# Project Title: Energy Consumption Metering (Time-series)

“An end-to-end batch data pipeline for smart electricity meter data, designed to provide billing insights, detect consumption anomalies, and optimize query performance using AWS services.”

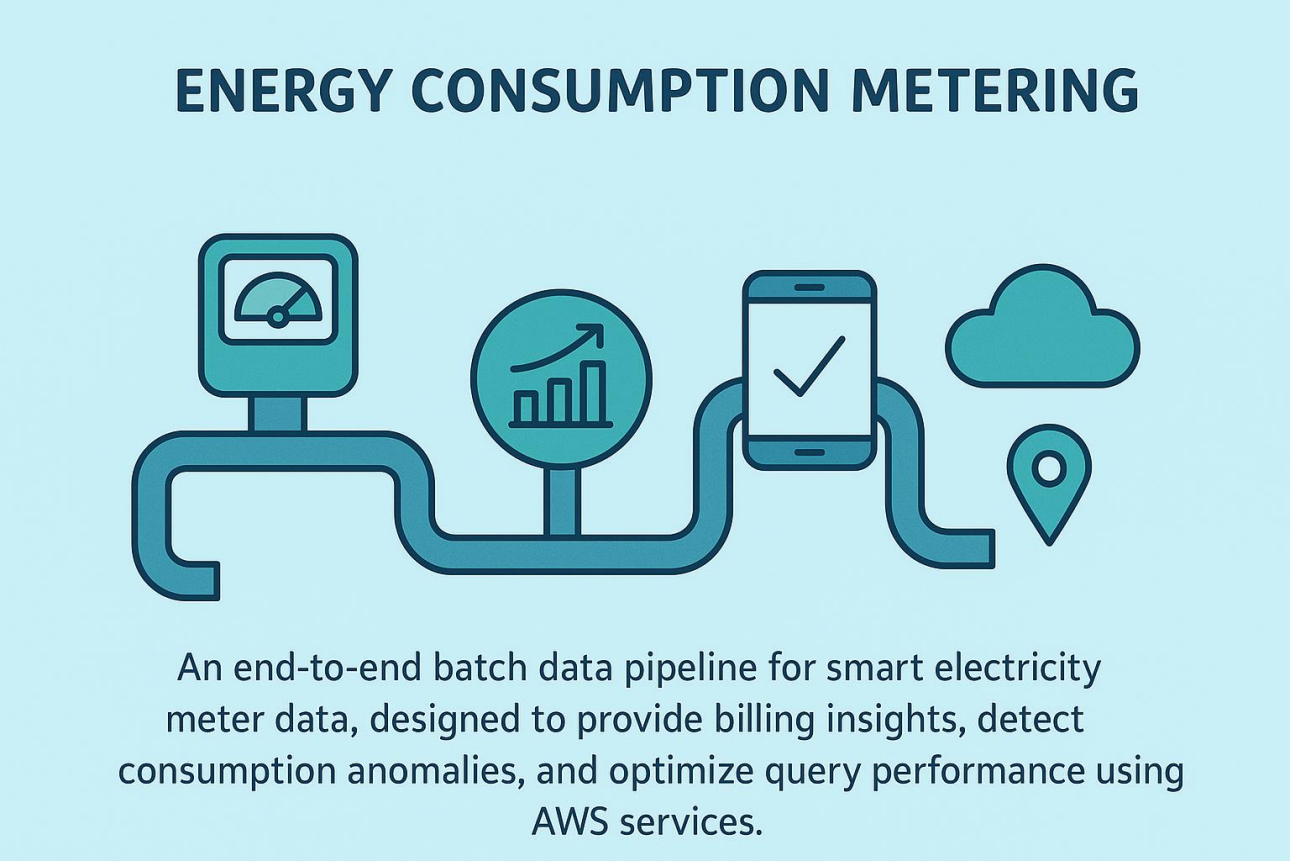
**Name:** Yashwanth Aravanti

**Batch:** AWS Data Engineering(2025)

**Scenario:** End-to-end pipeline for collecting meter readings → performing nightly ETL processing → generating billing summaries and anomaly alerts.

