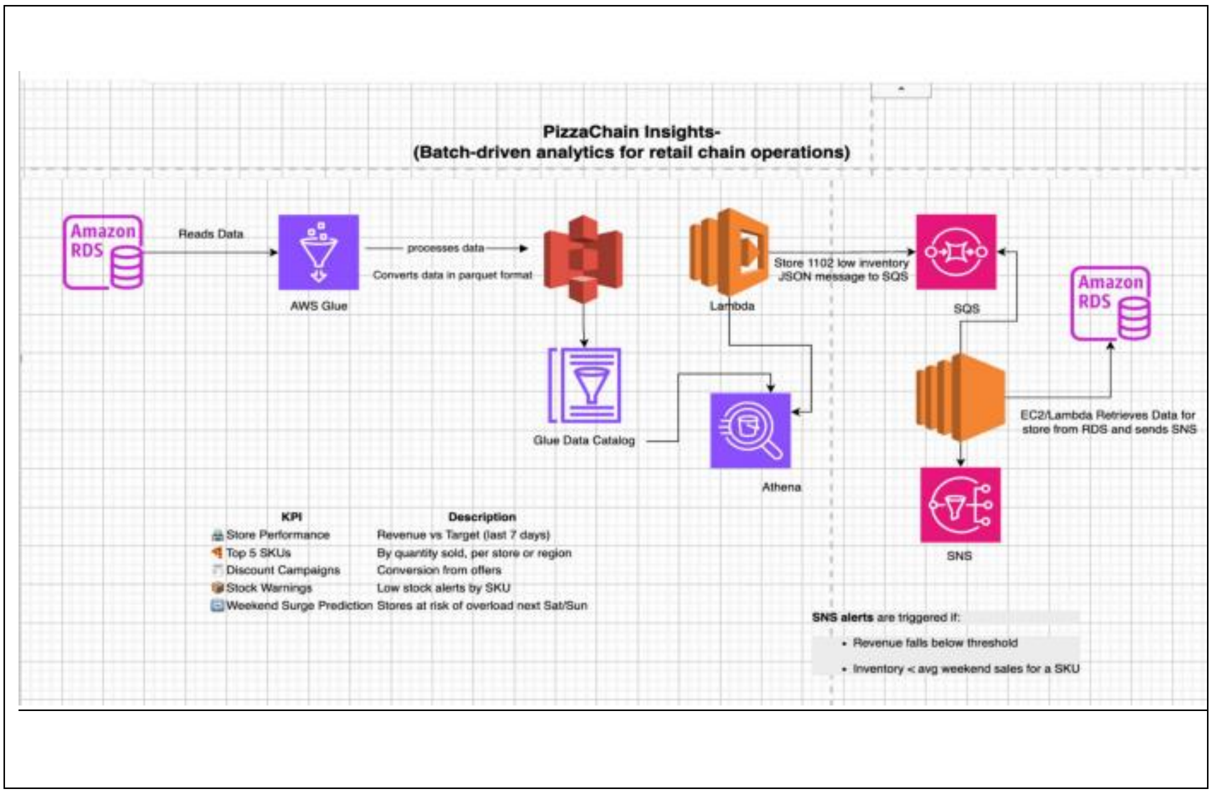
Project Report

Pizza Chain Insights

Priti Ranjan Samal

Contents

PizzaChain Insights - Project Overview


PizzaChain Insights-
(Batch-driven analytics for retail chain operations)
Amazon RDS
Reads Data
AWS Glue
processes data
Converts data in parquet format
Glue Data Catalog
Athena
Lambda
Store 1102 low inventory JSON message to SQS
SQS
Amazon RDS
EC2/Lambda Retrieves Data for store from RDS and sends SNS
SNS

KPI	Description
Store Performance	Revenue vs Target (last 7 days)
Top 5 SKUs	By quantity sold, per store or region
Discount Campaigns	Conversion from offers
Stock Warnings	Low stock alerts by SKU
Weekend Surge Prediction	Stores at risk of overload next Sat/Sun

SNS alerts are triggered if:

- Revenue falls below threshold
- Inventory < avg weekend sales for a SKU

Objective

PizzaChain Insights is a **cloud-native batch analytics pipeline** built to simulate the real-time operational and business intelligence needs of a multi-branch pizza retail chain. The pipeline aims to **ingest, process, and analyze store-level data** such as **orders, inventory, and discounts**, and provide **automated alerts and insights** through a cost-efficient and scalable architecture on **AWS**.

End-to-End Architecture Flow

1. Data Source (RDS)

All operational data is stored in a **MySQL RDS** instance with the following key tables:

- orders
- order_items
- inventory_logs
- discounts_applied
- sku_master

These tables capture data such as:

- Order and item-level transactions
- Discount applications
- SKU-level metadata
- Inventory levels at each store

2. ETL with AWS Glue

- **Glue Crawlers** scan the RDS schema and catalog metadata.
- **Glue ETL Jobs** extract data from RDS tables and write **cleaned, transformed data** into **Amazon S3** in a columnar format (e.g., Parquet or CSV).
- The data is structured and stored by table, enabling efficient query performance via Athena.

3. Analytics via Amazon Athena

- Data stored in S3 is made queryable via **Athena**.
- Complex analytical queries can be written in SQL and run on the S3-backed datasets.
- This enables **low-cost, serverless analytics** on large volumes of data.

4. Automated Alerting (Lambda + SQS + SNS)

- **AWS Lambda (Scheduled)** runs pre-defined Athena queries to:
    - Check for inventory below restock thresholds
    - Identify discount abuse (e.g., >30% discounts)
- Results are pushed to **Amazon SQS (Queue)**.
- A second **Lambda or EC2** service:
    - Reads SQS messages
    - Looks up additional metadata from RDS (like store_id, email_id)
    - Sends **SNS alerts** to store managers or business owners.

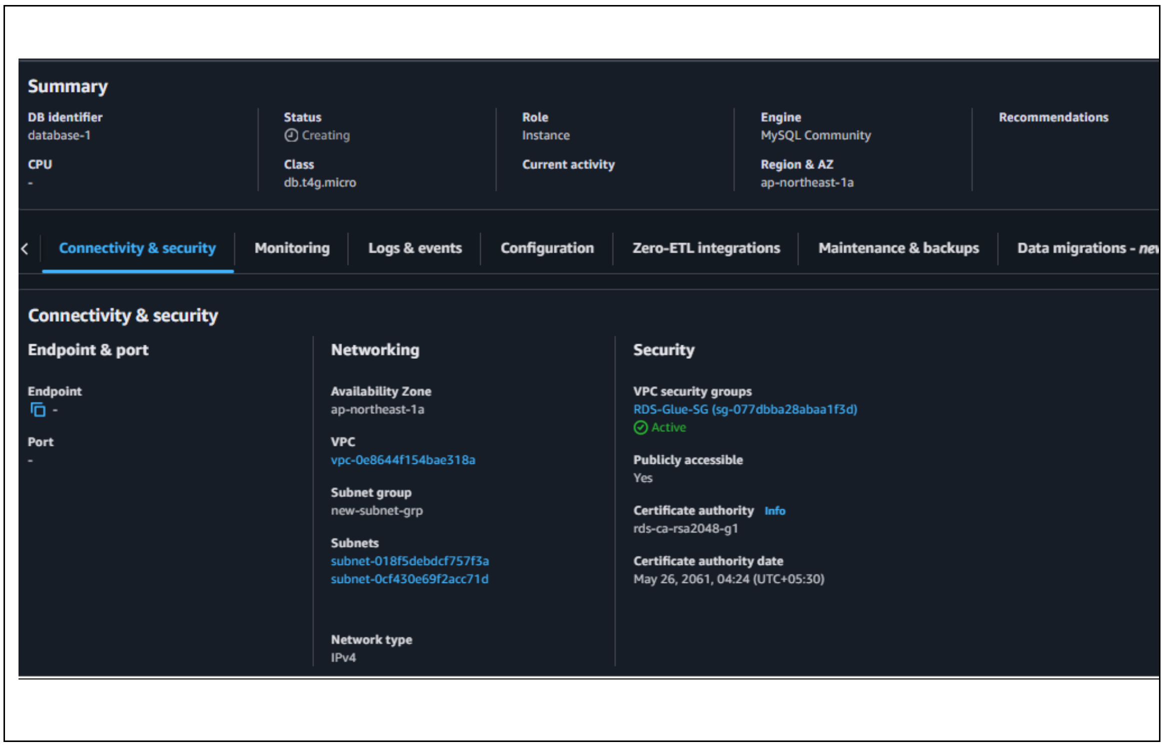
Business Impact

- Enables **data-driven decision-making** across stores and regions.
- Reduces manual effort in identifying **stockouts**, **discount abuse**, and **sales trends**.

Creating Glue Catalog from Data in RDS Database

Creating a RDS Database and inserting data to it

- First we have to create a RDS database with required configuration and attach an instance to it .
- Create an RDS with a database in it note down the VPC and Subnet as it will be used later on .



Summary

DB identifier
database-1

CPU
-

Status
Creating

Class
db.t4g.micro

Role
Instance

Current activity

Engine
MySQL Community

Region & AZ
ap-northeast-1a

Recommendations

Connectivity & security

Endpoint & port

Endpoint
-

Port
-

Networking

Availability Zone
ap-northeast-1a

VPC
vpc-0e8644f154bae318a

Subnet group
new-subnet-grp

Subnets
subnet-018f5debdcf757f3a
subnet-0cf430e69f2acc71d

Network type
IPv4

Security

VPC security groups
RDS-Glue-SG (sg-077dbba28abaa1f3d)
Active

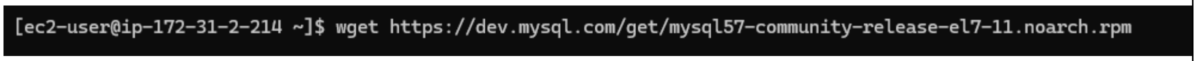
Publicly accessible
Yes

Certificate authority Info
rds-ca-rsa2048-g1

Certificate authority date
May 26, 2061, 04:24 (UTC+05:30)

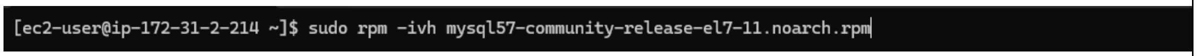
- Once created just save the changes and wait for it to be live .
- Once done we will start with setting up the EC2 instance to run the RDS database over there.
- Prior to that we will have to connect the EC2 instance to the RDS server .
- Once the EC2 instance is setup and logged in to it we will install mysql in EC2 instance and connect it to RDS .

- In order to run sql in our local system we need to install the service first so .



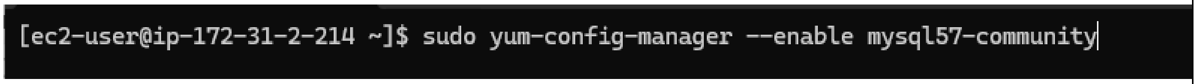
```
[ec2-user@ip-172-31-2-214 ~]$ wget https://dev.mysql.com/get/mysql57-community-release-el7-11.noarch.rpm
```

- This command " https://dev.mysql.com/get/mysql57-community-release-el7-11.noarch.rpm " downloads the MySQL 5.7 YUM repository package .
- This RPM configures system to use MySQL's official YUM repository, enabling to easily install MySQL 5.7 using yum and keep it updated.



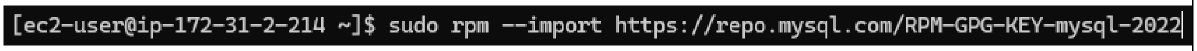
```
[ec2-user@ip-172-31-2-214 ~]$ sudo rpm -ivh mysql57-community-release-el7-11.noarch.rpm
```

- This command " sudo rpm -ivh mysql57-community-release-el7-11.noarch.rpm " installs the downloaded MySQL 5.7 repository package on your system using the RPM package manager.
- The flags -i means install, -v enables verbose output (**Verbose output** means the command will **show more detailed information** about what it's doing during execution), and -h shows a progress bar.
- This sets up the MySQL YUM repo so you can install MySQL 5.7 via yum install.



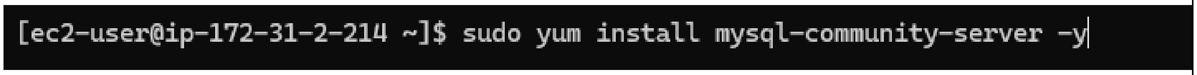
```
[ec2-user@ip-172-31-2-214 ~]$ sudo yum-config-manager --enable mysql57-community
```

- This command **enables the MySQL 5.7 YUM repository** on the system so you can install MySQL 5.7 using yum.



