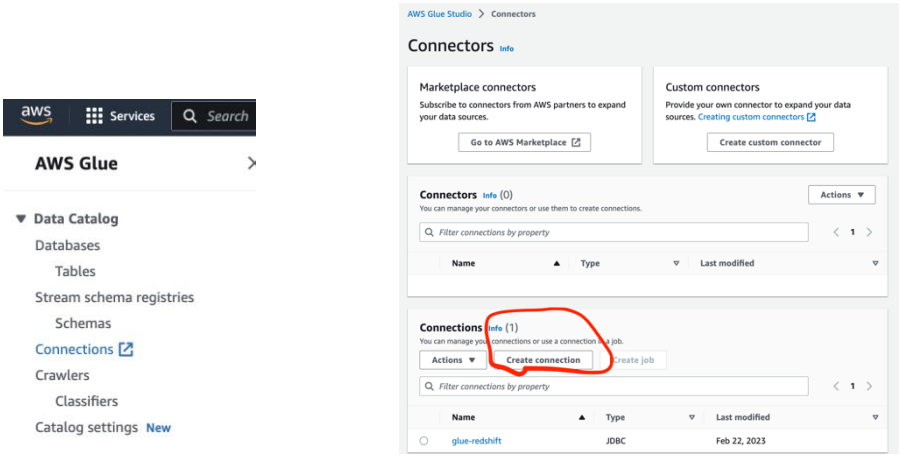
Note: make sure that default VPC and the AWS redshift cluster VPC are the same. This will allow us to perform smooth ETL operations.

- Create a database and table in AWS redshift. Note that the schema in the file (which was stored in s3) and in redshift are the same

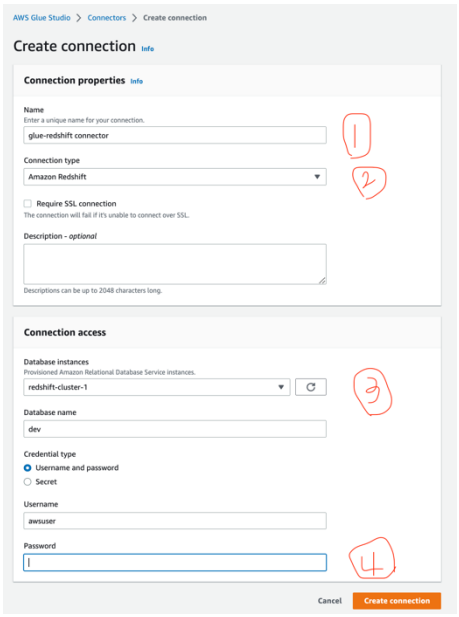


5 Creating the AWS Glue- Redshift connector

Navigate to the connectors in the AWS Glue catalog section and click on connectors and click on create connection.

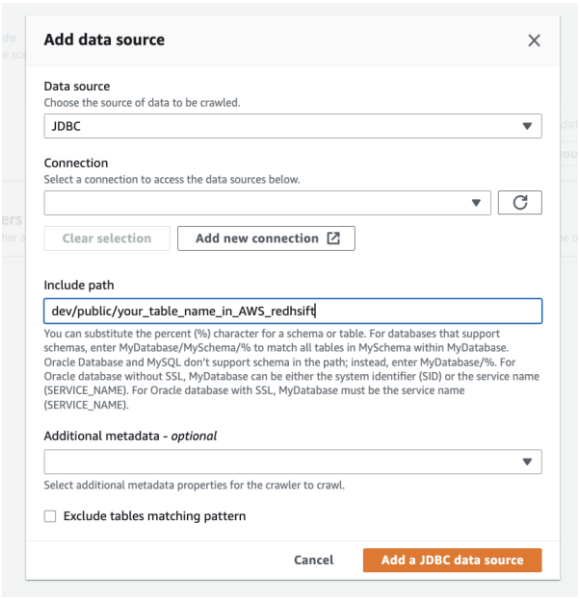


Give the connector a name, select the Aws redshift from the drop-down of connection type, select cluster your AWS redshift cluster from the dropdown of the database instance, and finally enter the password you gave while creating the AWS redshift cluster. The below image has the numbering of changes that one can make