**Pipeline Type:** Batch ETL with data partitioning for cost and performance optimization.

**Tools/Services:** S3, Databricks, Amazon Athena, SNS, AWS Lambda, API Gateway, Amazon QuickSight.

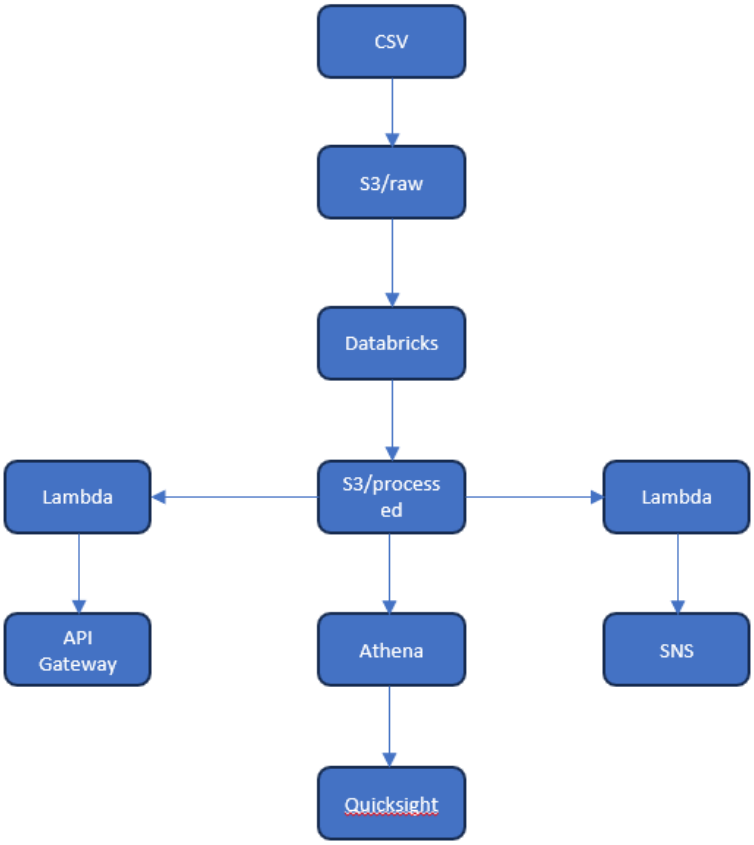


**Project Goal:** The project aims to build an automated, serverless data pipeline that ingests CSV-based energy data, processes it using Databricks, and serves it through multiple endpoints: an interactive BI dashboard, an API, and an automated alerting system.