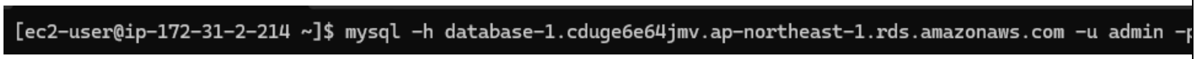
```
[ec2-user@ip-172-31-2-214 ~]$ sudo rpm --import https://repo.mysql.com/RPM-GPG-KEY-mysql-2022
```

- This command **imports the GPG (GNU Privacy Guard) key** used to verify the authenticity of MySQL packages downloaded from the MySQL YUM repository .



```
[ec2-user@ip-172-31-2-214 ~]$ sudo yum install mysql-community-server -y
```

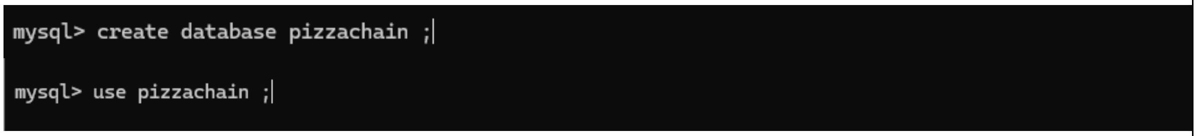
- his command installs the **MySQL Server** (from the MySQL YUM repository) on the EC2 instance using the yum package manager.
- Once the mysql service is installed we will have to start the service followed by connecting it to the RDS using the username and password that was given earlier while setting up the RDS .



```
[ec2-user@ip-172-31-2-214 ~]$ mysql -h database-1.cduge6e64jmv.ap-northeast-1.rds.amazonaws.com -u admin -p
```

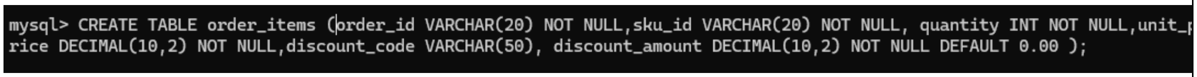
- In this we connect to the Mysql using RDS endpoint here -h represents the endpoint of the RDS , -u is for the username that was created in RDS and -p is for the password set for the same .
- Once logged in to mysql we will create database and create tables within that database .
- And will load the data from local CSV file to the tables ;

- In order to create the tables we will write certain queries .
- We would create a database which will store all our tables withing that particular database .



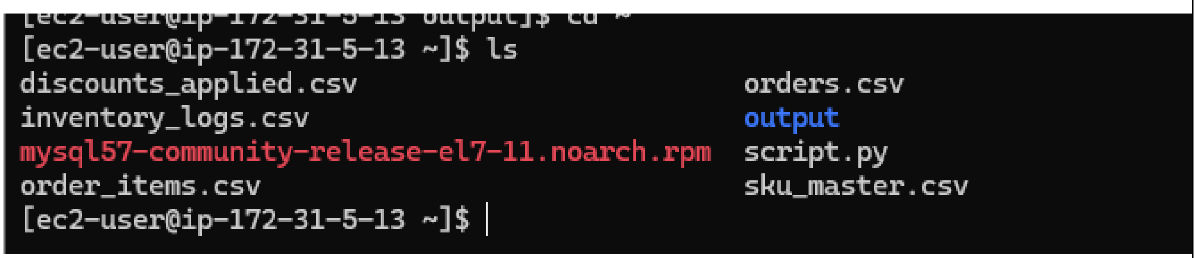
```
mysql> create database pizzachain ;

mysql> use pizzachain ;
```

- Once created the database and switched to the same database we have to create tables within it and insert data to it .
- The following query create a table called as order_items which create attributes such as order_id , sku_id , quantity ,unit_price ,discount_code and many more .



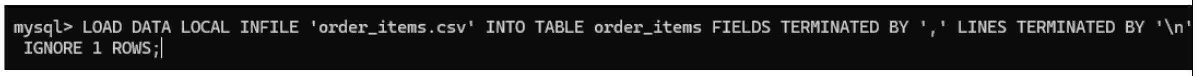
```
mysql> CREATE TABLE order_items (order_id VARCHAR(20) NOT NULL,sku_id VARCHAR(20) NOT NULL, quantity INT NOT NULL,unit_p
rice DECIMAL(10,2) NOT NULL,discount_code VARCHAR(50), discount_amount DECIMAL(10,2) NOT NULL DEFAULT 0.00 );
```

- Once this query is executed it would create table within the earlier mentioned database .
- Now we would have to load data to it . which we would be loading from the EC2 system where the files are stored earlier .



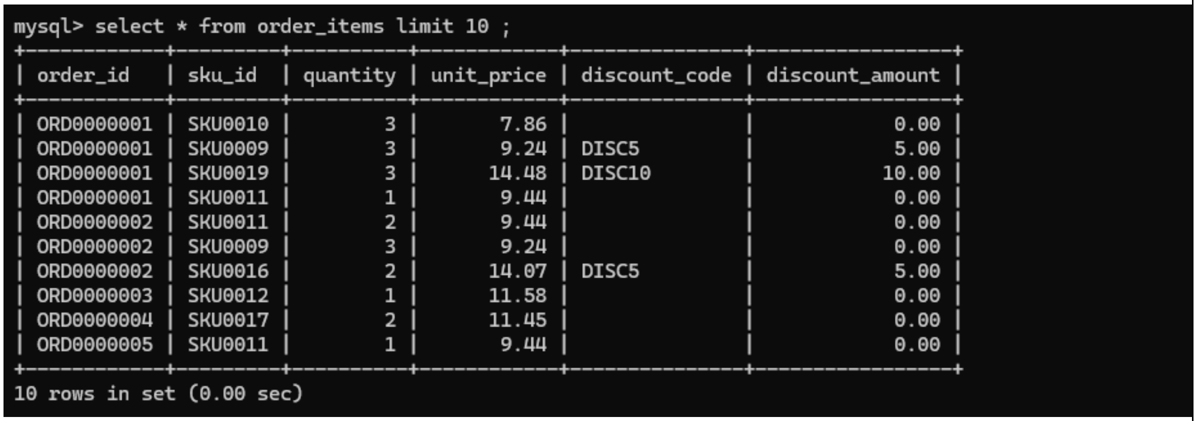
```
[ec2-user@ip-172-31-5-13 output]$ cd ~
[ec2-user@ip-172-31-5-13 ~]$ ls
discounts_applied.csv                          orders.csv
inventory_logs.csv                             output
mysql57-community-release-el7-11.noarch.rpm    script.py
order_items.csv                                sku_master.csv
[ec2-user@ip-172-31-5-13 ~]$
```

- We would be loading the datas from these datasets to the created table .



```
mysql> LOAD DATA LOCAL INFILE 'order_items.csv' INTO TABLE order_items FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
 IGNORE 1 ROWS;
```

- This query load the data into the table from the local csv file which is present in **' /home/ec2-user/ '** directoryand whose path is provided in the query .
- This query instructs to load data where each of the fields are terminated by **' , '** and each line is terminated by **' \n '** and to skip the first row as the first row or record contains the attribute name .

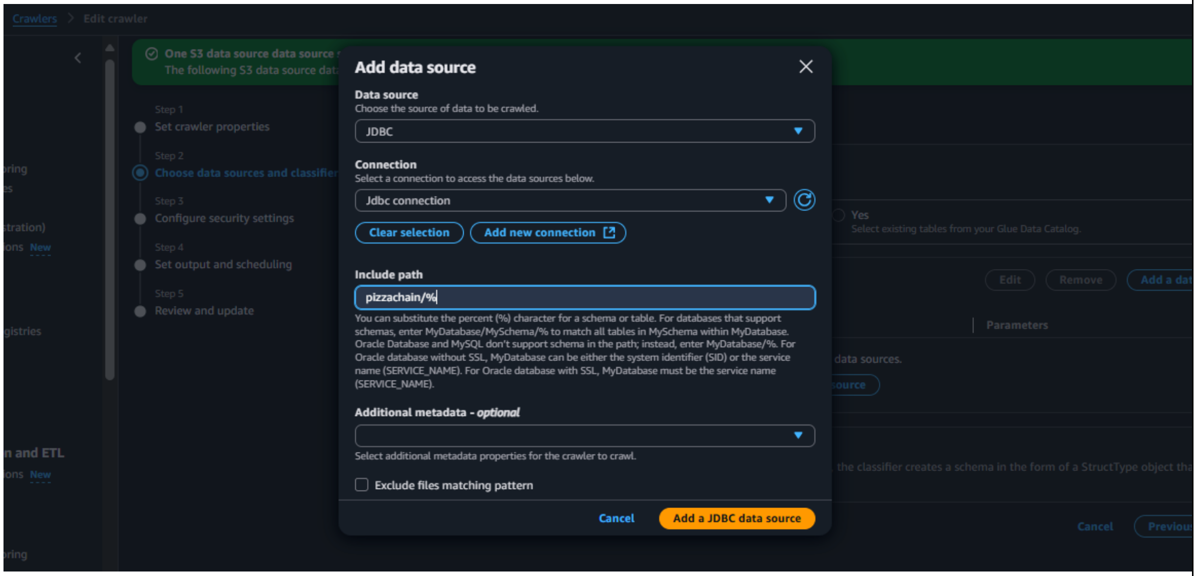


```
mysql> select * from order_items limit 10 ;
+------------+---------+----------+------------+---------------+-----------------+
| order_id   | sku_id  | quantity | unit_price | discount_code | discount_amount |
+------------+---------+----------+------------+---------------+-----------------+
| ORD0000001 | SKU0010 |        3 |       7.86 |               |            0.00 |
| ORD0000001 | SKU0009 |        3 |       9.24 | DISC5         |            5.00 |
| ORD0000001 | SKU0019 |        3 |      14.48 | DISC10        |           10.00 |
| ORD0000001 | SKU0011 |        1 |       9.44 |               |            0.00 |
| ORD0000002 | SKU0011 |        2 |       9.44 |               |            0.00 |
| ORD0000002 | SKU0009 |        3 |       9.24 |               |            0.00 |
| ORD0000002 | SKU0016 |        2 |      14.07 | DISC5         |            5.00 |
| ORD0000003 | SKU0012 |        1 |      11.58 |               |            0.00 |
| ORD0000004 | SKU0017 |        2 |      11.45 |               |            0.00 |
| ORD0000005 | SKU0011 |        1 |       9.44 |               |            0.00 |
+------------+---------+----------+------------+---------------+-----------------+
10 rows in set (0.00 sec)
```

- This query displays the first 10 records of the order_items table and validate that data is successfully loaded to the table .

Creating a Glue catalog from RDS Database using Crawler

- We are storing our raw data in an Amazon RDS (MySQL) database.
- Before we can process or transform this data using AWS Glue or query it using Athena, we need to make Glue aware of the structure of the data (i.e., the schema, table names, columns, data types, etc.).
- To do that, we use a Glue Crawler. The crawler will:
    - Connect to the RDS database
    - Automatically scan all the tables and their metadata
    - Create table definitions in the Glue Data Catalog
    - Let us access these tables in ETL jobs or Athena using SQL-like queries
- This step removes the need for manual schema creation and sets us up for automation and transformation.
- In order to create a glue Crawler go to aws glue and select crawler from left side .
- We need to configure our crawler with proper connection and the database which it would crawl .
- For this I would select data source as JDBC .
- Followed by providing the connection name of the connection between the RDS and the glue that has been already created earlier in the connection section .
- At last in include path I will provide Pizzachain/% so that it can crawl over all the tables within the pizzachain database .



Add data source

Data source
Choose the source of data to be crawled.
JDBC

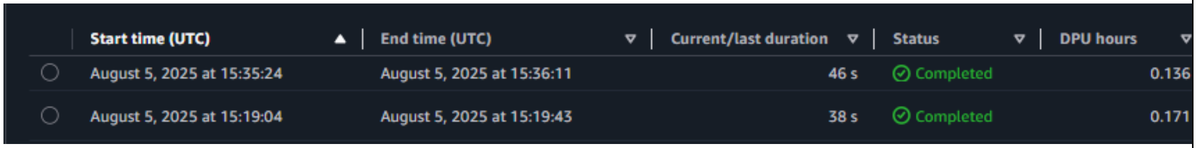
Connection
Select a connection to access the data sources below.
Jdbc connection

Include path
pizzachain/%
You can substitute the percent (%) character for a schema or table. For databases that support schemas, enter MyDatabase/MySchema/% to match all tables in MySchema within MyDatabase. Oracle Database and MySQL don't support schema in the path; instead, enter MyDatabase/%. For Oracle database without SSL, MyDatabase can be either the system identifier (SID) or the service name (SERVICE_NAME). For Oracle database with SSL, MyDatabase must be the service name (SERVICE_NAME).