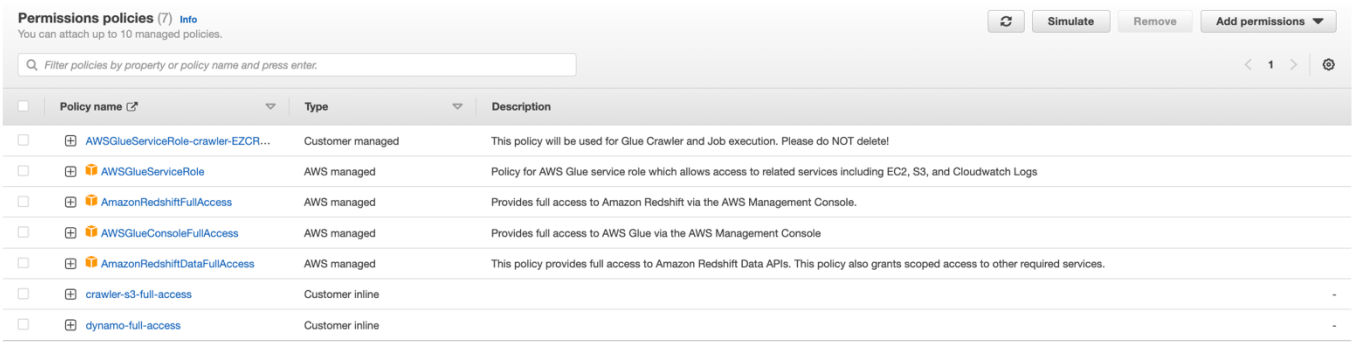
6 Crawling from AWS Redshift

Note that we need to make few changes while creating the crawler from redshift.



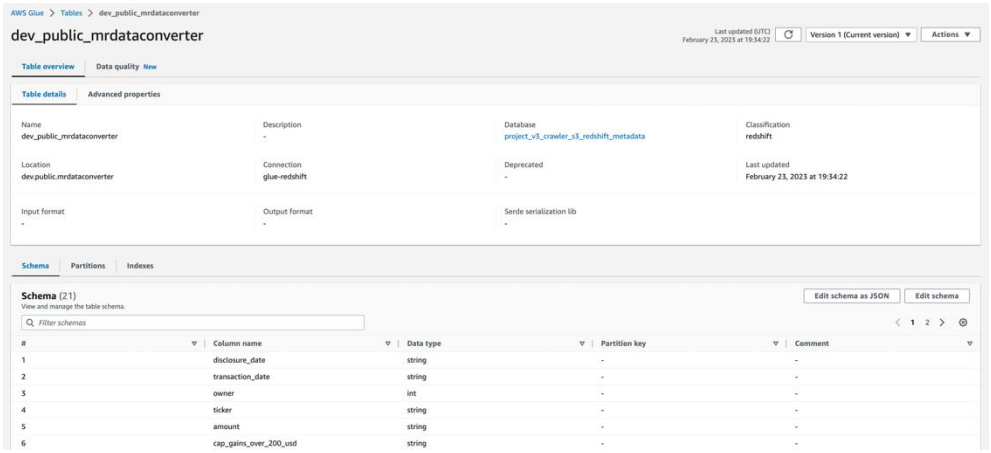
Select the JDBC from the data source, include the path of your table in redshift (make sure you give the same path that we created in the AWS redshift) and click on add a JDBC source.

Navigate to the IAM and attach the role "AWSRedshiftDataFullAccess" to the crawler role (i.e., the crawler role which we used to crawler data from S3)



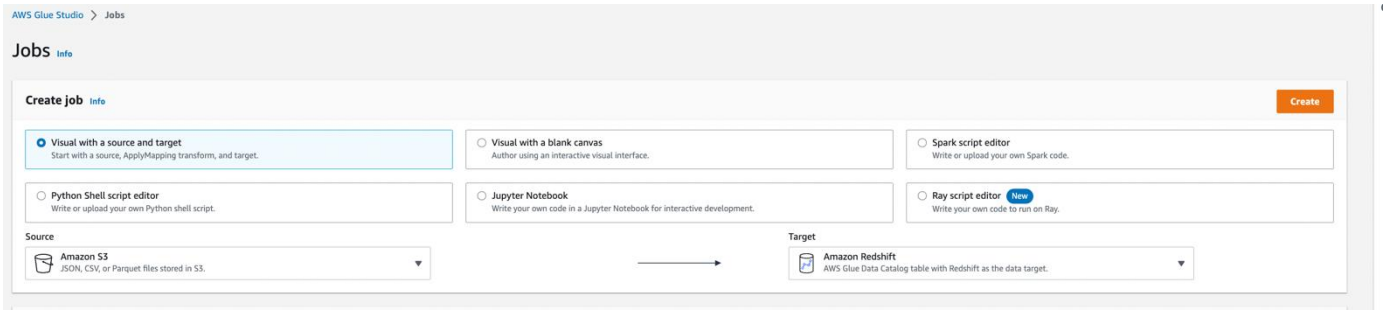
Note that to avoid complexity we are using a single role and attaching the required policies.
For this project, we have created a role to crawl data and attached S3 full access role, Glue service role, Glue console role, AWS redshift full access, and Aws redshift data full access roles. This allows the user to perform the smooth operations of ETL without worrying about the roles.