**Project Overview:**

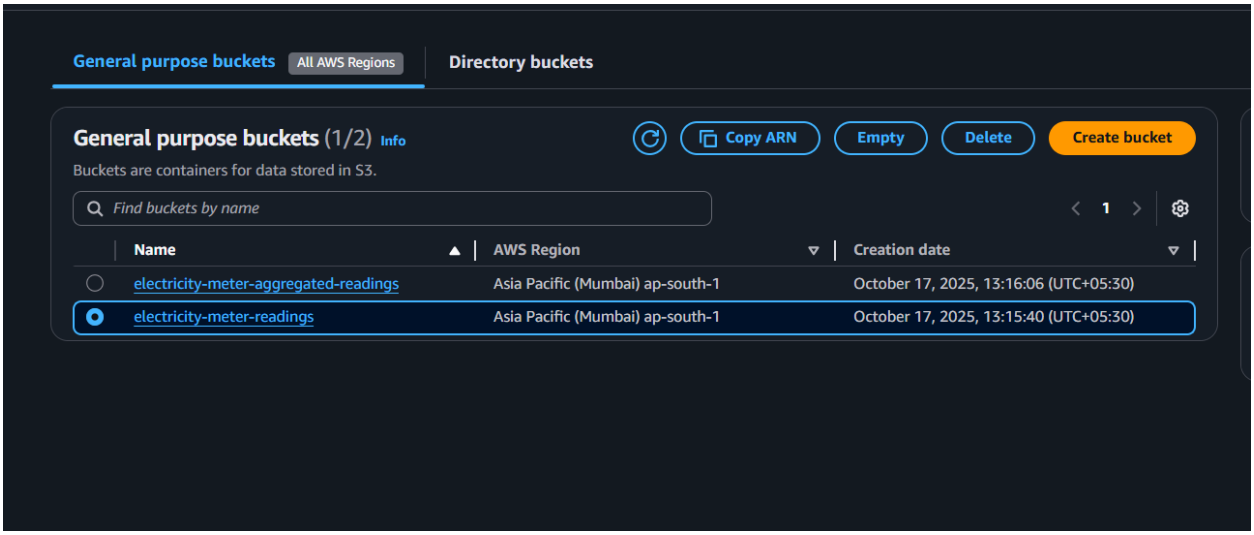
1. **Data Ingestion**
    - Source data is provided as **CSV files**, which are uploaded to a dedicated raw data landing zone in an **S3 bucket** (e.g., s3://your-bucket/raw/).
2. **ETL with Databricks**
    - A **Databricks** job is triggered to read new CSV files from the raw S3 path.
    - The job performs all cleaning, transformation, aggregation, and enrichment logic, writing the final, query-ready data (as Parquet) to a separate **processed S3 path** (e.g., s3://your-bucket/processed/).
3. **Interactive Analytics & Dashboards**
    - **Amazon Athena** is used as a serverless query engine to directly read the processed Parquet data from the processed S3 bucket.
    - **Amazon Quicksight** connects to Athena, providing a rich, interactive dashboard for business users to analyze consumption trends and patterns.
4. **Automated Alerting**
    - An **AWS Lambda** function is configured to analyze the data in the processed S3 bucket after the Databricks job runs.
    - If any anomalies (e.g., consumption spikes) are detected, the Lambda function sends a notification to an **Amazon SNS** topic, which then distributes the alert to subscribers.
5. **API Data Access**
    - An **API Gateway** provides a RESTful endpoint for applications to programmatically access the energy data.
    - Requests to the API Gateway trigger a dedicated **AWS Lambda** function, which fetches the required data from the processed S3 bucket and returns it to the client.

**Architectural Diagram:**



**Step 1: Create an S3 Bucket (Raw Data Storage)**

1. In AWS Console → Search “S3” → Open it.
2. Click “Create bucket.”
3. Give it a name, e.g., electricity-meter-readings
4. Choose region: ap-south-1 (Mumbai).
5. Keep defaults → Click Create bucket.




**Purpose:** This S3 bucket serves as the durable, scalable, and cost-effective **landing zone** for all incoming raw data before any processing occurs.

**Step 2: Create and Upload Readings.csv to S3**

**1. About Dataset**: This schema is from the well-known "Individual household electric power consumption" dataset. CSV file represents a significant batch of historical data, containing 246,033 rows of measurements. These readings are captured every minute, spanning the period from December 16, 2006, to June 5, 2007.

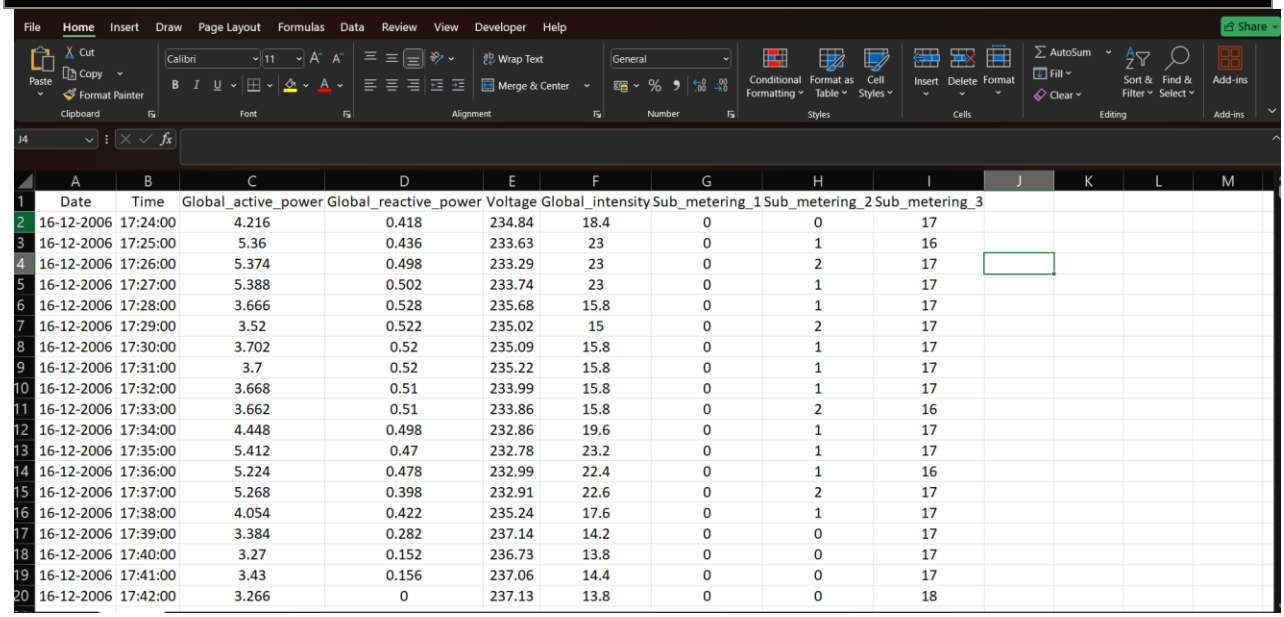
**2. Schema Explanation**:Each column in your readings.csv file represents a specific measurement from a single household's electricity meter:

- Date & Time: The exact date and time of the measurement.
- Global_active_power (kilowatt): This is the most important value—the real power that the household is consuming. 💡
- Global_reactive_power (kilowatt): This is the portion of power that is not consumed and is returned to the electrical grid.
- Voltage (volt): The average voltage being supplied to the household.
- Global_intensity (ampere): The average current being drawn by the household.

- Sub_metering_1 (watt-hour): Energy consumption from a specific circuit, typically the kitchen, including appliances like the microwave and dishwasher.
- Sub_metering_2 (watt-hour): Energy consumption from another circuit, often the laundry room (washing machine, dryer).
- Sub_metering_3 (watt-hour): Energy consumption from a third circuit, usually an electric water heater and/or an air-conditioner.

**3. Data Sample CSV file (readings.csv) will be formatted exactly like this snippet:**

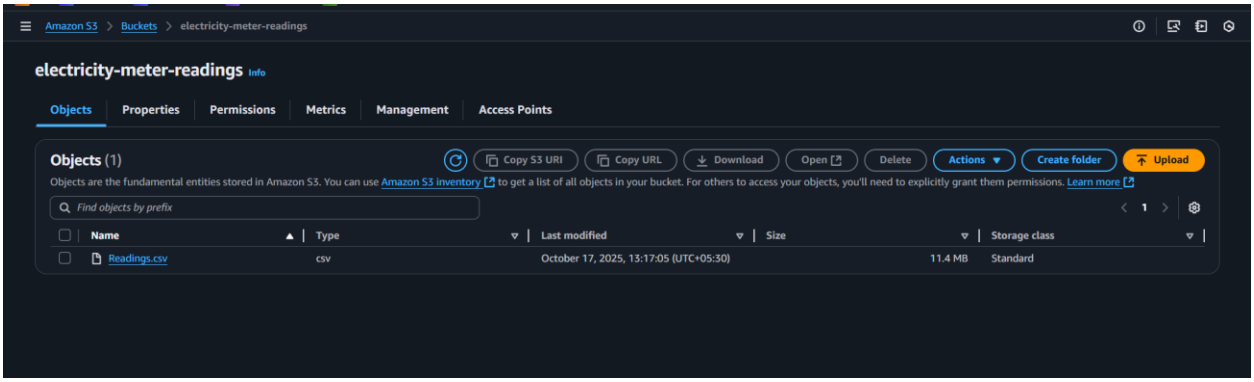
```
Date,Time,Global_active_power,Global_reactive_power,Voltage,Global_intensity,Sub_metering_1,Sub_metering_2,Sub_metering_3
16-12-2006,17:24:00,4.216,0.418,234.84,18.4,0,1,17
16-12-2006,17:25:00,5.36,0.436,233.63,23,0,1,16
16-12-2006,17:26:00,5.374,0.498,233.29,23,0,2,17
...
```