Additional metadata - *optional*
Select additional metadata properties for the crawler to crawl.

☐ Exclude files matching pattern

- Once configured the crawler we would run the crawler to check whether it succeeded or failed .

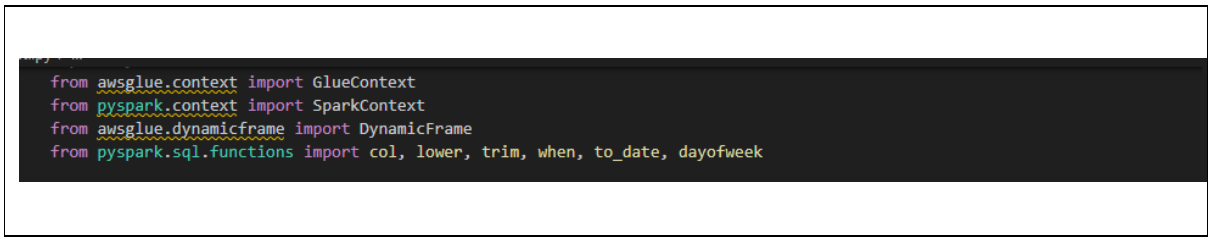


Start time (UTC)	End time (UTC)	Current/last duration	Status	DPU hours
August 5, 2025 at 15:35:24	August 5, 2025 at 15:36:11	46 s	Completed	0.136
August 5, 2025 at 15:19:04	August 5, 2025 at 15:19:43	38 s	Completed	0.171

- As can be seen it successfully crawled the database and created the catalogs . So now we can directly run a spark job to create the datasource from it and store it in S3 in parquet format .

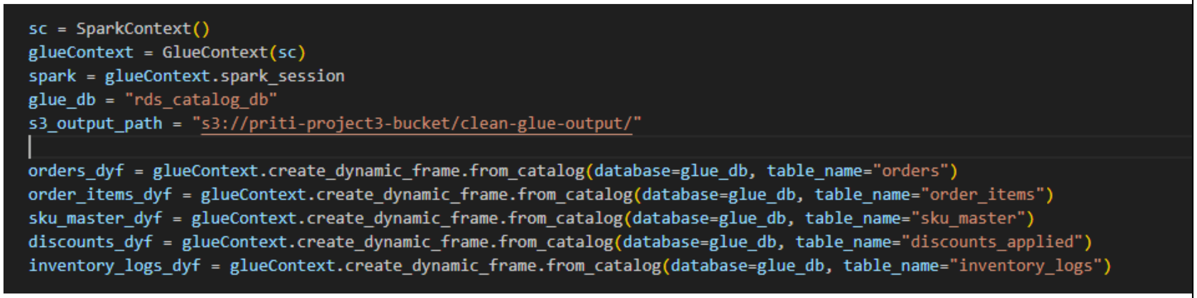
Reading Data from RDS and ingesting it to S3 bucket in parquet format

- Once the data is stored in Amazon RDS, we want to process it and move it to Amazon S3 in a clean, structured format like Parquet so it can be queried using Athena or used for downstream analytics.
- To do this, we create a Glue ETL job using PySpark. This job will read from the Glue Data Catalog (which references RDS tables), apply basic cleaning, and write the data to S3 in Parquet format.
- This code use GlueContext to work with AWS Glue features like reading tables from the Data Catalog , SparkContext to start Spark so my job can run.
- It uses DynamicFrame because it works well with Glue and handles semi-structured data easily , and import functions like col, lower, trim, when, to_date, and dayofweek to clean and transform my data during the ETL process.



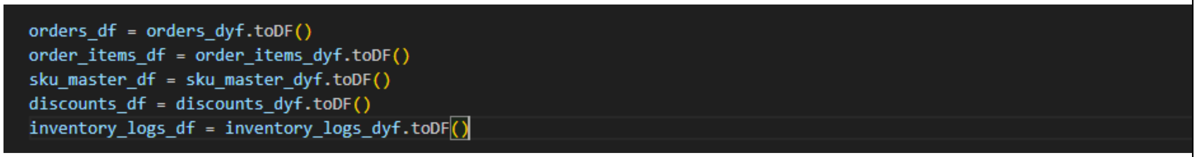
```
from awsglue.context import GlueContext
from pyspark.context import SparkContext
from awsglue.dynamicframe import DynamicFrame
from pyspark.sql.functions import col, lower, trim, when, to_date, dayofweek
```

- Here we Start Spark using SparkContext() so the job can run. Create GlueContext is used to access Glue features.
- Get the Spark session from glueContext for DataFrame operations and Set glue_db as the source database from the Glue Data Catalog.
- Define s3_output_path as the S3 location to store the cleaned data.



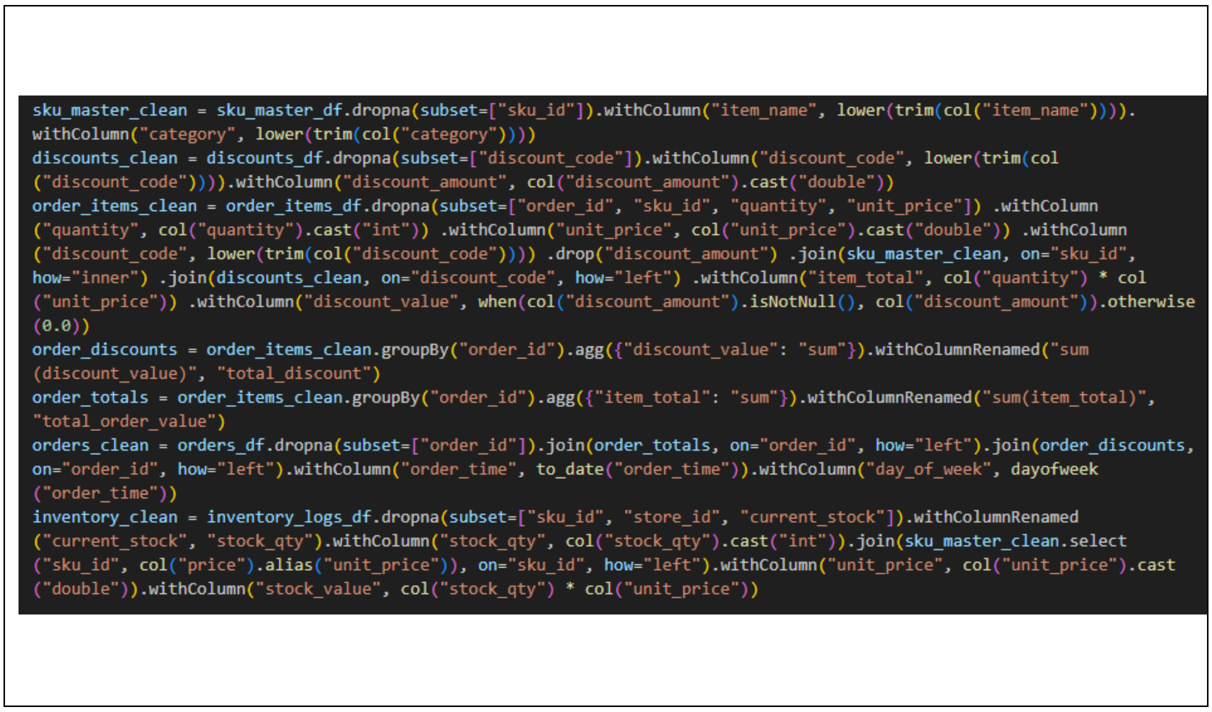
```
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
glue_db = "rds_catalog_db"
s3_output_path = "s3://priti-project3-bucket/clean-glue-output/"

orders_dyf = glueContext.create_dynamic_frame.from_catalog(database=glue_db, table_name="orders")
order_items_dyf = glueContext.create_dynamic_frame.from_catalog(database=glue_db, table_name="order_items")
sku_master_dyf = glueContext.create_dynamic_frame.from_catalog(database=glue_db, table_name="sku_master")
discounts_dyf = glueContext.create_dynamic_frame.from_catalog(database=glue_db, table_name="discounts_applied")
inventory_logs_dyf = glueContext.create_dynamic_frame.from_catalog(database=glue_db, table_name="inventory_logs")
```

- I'm reading five tables from the **Glue Data Catalog**, which holds metadata about my RDS tables after running the crawler.
- I use from_catalog() to load each table by giving the **Glue database name** and **table name**.
- Each table is loaded into a **DynamicFrame**:
  - orders_dyf → for the orders table
  - order_items_dyf → for order_items, and so on
- It's Glue's version of a DataFrame but with **more flexibility**.
- Handles **semi-structured data** better than normal DataFrames (like nested JSON, missing fields).
- Allows automatic schema inference and self-healing during transformations.
- This gives me a way to start transforming or cleaning the data inside Glue.



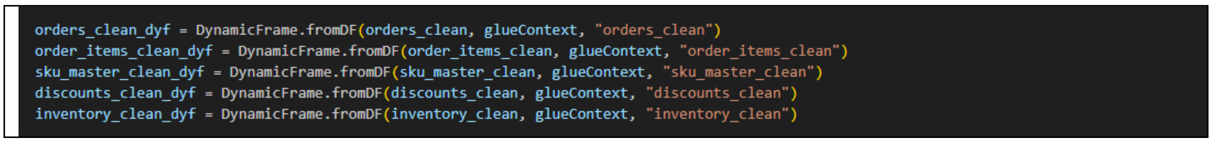
```
orders_df = orders_dyf.toDF()
order_items_df = order_items_dyf.toDF()
sku_master_df = sku_master_dyf.toDF()
discounts_df = discounts_dyf.toDF()
inventory_logs_df = inventory_logs_dyf.toDF()
```

- I'm converting each **DynamicFrame** into a **Spark DataFrame** using .toDF()
- This gives me more power to apply Spark SQL functions and transformations.



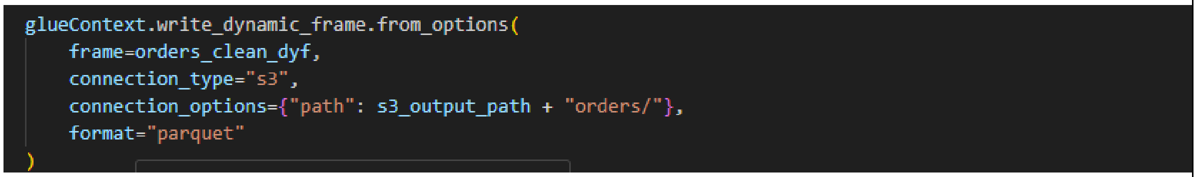
```
sku_master_clean = sku_master_df.dropna(subset=["sku_id"]).withColumn("item_name", lower(trim(col("item_name")))).
withColumn("category", lower(trim(col("category"))))
discounts_clean = discounts_df.dropna(subset=["discount_code"]).withColumn("discount_code", lower(trim(col
("discount_code")))).withColumn("discount_amount", col("discount_amount").cast("double"))
order_items_clean = order_items_df.dropna(subset=["order_id", "sku_id", "quantity", "unit_price"]) .withColumn
("quantity", col("quantity").cast("int")) .withColumn("unit_price", col("unit_price").cast("double")) .withColumn
("discount_code", lower(trim(col("discount_code")))) .drop("discount_amount") .join(sku_master_clean, on="sku_id",
how="inner") .join(discounts_clean, on="discount_code", how="left") .withColumn("item_total", col("quantity") * col
("unit_price")) .withColumn("discount_value", when(col("discount_amount").isNotNull(), col("discount_amount")).otherwise
(0.0))
order_discounts = order_items_clean.groupBy("order_id").agg({"discount_value": "sum"}).withColumnRenamed("sum
(discount_value)", "total_discount")
order_totals = order_items_clean.groupBy("order_id").agg({"item_total": "sum"}).withColumnRenamed("sum(item_total)",
"total_order_value")
orders_clean = orders_df.dropna(subset=["order_id"]).join(order_totals, on="order_id", how="left").join(order_discounts,
on="order_id", how="left").withColumn("order_time", to_date("order_time")).withColumn("day_of_week", dayofweek
("order_time"))
inventory_clean = inventory_logs_df.dropna(subset=["sku_id", "store_id", "current_stock"]).withColumnRenamed
("current_stock", "stock_qty").withColumn("stock_qty", col("stock_qty").cast("int")).join(sku_master_clean.select
("sku_id", col("price").alias("unit_price")), on="sku_id", how="left").withColumn("unit_price", col("unit_price").cast
("double")).withColumn("stock_value", col("stock_qty") * col("unit_price"))
```