Once the crawler is created and runs successfully, we must see this schema of the AWS redshift table. The below picture shows the schema of the table from redshift

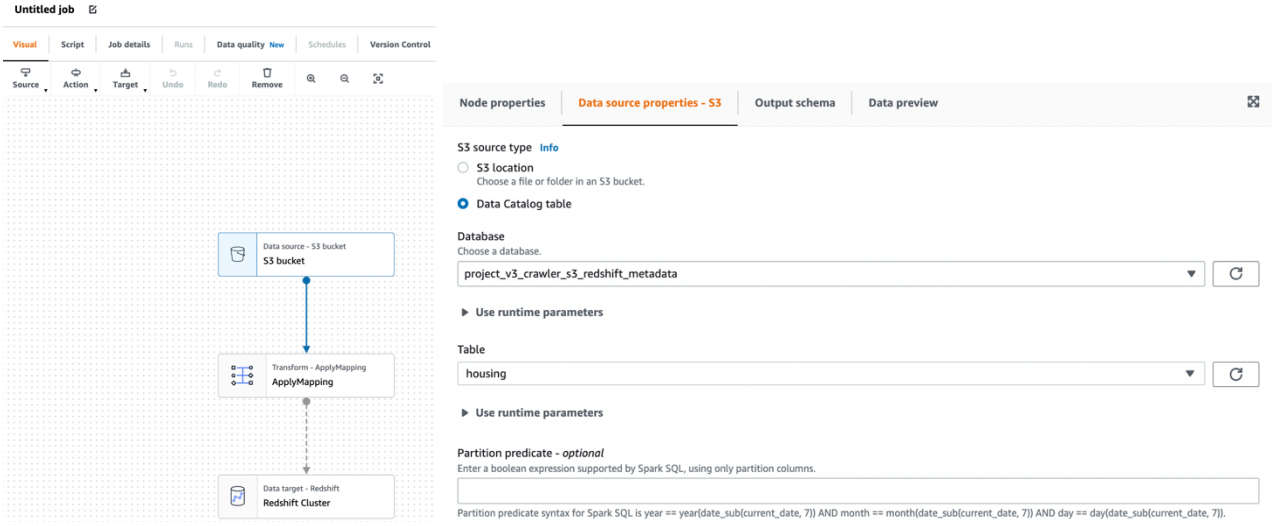


7 Creating and running Glue jobs

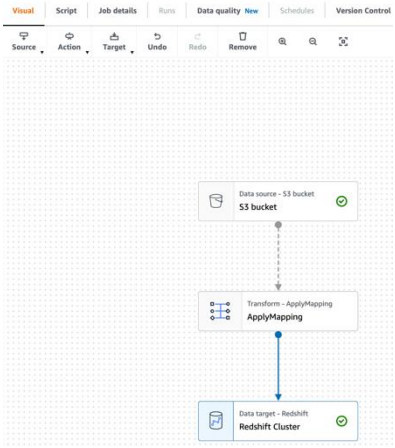
Navigate to the jobs in Data integration and ETL in glue console and select your source as S3 and target as AWS Redshift and click on create job.