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Date | Time | Global_active_power | Global_reactive_power | Voltage | Global_intensity | Sub_metering_1 | Sub_metering_2 | Sub_metering_3 |
| 2 | 16-12-2006 | 17:24:00 | 4.216 | 0.418 | 234.84 | 18.4 | 0 | 0 | 17 |
| 3 | 16-12-2006 | 17:25:00 | 5.36 | 0.436 | 233.63 | 23 | 0 | 1 | 16 |
| 4 | 16-12-2006 | 17:26:00 | 5.374 | 0.498 | 233.29 | 23 | 0 | 2 | 17 |
| 5 | 16-12-2006 | 17:27:00 | 5.388 | 0.502 | 233.74 | 23 | 0 | 1 | 17 |
| 6 | 16-12-2006 | 17:28:00 | 3.666 | 0.528 | 235.68 | 15.8 | 0 | 1 | 17 |
| 7 | 16-12-2006 | 17:29:00 | 3.52 | 0.522 | 235.02 | 15 | 0 | 2 | 17 |
| 8 | 16-12-2006 | 17:30:00 | 3.702 | 0.52 | 235.09 | 15.8 | 0 | 1 | 17 |
| 9 | 16-12-2006 | 17:31:00 | 3.7 | 0.52 | 235.22 | 15.8 | 0 | 1 | 17 |
| 10 | 16-12-2006 | 17:32:00 | 3.668 | 0.51 | 233.99 | 15.8 | 0 | 1 | 17 |
| 11 | 16-12-2006 | 17:33:00 | 3.662 | 0.51 | 233.86 | 15.8 | 0 | 2 | 16 |
| 12 | 16-12-2006 | 17:34:00 | 4.448 | 0.498 | 232.86 | 19.6 | 0 | 1 | 17 |
| 13 | 16-12-2006 | 17:35:00 | 5.412 | 0.47 | 232.78 | 23.2 | 0 | 1 | 17 |
| 14 | 16-12-2006 | 17:36:00 | 5.224 | 0.478 | 232.99 | 22.4 | 0 | 1 | 16 |
| 15 | 16-12-2006 | 17:37:00 | 5.268 | 0.398 | 232.91 | 22.6 | 0 | 2 | 17 |
| 16 | 16-12-2006 | 17:38:00 | 4.054 | 0.422 | 235.24 | 17.6 | 0 | 1 | 17 |
| 17 | 16-12-2006 | 17:39:00 | 3.384 | 0.282 | 237.14 | 14.2 | 0 | 0 | 17 |
| 18 | 16-12-2006 | 17:40:00 | 3.27 | 0.152 | 236.73 | 13.8 | 0 | 0 | 17 |
| 19 | 16-12-2006 | 17:41:00 | 3.43 | 0.156 | 237.06 | 14.4 | 0 | 0 | 17 |
| 20 | 16-12-2006 | 17:42:00 | 3.266 | 0 | 237.13 | 13.8 | 0 | 0 | 18 |

**4. How to Upload the File**

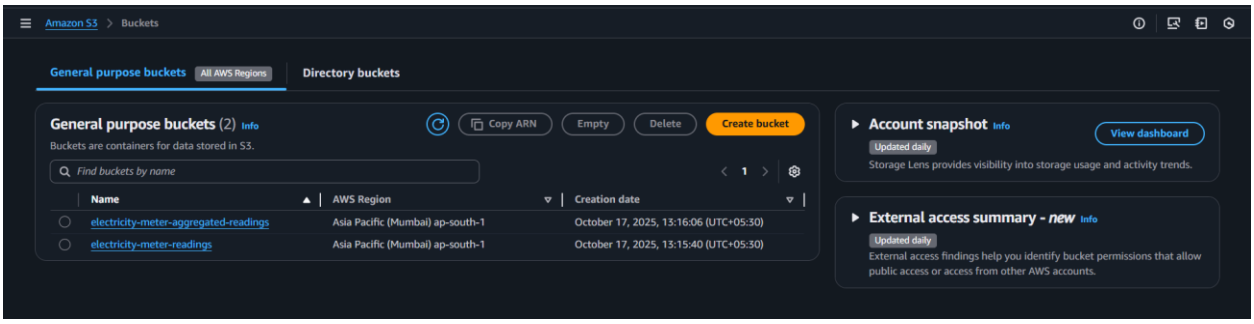
1. Navigate to the S3 bucket you created (e.g., electricity-meter-readings).
2. Click the **"Upload"** button.
3. Click **"Add files,"** select your readings.csv file, and then click **"Upload"** at the bottom of the page.