- This block is cleaning and preparing multiple datasets for analysis. I start by removing records with missing critical values, like IDs or prices. Then I clean up text fields (like converting to lowercase and trimming spaces) and make sure numbers like prices, quantities, and discounts are in the correct format.
- After cleaning, I combine the order items with product and discount information. I calculate how much each item costs in total and what discount applies. Then I aggregate the data to get total revenue and total discount per order.
- For orders, I also convert the order time to a date and extract the day of the week to help with trend analysis. Similarly, I process the inventory data to compute how much value each product holds in stock, store-wise.



```
orders_clean_dyf = DynamicFrame.fromDF(orders_clean, glueContext, "orders_clean")
order_items_clean_dyf = DynamicFrame.fromDF(order_items_clean, glueContext, "order_items_clean")
sku_master_clean_dyf = DynamicFrame.fromDF(sku_master_clean, glueContext, "sku_master_clean")
discounts_clean_dyf = DynamicFrame.fromDF(discounts_clean, glueContext, "discounts_clean")
inventory_clean_dyf = DynamicFrame.fromDF(inventory_clean, glueContext, "inventory_clean")
```

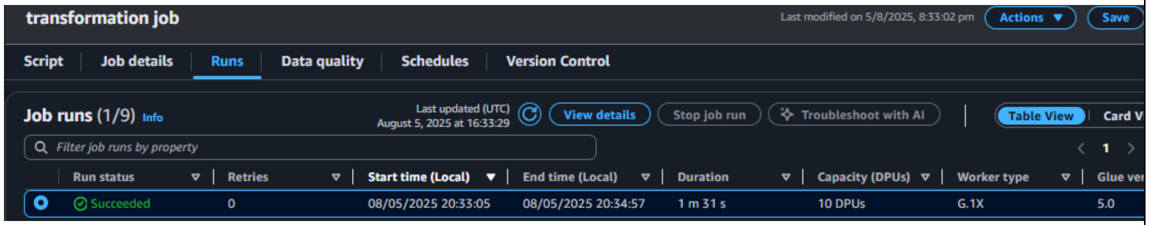
- This block is converting them back into DynamicFrames using DynamicFrame.fromDF() so that I can work with them in AWS Glue and write them easily to S3 or use them in further Glue jobs.



```
glueContext.write_dynamic_frame.from_options(
    frame=orders_clean_dyf,
    connection_type="s3",
    connection_options={"path": s3_output_path + "orders/"},
    format="parquet"
)
```

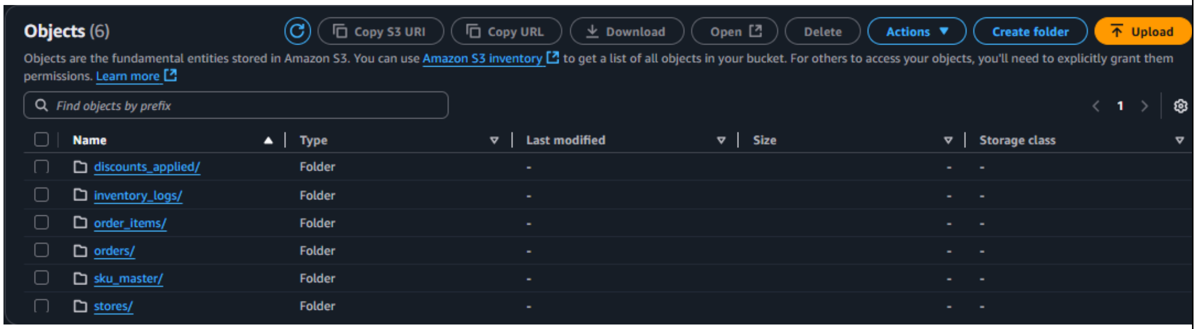
This block of code is writing the cleaned orders_clean_dyf DynamicFrame to S3 in **Parquet** format.

- I specify the **frame** to write (orders_clean_dyf)
- Use "s3" as the **connection_type**
- Set the **output path** to a subfolder called "orders/" inside my main S3 path
- Choose "parquet" as the **file format** for better performance and compression in analytics.
- And did this for all other dynamic dataframes also .
- After successfully saving the data, a confirmation message is printed, and the Spark session is stopped with spark.stop() to release resources and end the ETL job cleanly.
- Once this is done save the job and execute it and check if the code is executed successfully or not



Run status	Retries	Start time (Local)	End time (Local)	Duration	Capacity (DPUs)	Worker type	Glue ver
Succeeded	0	08/05/2025 20:33:05	08/05/2025 20:34:57	1 m 31 s	10 DPUs	G.1X	5.0

- Once this is done go to S3 bucket and do check for if the data is stored in the earlier provided S3 bucket location .

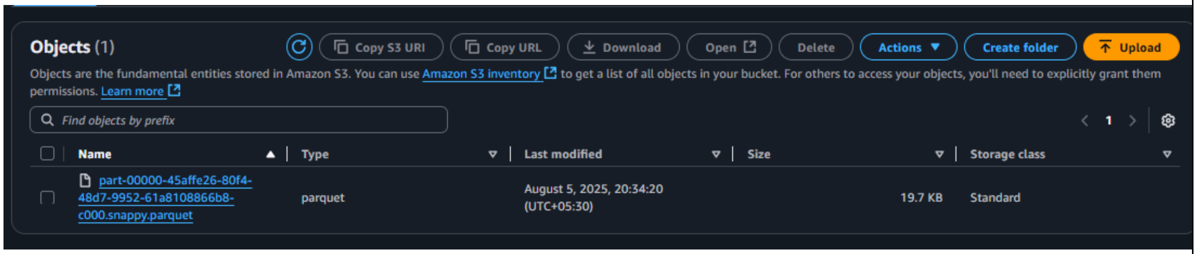


Objects (6)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
discounts_applied/	Folder	-	-	-
inventory_logs/	Folder	-	-	-
order_items/	Folder	-	-	-
orders/	Folder	-	-	-
sku_master/	Folder	-	-	-
stores/	Folder	-	-	-

- As can be seen data is successfully loaded into the desired directory .



Objects (1)

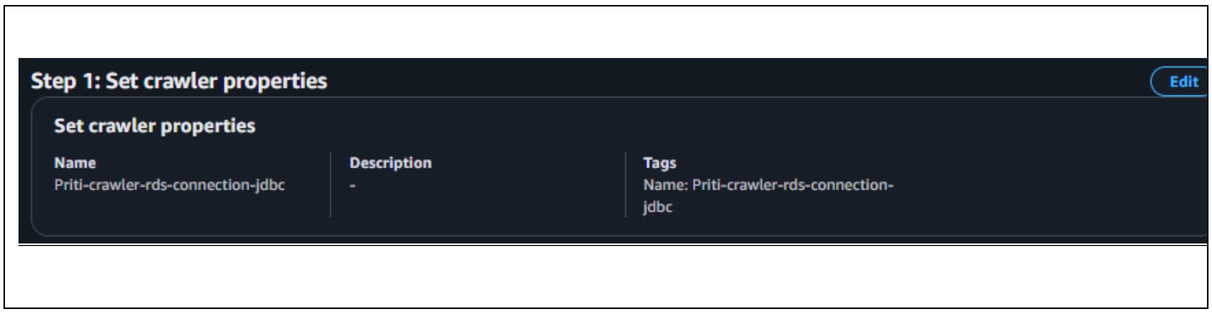
Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
part-00000-45affe26-80f4-48d7-9952-61a8108866b8-c000.snappy.parquet	parquet	August 5, 2025, 20:34:20 (UTC+05:30)	19.7 KB	Standard

- Also as can be seen as we earlier mentioned that the data is to be stored in parquet format we have achieved it over here.
- Further we will be loading the data in Athena using S3 bucket as the data source and by using crawler .

Creating Athena Tables using S3 bucket as a data source

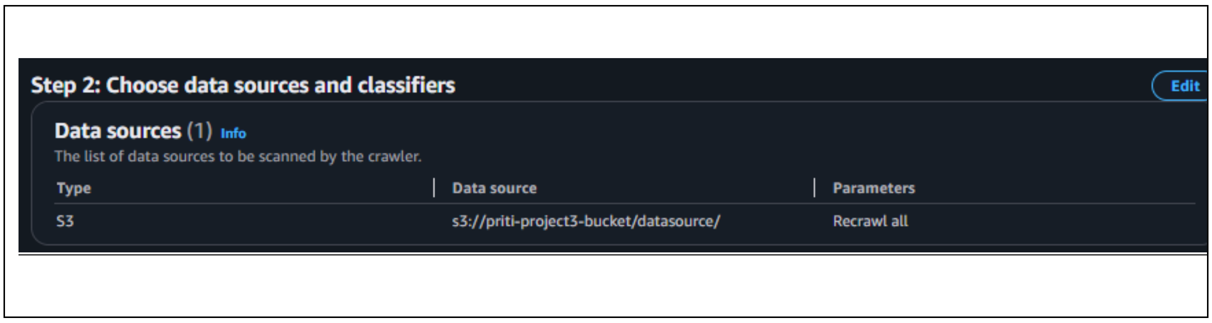
- In order to run queries first we will have to create tables in Athena using the datasource stored in the S3 bucket .
- In order to load it first we will create a crawler which can access the S3 bucket data source .
- Create a crawler with . Provide correct data source and the destination database .


Step 1: Set crawler properties

Set crawler properties

Name	Description	Tags
Priti-crawler-rds-connection-jdbc	-	Name: Priti-crawler-rds-connection-jdbc

- Here we do set a name for the crawler which can be easily recognizable and easy to understand

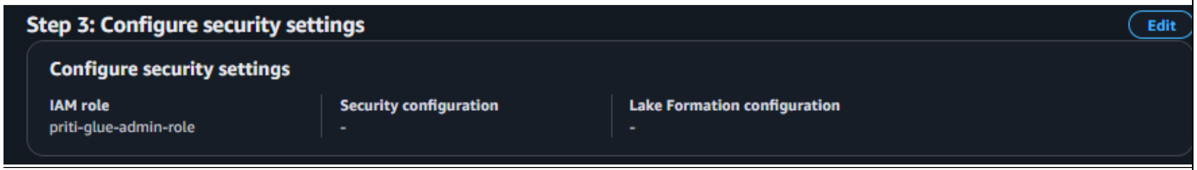

Step 2: Choose data sources and classifiers

Data sources (1)

The list of data sources to be scanned by the crawler.

Type	Data source	Parameters
S3	s3://priti-project3-bucket/datasource/	Recrawl all

- This block defines the data source from where it will access the parquet file with the data in it and will create the table in athena using the same .
- I am directly giving the home directory path because crawlers have a feature to crawl all the subfolders until they get a data so selecting a main directory containing the directories for each of the tables and data in it will prevent us from creating multiple sources or crawlers .


Step 3: Configure security settings

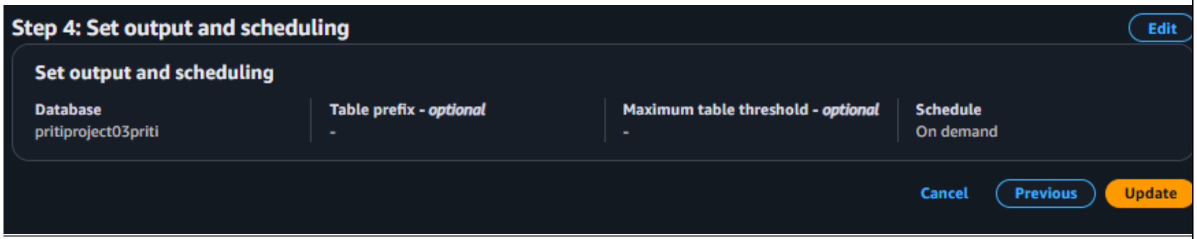
Configure security settings

IAM role
priti-glue-admin-role

Security configuration
-

Lake Formation configuration
-

- In order to able to access the S3 and read data from it the glue crawler should have rightful permission to do so. So here we create an IAM role and attach it over here in order to give access to the crawler to access the S3 .