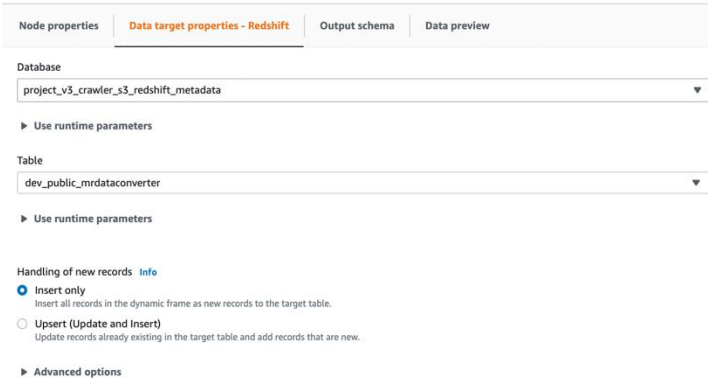


Click on the S3 bucket on data source ➔ select s3 source type as data catalog table ➔ select the database (database where s3 crawled metadata is stored) ➔ select table



Now select the redshift cluster in data target ➔ select the database (database where redshift crawled metadata is stored) ➔ select the table and save the job and run the job.

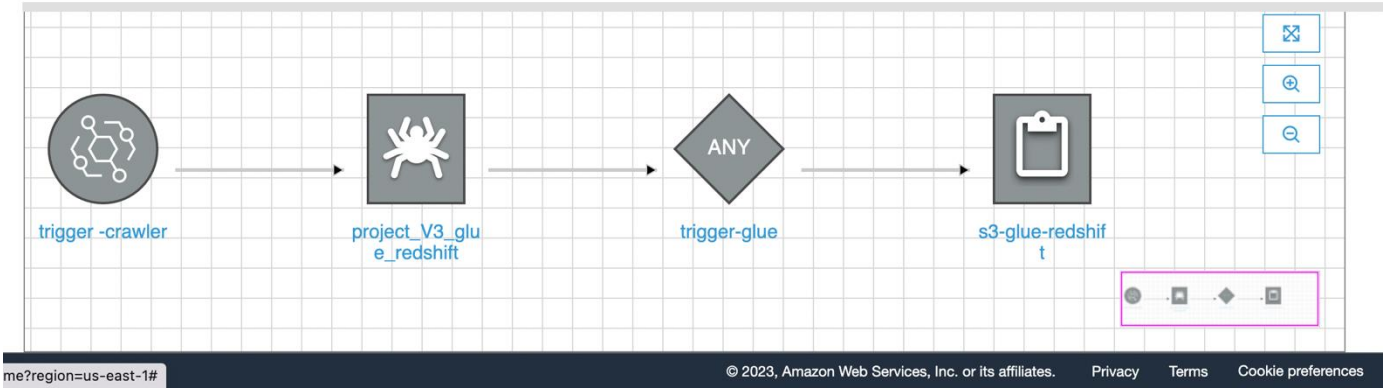




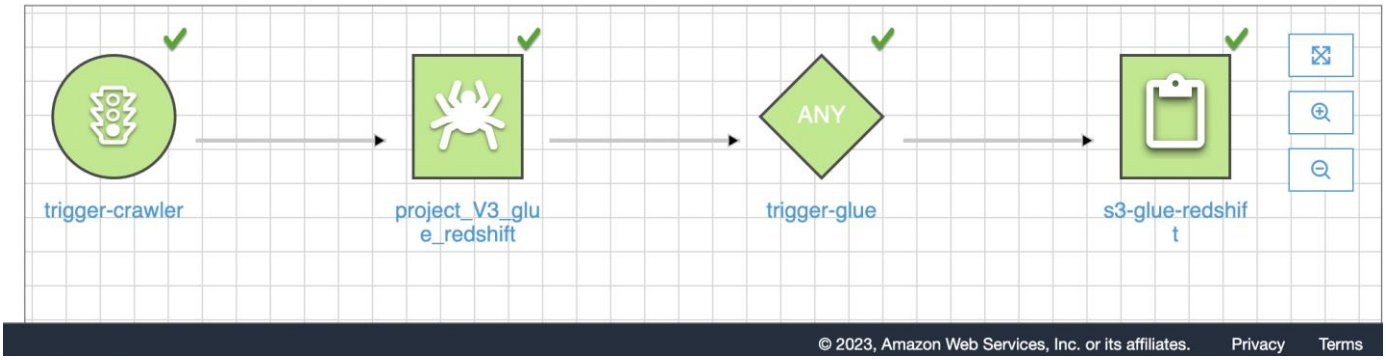
Once the ran successfully, our data will be reflected in the redshift table. To cross-check the data, we should run a select query on the table (select * from table_name) in the AWS redshift table.