Step 4: Set output and scheduling

Set output and scheduling

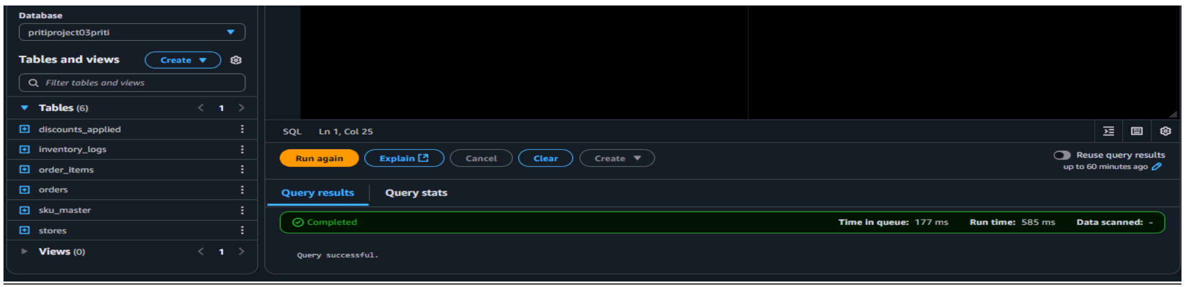
Database
pritiproject03priti

Table prefix - *optional*
-

Maximum table threshold - *optional*
-

Schedule
On demand

- This gives the path to store the tables i.e it specifies which database will be used in order to store the tables created from the parquet data source stored in the S3 bucket.
- Once created run the crawler and do check if it succeeded or not .
- Once succeeded go to Athena and check for tables and data in it . Check If it is correct or not .


Database
pritiproject03priti

Tables and views

Tables (6)
discounts_applied
inventory_logs
order_items
orders
sku_master
stores

Views (0)

SQL Ln 1, Col 25

Reuse query results up to 60 minutes ago

Completed Time in queue: 177 ms Run time: 585 ms Data scanned: -

```
Query successful.
```

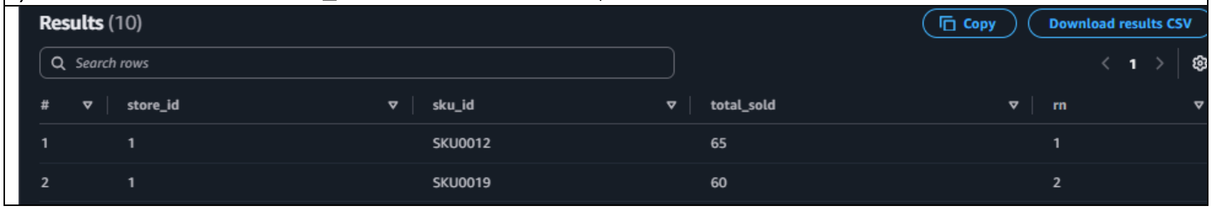
Performing Queries on Athena table's

- Once the tables are created in athena we will have to perform queries in it in order to check if we get the desired results .
- As checked above the data is correctly loading into the tables so now we will be working with the queries .

Query Execution

1. Top 5 Selling SKUs per Store in the Last 7 Days

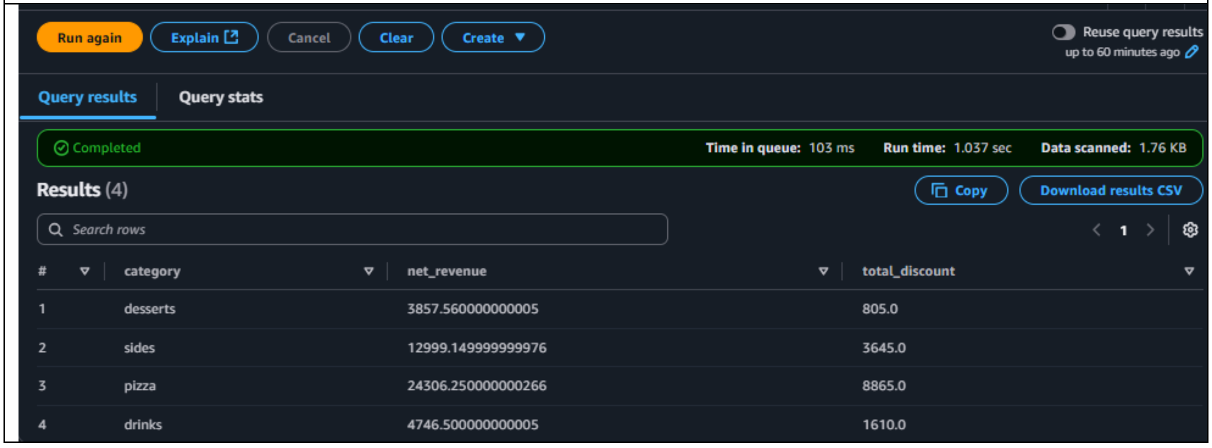
```
WITH ranked_sales AS (
 SELECT
  o.store_id,
  oi.sku_id,
  SUM(oi.quantity) AS total_sold,
  ROW_NUMBER() OVER (PARTITION BY o.store_id ORDER BY SUM(oi.quantity) DESC) AS rn
 FROM orders o
 JOIN order_items oi ON o.order_id = oi.order_id
 WHERE o.order_time >= current_timestamp - INTERVAL '7' DAY
 GROUP BY o.store_id, oi.sku_id
)SELECT * FROM ranked_sales WHERE rn <= 5;
```



#	store_id	sku_id	total_sold	rn
1	1	SKU0012	65	1
2	1	SKU0019	60	2

2. Category-wise Revenue Breakdown with Discounts Applied

```
SELECT
    sm.category,
    SUM((oi.unit_price * oi.quantity) - oi.discount_amount) AS net_revenue,
    SUM(oi.discount_amount) AS total_discount
FROM order_items oi
JOIN sku_master sm ON oi.sku_id = sm.sku_id
GROUP BY sm.category;
```



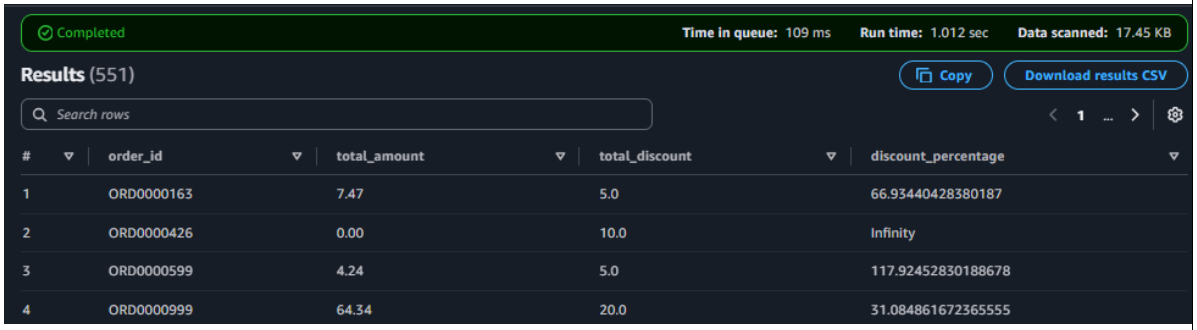
Completed Time in queue: 103 ms Run time: 1.037 sec Data scanned: 1.76 KB

Results (4)

#	category	net_revenue	total_discount
1	desserts	3857.560000000005	805.0
2	sides	12999.149999999976	3645.0
3	pizza	24306.250000000266	8865.0
4	drinks	4746.500000000005	1610.0

3. Identify Orders Where Discount > 30% of Total Value

```
SELECT
   o.order_id,
   o.total_amount,
   SUM(oi.discount_amount) AS total_discount,
   SUM(oi.discount_amount) / o.total_amount * 100 AS
discount_percentage
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
GROUP BY o.order_id, o.total_amount
HAVING (SUM(oi.discount_amount) / o.total_amount) > 0.30;
```



Time in queue: 109 ms Run time: 1.012 sec Data scanned: 17.45 KB

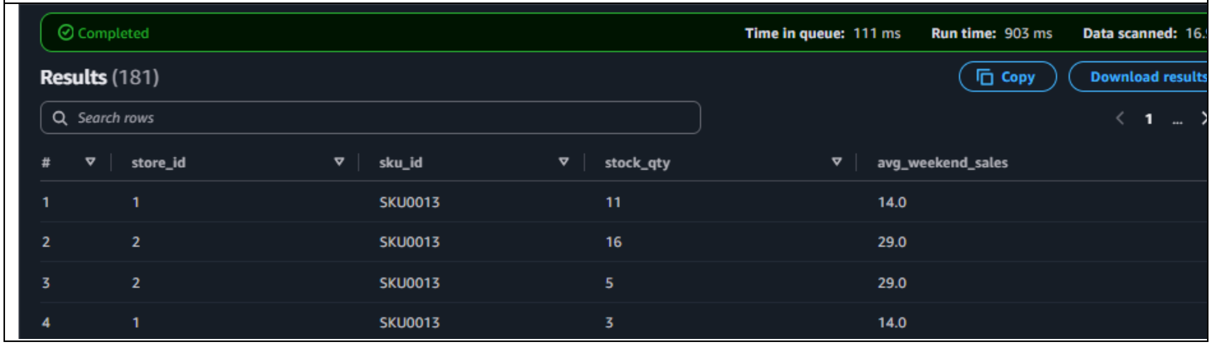
Results (551)

#	order_id	total_amount	total_discount	discount_percentage
1	ORD0000163	7.47	5.0	66.93440428380187
2	ORD0000426	0.00	10.0	Infinity
3	ORD0000599	4.24	5.0	117.92452830188678
4	ORD0000999	64.34	20.0	31.084861672365555

4. Low Inventory Alert Based on Last Weekend's Average Sales

```
WITH weekend_sales AS (
  SELECT
    oi.sku_id,
    o.store_id,
    SUM(oi.quantity) / 2.0 AS avg_weekend_sales
  FROM orders o
  JOIN order_items oi ON o.order_id = oi.order_id
  WHERE day_of_week(o.order_time) IN (6, 7)
  GROUP BY oi.sku_id, o.store_id
)
SELECT
  il.store_id,
  il.sku_id,
  il.stock_qty,
  ws.avg_weekend_sales
FROM inventory_logs il
JOIN weekend_sales ws
```

```
  ON il.sku_id = ws.sku_id AND il.store_id = ws.store_id
WHERE il.stock_qty < ws.avg_weekend_sales;
```

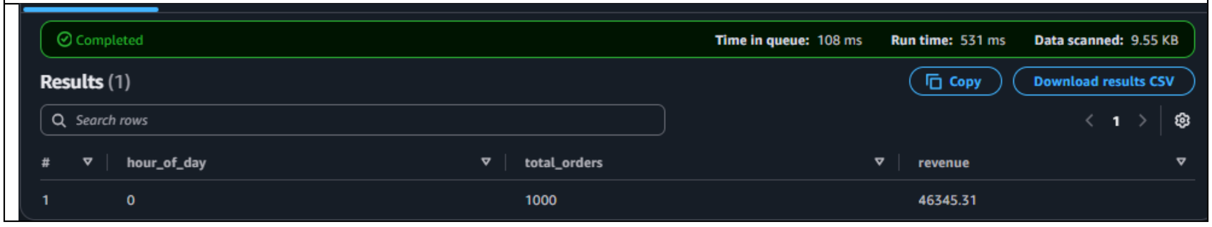


```
Completed                    Time in queue: 111 ms   Run time: 903 ms   Data scanned: 16.
Results (181)
```

#	store_id	sku_id	stock_qty	avg_weekend_sales
1	1	SKU0013	11	14.0
2	2	SKU0013	16	29.0
3	2	SKU0013	5	29.0
4	1	SKU0013	3	14.0

5. Revenue and Orders by Hour of Day

```
SELECT
    hour(order_time) AS hour_of_day,
    COUNT(DISTINCT order_id) AS total_orders,
    SUM(total_amount) AS revenue
FROM orders
GROUP BY hour(order_time)
ORDER BY hour(order_time);
```

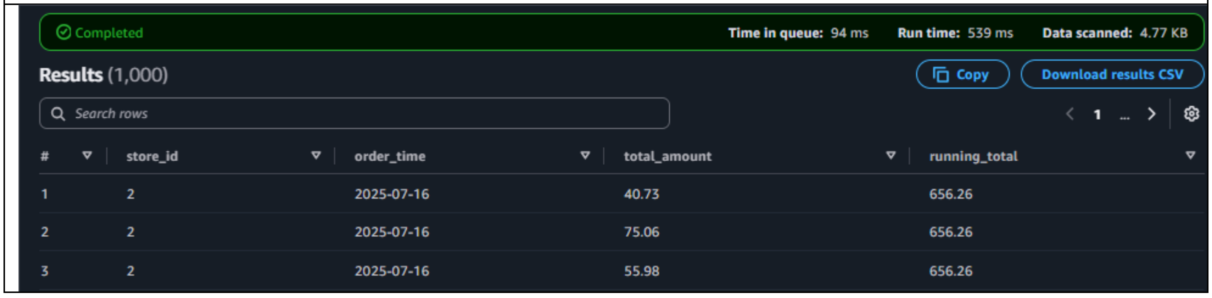


```
Completed                    Time in queue: 108 ms   Run time: 531 ms   Data scanned: 9.55 KB
Results (1)
```