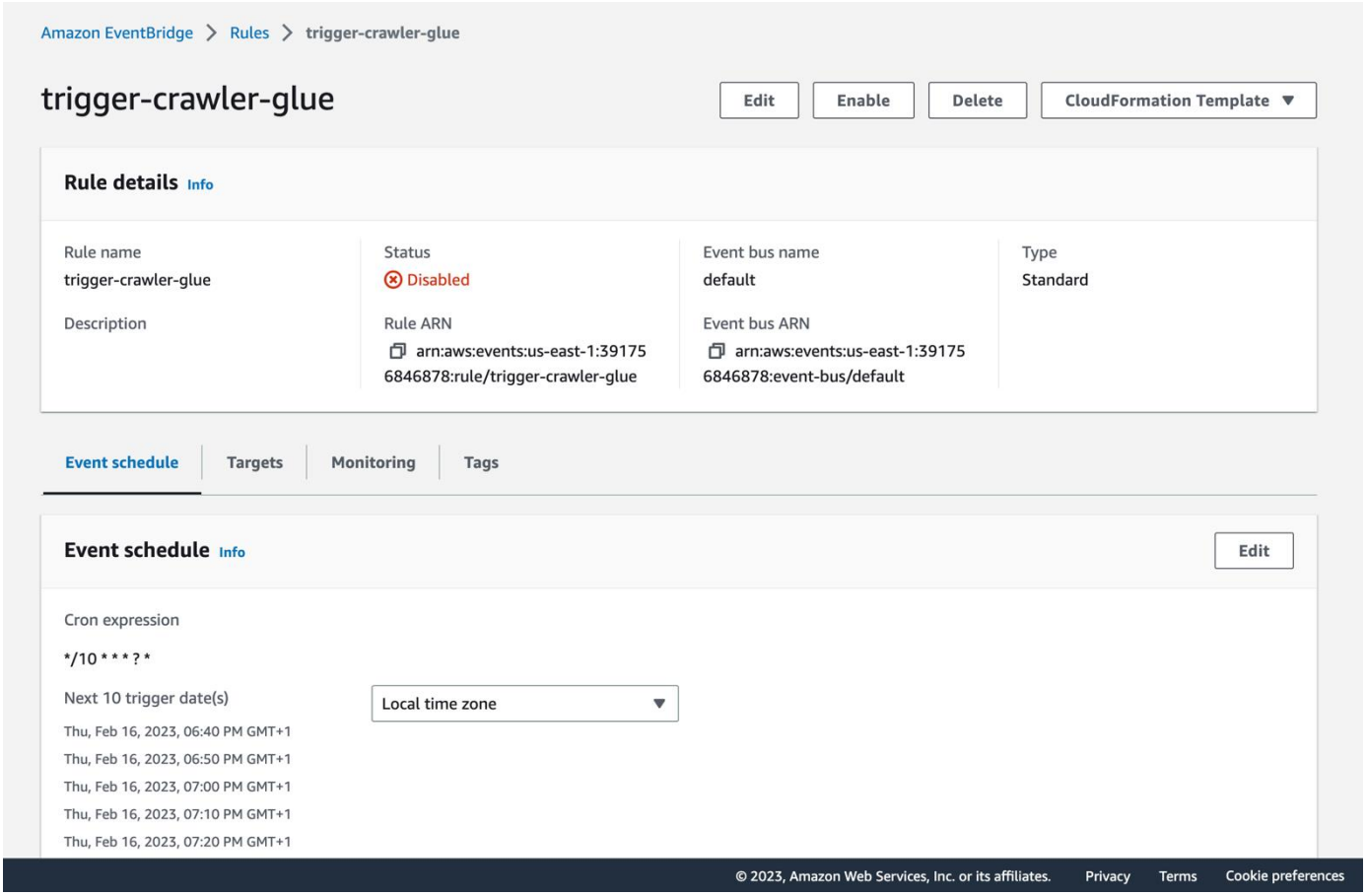
8 Creating the workflow and trigger

create a workflow using crawler (on redshift side) and glue job.



Once we build the trigger using Event bridge, the workflow looks as below picture after running the successful flow.





The above picture shows scheduler to trigger workflow of crawler and glue.