#	hour_of_day	total_orders	revenue
1	0	1000	46345.31

6. Running Total of Revenue by Store

```
SELECT

   store_id,

   order_time,

   total_amount,

   SUM(total_amount) OVER (PARTITION BY store_id ORDER BY
order_time) AS running_total

FROM orders;
```



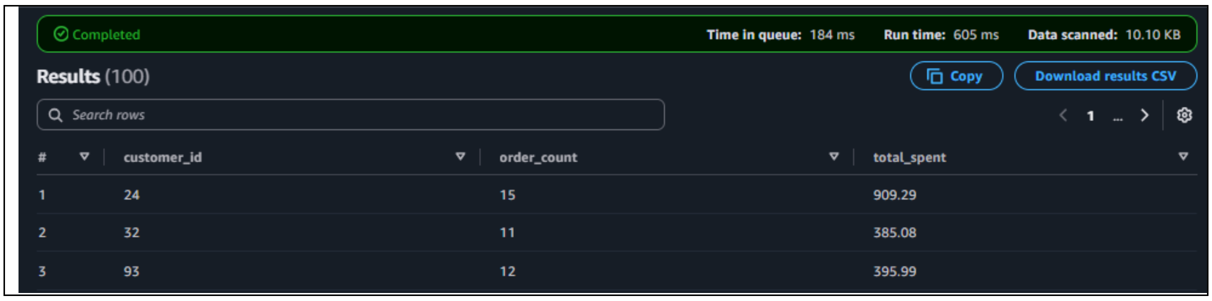
Time in queue: 94 ms Run time: 539 ms Data scanned: 4.77 KB

Results (1,000)

#	store_id	order_time	total_amount	running_total
1	2	2025-07-16	40.73	656.26
2	2	2025-07-16	75.06	656.26
3	2	2025-07-16	55.98	656.26

7. Customers with High Frequency and High Value Orders

```
SELECT

   customer_id,

   COUNT(order_id) AS order_count,

   SUM(total_amount) AS total_spent

FROM orders

GROUP BY customer_id

HAVING COUNT(order_id) >= 2 AND SUM(total_amount) >= 100;
```



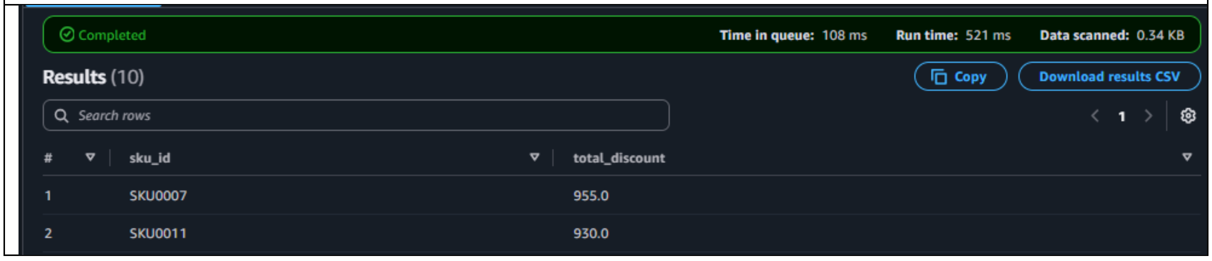
Completed Time in queue: 184 ms Run time: 605 ms Data scanned: 10.10 KB

Results (100)

#	customer_id	order_count	total_spent
1	24	15	909.29
2	32	11	385.08
3	93	12	395.99

8. Most Discounted Products by Total Discount Given

```
SELECT
   sku_id,
   SUM(discount_amount) AS total_discount
FROM order_items
GROUP BY sku_id
ORDER BY total_discount DESC
LIMIT 10;
```



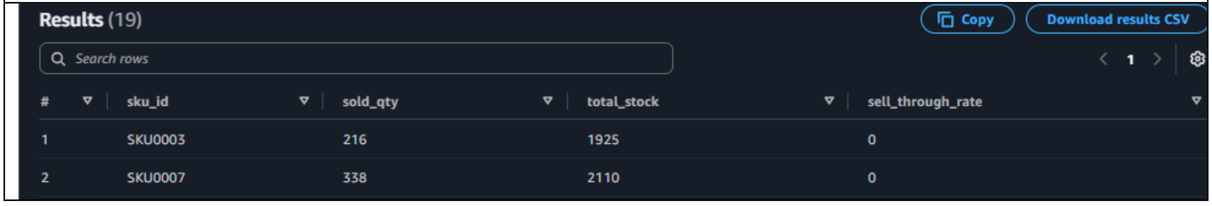
Completed Time in queue: 108 ms Run time: 521 ms Data scanned: 0.34 KB

Results (10)

#	sku_id	total_discount
1	SKU0007	955.0
2	SKU0011	930.0

9. SKU Sell-Through Rate (Sold Quantity vs. Inventory)

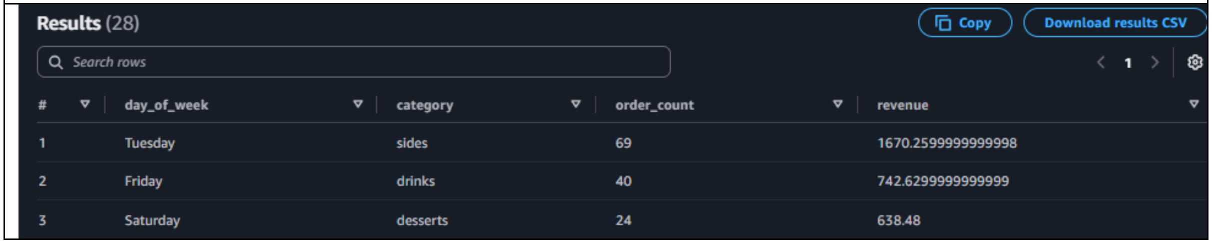
```
WITH sold AS (
  SELECT sku_id, SUM(quantity) AS sold_qty
  FROM order_items
  GROUP BY sku_id
), stock AS (
  SELECT sku_id, SUM(stock_qty) AS total_stock
  FROM inventory_logs
  GROUP BY sku_id
)
SELECT
   s.sku_id,
   sold_qty,
   total_stock,
   ROUND(sold_qty / (sold_qty + total_stock) * 100, 2) AS
sell_through_rate
FROM sold s
JOIN stock i ON s.sku_id = i.sku_id;
```


Results (19)

#	sku_id	sold_qty	total_stock	sell_through_rate
1	SKU0003	216	1925	0
2	SKU0007	338	2110	0

10. Order Trends with Day of Week and Category

```
SELECT

   format_datetime(order_time, 'EEEE') AS day_of_week,

   sm.category,

   COUNT(DISTINCT o.order_id) AS order_count,

   SUM(oi.quantity * oi.unit_price - oi.discount_amount) AS
revenue

FROM orders o

JOIN order_items oi ON o.order_id = oi.order_id

JOIN sku_master sm ON oi.sku_id = sm.sku_id

GROUP BY format_datetime(order_time, 'EEEE'), sm.category;
```

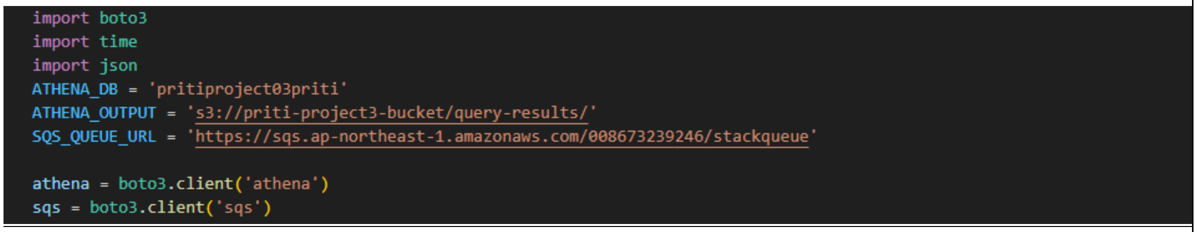


Results (28)

#	day_of_week	category	order_count	revenue
1	Tuesday	sides	69	1670.2599999999998
2	Friday	drinks	40	742.6299999999999
3	Saturday	desserts	24	638.48

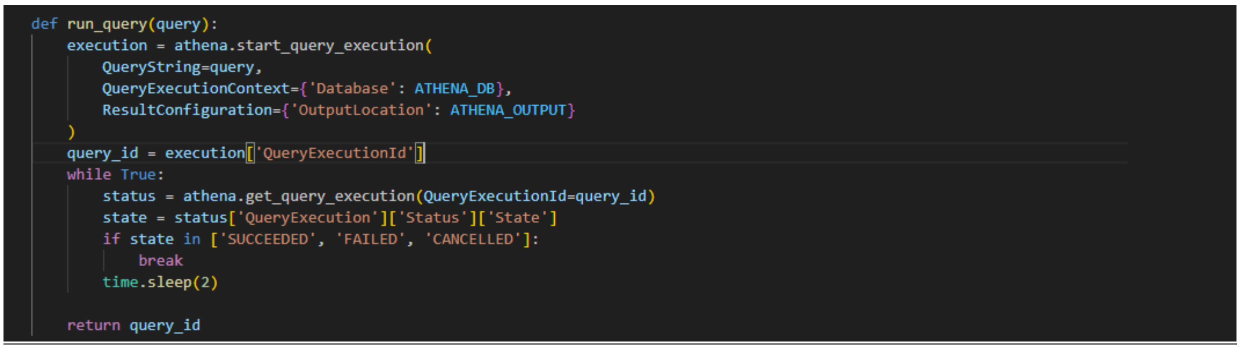
Setting up lambda function in order to send the details to SQS when a query is satisfied

- We will be creating a Lambda function which will run a query on Athena tables and if the query executes and returns some response successfully then the data will be uploaded to SQS for further operation .
- For this we do need to create a function and it should have the permission to access athena and SQS .



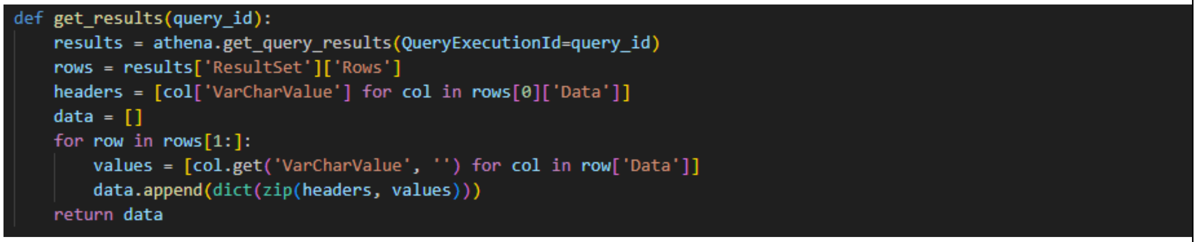
```
import boto3
import time
import json
ATHENA_DB = 'pritiproject03priti'
ATHENA_OUTPUT = 's3://priti-project3-bucket/query-results/'
SQS_QUEUE_URL = 'https://sqs.ap-northeast-1.amazonaws.com/008673239246/stackqueue'

athena = boto3.client('athena')
sqs = boto3.client('sqs')
```

- In this part of code we are importing required libraries such as boto3 , time and json .
- Along with libraries I am explicitly mentioning the Athena database name where my tables are present , the S3 bucket where the query output will be stored , and the SQS queue url into which the data is to be sent .
- In this code we are also creating two different boto3 clients one for athena and other for sqs using boto3.client('…').



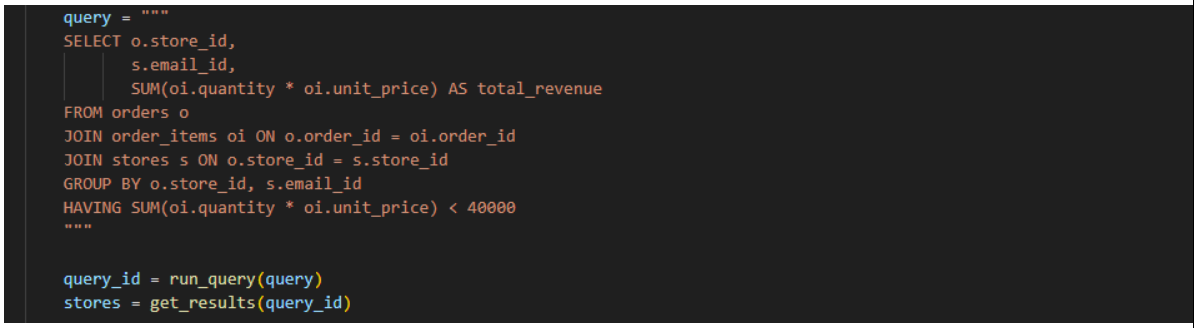
```python
def run_query(query):
    execution = athena.start_query_execution(
        QueryString=query,
        QueryExecutionContext={'Database': ATHENA_DB},
        ResultConfiguration={'OutputLocation': ATHENA_OUTPUT}
    )
    query_id = execution['QueryExecutionId']
    while True:
        status = athena.get_query_execution(QueryExecutionId=query_id)
        state = status['QueryExecution']['Status']['State']
        if state in ['SUCCEEDED', 'FAILED', 'CANCELLED']:
            break
        time.sleep(2)

    return query_id
```

- The function run_query(query) takes a SQL query as input .It starts execution of that query in AWS Athena using start_query_execution.The Athena database to run the query on is specified by ATHENA_DB.
- The query result is configured to be saved to an S3 bucket specified by ATHENA_OUTPUT. After starting the query, it receives a QueryExecutionId which uniquely identifies this query.
- A loop begins to continuously check the status of the query .The query status is checked using get_query_execution and the current state is read.
- The loop keeps waiting (with a 2-second pause each time) until the state becomes either SUCCEEDED, FAILED, or CANCELLED. Once the query finishes (either successfully or not), the loop breaks.
- The function returns the QueryExecutionId at the end.
- This function will be called within the lamda handler function with the query that is needed to be checked as a parameter passed to this function



```
def get_results(query_id):
    results = athena.get_query_results(QueryExecutionId=query_id)
    rows = results['ResultSet']['Rows']
    headers = [col['VarCharValue'] for col in rows[0]['Data']]
    data = []
    for row in rows[1:]:
        values = [col.get('VarCharValue', '') for col in row['Data']]
        data.append(dict(zip(headers, values)))
    return data
```

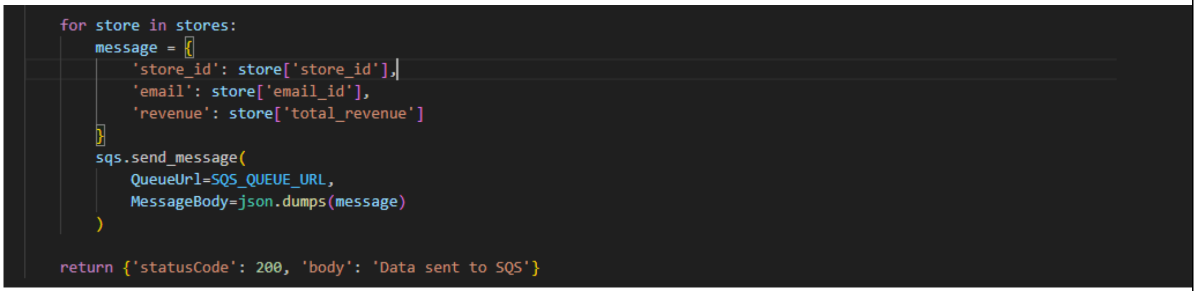
- The function get_results(query_id) takes a query_id (from Athena) as input.It fetches the result of the Athena query using get_query_results.
- The result contains all rows in results['ResultSet']['Rows'] . The first row (rows[0]) contains the column headers.It extracts the column names (headers) from that first row.
- It creates an empty list data to store final results.It loops over each remaining row (excluding the header row).For each row, it extracts column values using .get('VarCharValue', '').
- It creates a dictionary by zipping headers with the corresponding values.It adds each row dictionary to the data list.
- Finally, it returns the data list containing all rows as dictionaries.



```
query = """
SELECT o.store_id,
       s.email_id,
       SUM(oi.quantity * oi.unit_price) AS total_revenue
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN stores s ON o.store_id = s.store_id
GROUP BY o.store_id, s.email_id
HAVING SUM(oi.quantity * oi.unit_price) < 40000
"""

query_id = run_query(query)
stores = get_results(query_id)
```

- A SQL query is written as a string and assigned to the variable query.
- The query selects the following:

    - store_id from the orders table.
    - email_id from the stores table.
    - Total revenue, calculated as the sum of quantity * unit_price from the order_items table.
- The query joins:
    - orders (alias o) with order_items (alias oi) on order_id.
    - orders with stores (alias s) on store_id.
- It groups the results by store_id and email_id.
- It filters the grouped results using HAVING to keep only those stores where total revenue is less than 40,000.
- run_query(query) executes the SQL query on Athena and returns the query_id.
- get_results(query_id) fetches and parses the result using that query_id.
- The final result is stored in the variable stores as a list of dictionaries.



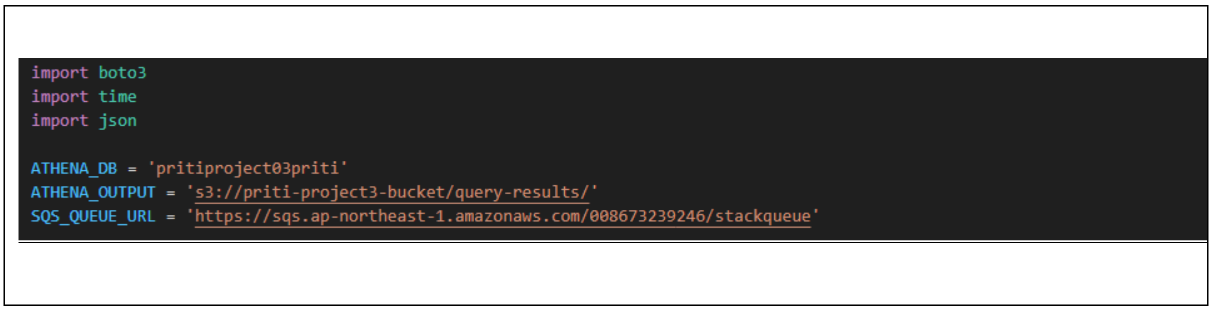
```
for store in stores:
    message = {
        'store_id': store['store_id'],
        'email': store['email_id'],
        'revenue': store['total_revenue']
    }
    sqs.send_message(
        QueueUrl=SQS_QUEUE_URL,
        MessageBody=json.dumps(message)
    )

return {'statusCode': 200, 'body': 'Data sent to SQS'}
```

- It starts a loop over each store in the stores list (which contains results from the Athena query).
- For each store, it creates a dictionary called message containing:
    - 'store_id': the store's ID from the query result.
    - 'email': the email ID of the store.
    - 'revenue': the total revenue calculated for that store.

- It sends this message to an AWS SQS queue using sqs.send_message(). The QueueUrl specifies which SQS queue to send the message to, using the SQS_QUEUE_URL.
- The MessageBody is the JSON string version of the message dictionary.
- After all messages are sent, the function returns a response:
    - statusCode: 200 to indicate success.
    - body: 'Data sent to SQS' as a confirmation message.

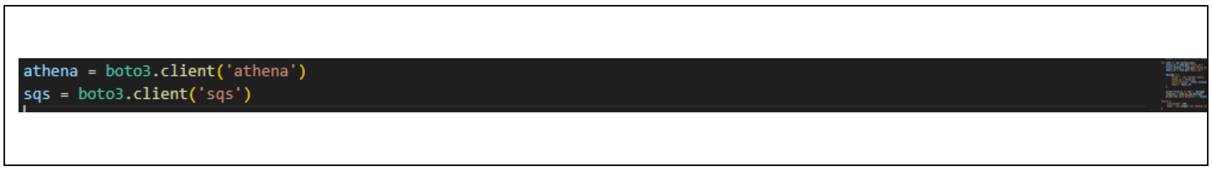
Setting up EC2 in order to pool data from SQS and send it to respective mail using SNS

- Setting up and EC2 instance in order to pool messages or data from SQS and send it to the respective mail queried out from the RDS with respect to the data stored in the queue using SNS.
- We will be writing a python script and execute it to check if it is working fine .



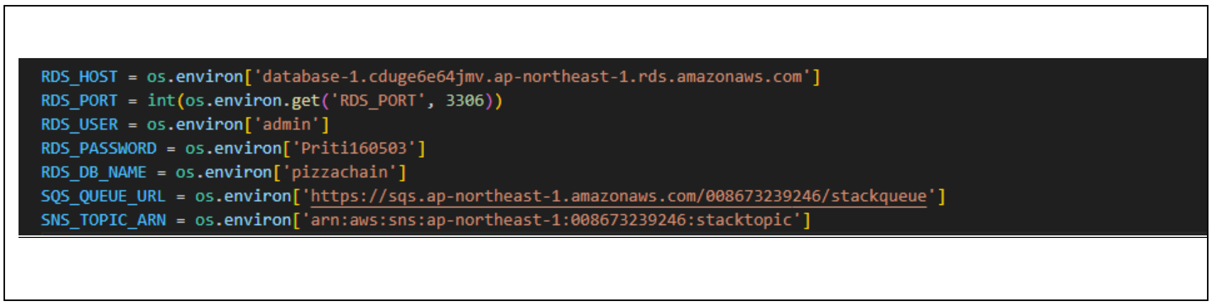
```
import boto3
import time
import json

ATHENA_DB = 'pritiproject03priti'
ATHENA_OUTPUT = 's3://priti-project3-bucket/query-results/'
SQS_QUEUE_URL = 'https://sqs.ap-northeast-1.amazonaws.com/008673239246/stackqueue'
```

- This part of code imports the required libraries that are required for execution
- In this we also do specify the athena db url where the data will be uploaded , The S3 bucket mentioned will store the outputs in that bucket only .



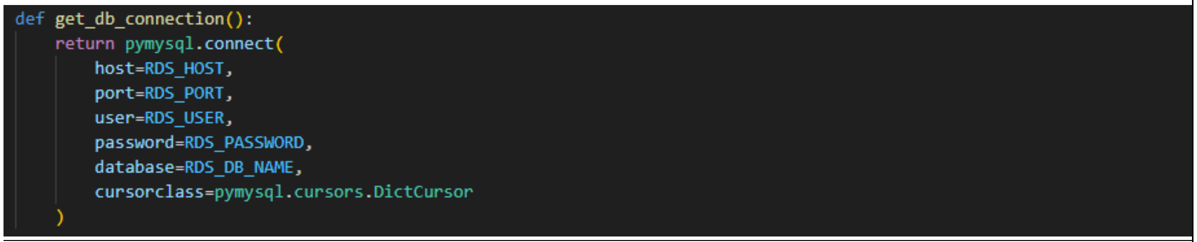
```
athena = boto3.client('athena')
sqs = boto3.client('sqs')
```

- This code basically specifies and create 2 boto3 clients one for athena and other for sqs .



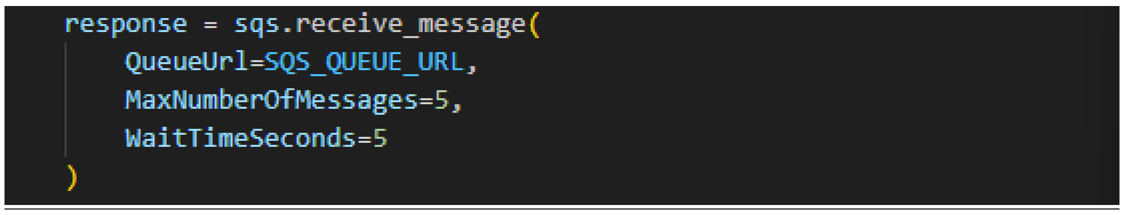
```
RDS_HOST = os.environ['database-1.cduge6e64jmv.ap-northeast-1.rds.amazonaws.com']
RDS_PORT = int(os.environ.get('RDS_PORT', 3306))
RDS_USER = os.environ['admin']
RDS_PASSWORD = os.environ['Priti160503']
RDS_DB_NAME = os.environ['pizzachain']
SQS_QUEUE_URL = os.environ['https://sqs.ap-northeast-1.amazonaws.com/008673239246/stackqueue']
SNS_TOPIC_ARN = os.environ['arn:aws:sns:ap-northeast-1:008673239246:stacktopic']
```

- In this code we are providing the credentials of the RDS along with SQS and SNS .
- These credentials are used to connect to RDS for lookup for the email along with pooling data from SQS using BOTO3 and sending mail to the extracted mail using SNS .



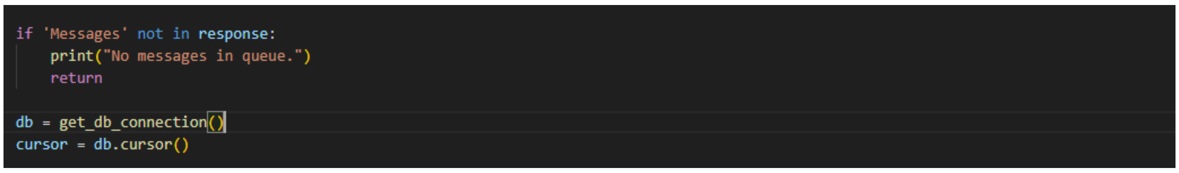
```
def get_db_connection():
    return pymysql.connect(
        host=RDS_HOST,
        port=RDS_PORT,
        user=RDS_USER,
        password=RDS_PASSWORD,
        database=RDS_DB_NAME,
        cursorclass=pymysql.cursors.DictCursor
    )
```

- In this we are setiing up connection with RDS using PyMysql and the credentials provided earlies such as endpoint,port number , user name , password , database name .
- Using **pymysql.cursors.DictCursor** we are telling PyMysql to return the results of the SQL queries as dictionaries rather than as default tuples.
- A **cursor** in database programming is an object that:
  - Executes SQL queries,
  - Retrieves results from the database,
  - Iterates over rows returned by a query.
  - It's like a pointer that helps you interact with the database result set.
- It will overall connect to the RDS database mentioned to it in the paramas using the above credentials and also while working on the database it will return the results in the Dictonary format rather than the general tuple format .
- Using a dictionary lets us access data by **column name**, which is **easier to understand** than using index positions.



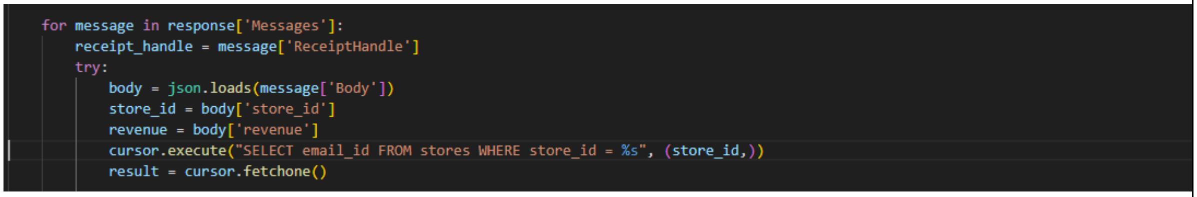
```python
response = sqs.receive_message(
    QueueUrl=SQS_QUEUE_URL,
    MaxNumberOfMessages=5,
    WaitTimeSeconds=5
)
```

- In this code we are pooling the SQS for the message stored in it in order to access the store id we are passing various parameters to it .
- **QueueUrl :** it contains the url of the SQS queue which is being used by us in order to let the sqs client know which queue is to be accessed .
- **MaxNumberOfMessages :** this gives the idea of how many messages is to be pooled from the Queue here in this case a maximum of 5 messages are allowed .
- **WaitTimeSecond:** how long to wait during this request for messages to appear before giving up and returning an empty response.
- If a message arrives within those 5 seconds, it is returned immediately.
- If no message arrives during the 5-second window, SQS will return an empty response.
- This is called lazy pooling
- If we don't provide the WaitTimeSeconds the API just checks once and immediately returns, even if the queue is empty .
- This is called short polling .
- Basically we can say that WaitTimeSeconds tells SQS to "wait this long for a message to arrive before giving up and returning nothing."



```
if 'Messages' not in response:
    print("No messages in queue.")
    return

db = get_db_connection()
cursor = db.cursor()
```

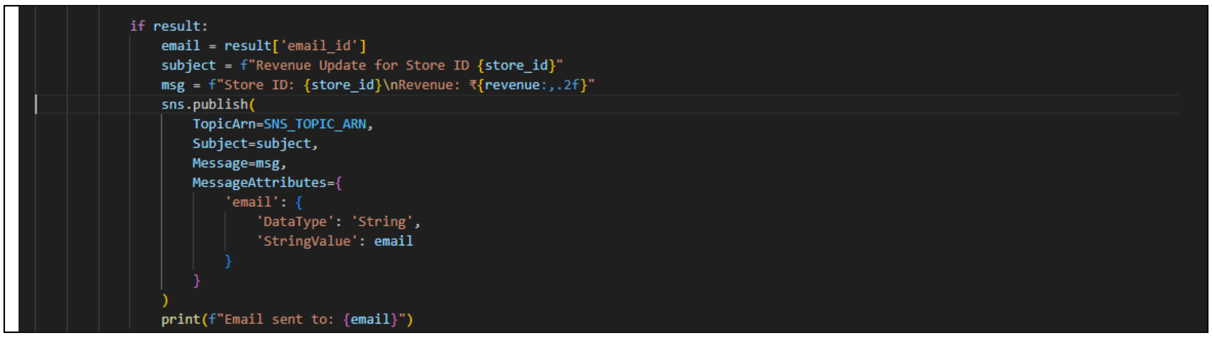
- Thid code checks if the SQS response contains any messages.If no messages are found in the response, it prints "No messages in queue." to the logs.
- It then uses return to stop further execution of the function early since there’s nothing to process.
- If there **are** messages, it proceeds to connect to the RDS database by calling get_db_connection().
- This function returns a connection object, which is assigned to the variable db.
- Then, cursor = db.cursor() creates a cursor object from the database connection.
- The cursor is used to execute SQL queries and retrieve results from the database .



```
for message in response['Messages']:
    receipt_handle = message['ReceiptHandle']
    try:
        body = json.loads(message['Body'])
        store_id = body['store_id']
        revenue = body['revenue']
        cursor.execute("SELECT email_id FROM stores WHERE store_id = %s", (store_id,))
        result = cursor.fetchone()
```

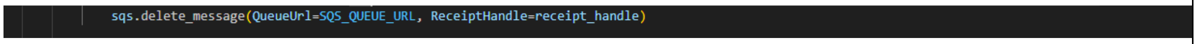
- The loop goes through each message received from the SQS queue and for each message, it extracts the ReceiptHandle which is used later to delete the message from the queue.
- The message body is in JSON string format, so json.loads(message[‘Body’]) is used to convert it into a Python dictionary.

- From the parsed message body, it extracts the value of store_id and assigns it to the variable store_id.
- It also extracts the revenue value and stores it in the variable revenue.
- A SQL query is executed using the cursor to select the email_id from the stores table where the store_id matches the one from the message.
- The cursor.fetchone() call fetches the first result (if any) of that query and stores it in the variable result.



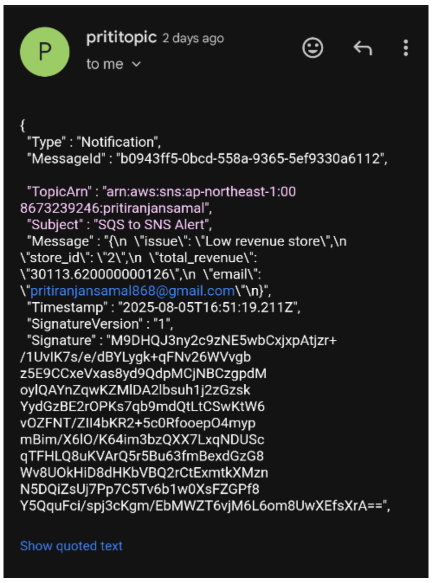
```
if result:
    email = result['email_id']
    subject = f"Revenue Update for Store ID {store_id}"
    msg = f"Store ID: {store_id}\nRevenue: ₹{revenue:,.2f}"
    sns.publish(
        TopicArn=SNS_TOPIC_ARN,
        Subject=subject,
        Message=msg,
        MessageAttributes={
            'email': {
                'DataType': 'String',
                'StringValue': email
            }
        }
    )
    print(f"Email sent to: {email}")
```

- The if result: condition checks if the query to the database returned a result if a result is found, it extracts the email_id from the result and assigns it to the variable email.
- A subject line is created for the email using an f-string that includes the store_id. Another f-string is used to build the message body msg, which includes the store_id and revenue formatted with commas and two decimal places.
- The sns.publish() function sends this message to the SNS topic defined by SNS_TOPIC_ARN.
- A confirmation is printed to the log saying the email was sent to the retrieved email address.
- If no result was found from the database (i.e., no matching store_id), it prints a message saying no email was found for that store ID.



```
        sqs.delete_message(QueueUrl=SQS_QUEUE_URL, ReceiptHandle=receipt_handle)
```

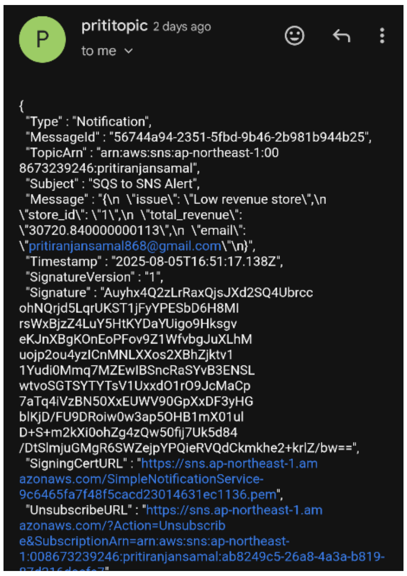
- Once the message is retrieved from the queue the message is needed to be clear in order to prevent multiple processing the same data .
- Once everything is done we close the database connection .
- Once everything is done we will have to check if the mail is correctly sent to the desired mail or not .



pritittopic 2 days ago
to me

```
{
  "Type" : "Notification",
  "MessageId" : "b0943ff5-0bcd-558a-9365-5ef9330a6112",

  "TopicArn" : "arn:aws:sns:ap-northeast-1:00
8673239246:pritiranjansamal",
  "Subject" : "SQS to SNS Alert",
  "Message" : "{\n  \"issue\": \"Low revenue store\",\n
\"store_id\": \"2\",\n  \"total_revenue\":
\"30113.620000000126\",\n  \"email\":
\"pritiranjansamal868@gmail.com\"\n}",
  "Timestamp" : "2025-08-05T16:51:19.211Z",
  "SignatureVersion" : "1",
  "Signature" : "M9DHQJ3ny2c9zNE5wbCxjxpAtjzr+
/1UvIK7s/e/dBYLygk+qFNv26WVvgb
z5E9CCxeVxas8yd9QdpMCjNBCzgpdM
oylQAYnZqwKZMlDA2lbsuh1j2zGzsk
YydGzBE2rOPKs7qb9mdQtLtCSwKtW6
vOZFNT/ZIl4bKR2+5c0RfooepO4myp
mBim/X6lO/K64im3bzQXX7LxqNDUSc
qTFHLQ8uKVArQ5r5Bu63fmBexdGzG8
Wv8UOkHiD8dHKbVBQ2rCtExmtkXMzn
N5DQiZsUj7Pp7C5Tv6b1w0XsFZGPf8
Y5QquFci/spj3cKgm/EbMWZT6vjM6L6om8UwXEfsXrA==",
```

Show quoted text



pritittopic 2 days ago
to me

```
{
  "Type" : "Notification",
  "MessageId" : "56744a94-2351-5fbd-9b46-2b981b944b25",
  "TopicArn" : "arn:aws:sns:ap-northeast-1:00
8673239246:pritiranjansamal",
  "Subject" : "SQS to SNS Alert",
  "Message" : "{\n  \"issue\": \"Low revenue store\",\n
\"store_id\": \"1\",\n  \"total_revenue\":
\"30720.840000000113\",\n  \"email\":
\"pritiranjansamal868@gmail.com\"\n}",
  "Timestamp" : "2025-08-05T16:51:17.138Z",
  "SignatureVersion" : "1",
  "Signature" : "Auyhx4Q2zLrRaxQjsJXd2SQ4Ubrcc
ohNQrjd5LqrUKST1jFyYPESbD6H8MI
rsWxBjzZ4LuY5HtKYDaYUigo9Hksgv
eKJnXBgKOnEoPFov9Z1WfvbgJuXLhM
uojp2ou4yzICnMNLXXos2XBhZjktv1
1Yudi0Mmq7MZEwIBSncRaSYvB3ENSL
wtvoSGTSYTYTsV1UxxdO1rO9JcMaCp
7aTq4iVzBN50XxEUWV90GpXxDF3yHG
blKjD/FU9DRoiw0w3ap5OHB1mX01ul
D+S+m2kXi0ohZg4zQw50fij7Uk5d84
/DtSlmjuGMgR6SWZejpYPQieRVQdCkmkhe2+krlZ/bw==",
  "SigningCertURL" : "https://sns.ap-northeast-1.am
azonaws.com/SimpleNotificationService-
9c6465fa7f48f5cacd23014631ec1136.pem",
  "UnsubscribeURL" : "https://sns.ap-northeast-1.am
azonaws.com/?Action=Unsubscrib
e&SubscriptionArn=arn:aws:sns:ap-northeast-
1:008673239246:pritiranjansamal:ab8249c5-26a8-4a3a-b819-
```

- As can be seen the message is successfully delivered to the given registered mail .
- We have to keep in note that the email should be subscribed prior sending the message .