**Purpose:** This action simulates the arrival of a large, real-world data batch, which acts as the **trigger** for your entire data processing pipeline.

**Step 3: Create S3 Bucket for Processed Data**

1. In the AWS Console, navigate back to the S3 service dashboard.
2. Click “Create bucket.”
3. Give it a globally unique name that signifies its purpose, e.g., electricity-meter-aggregated-readings-[yourname]
4. Choose the same region as your raw bucket for consistency: ap-south-1 (Mumbai).
5. Keep all other defaults and click Create bucket.




**Purpose:** This S3 bucket serves as the curated or processed layer of your data lake. It will store the clean, structured, and aggregated data output by your Databricks job, making it ready for analysis and consumption.

**Step 4: Set Up Databricks & Develop the ETL Notebook**

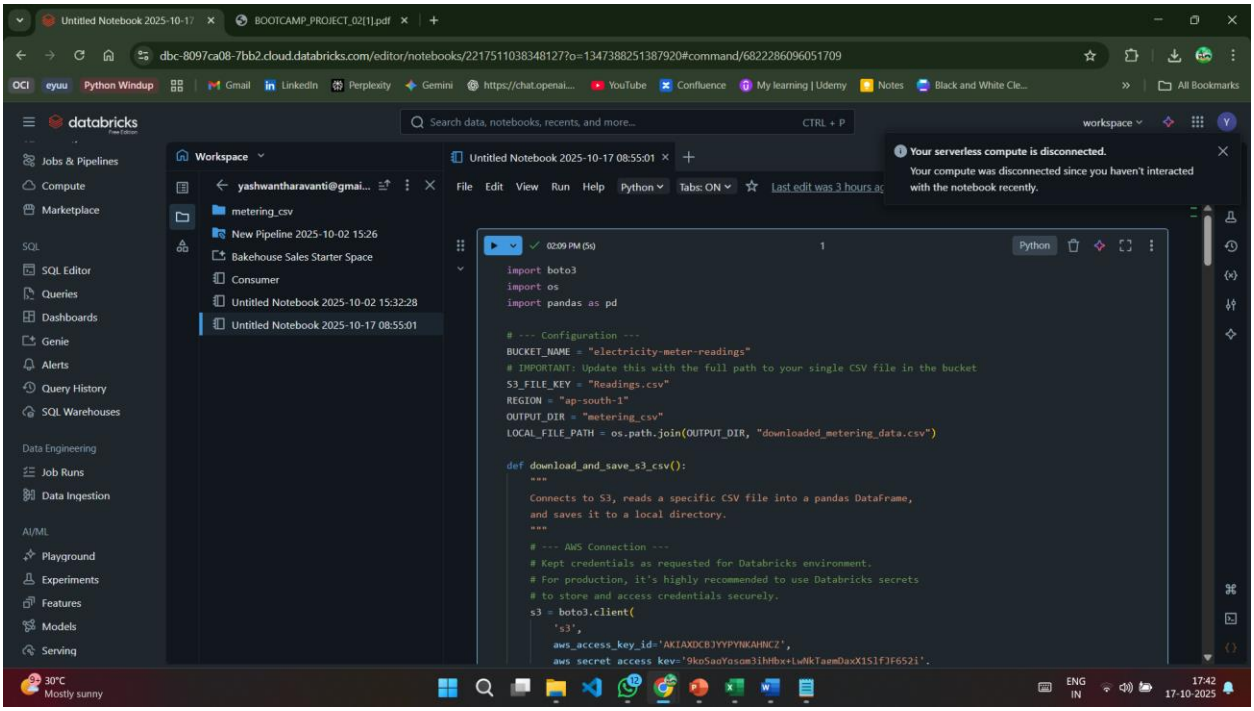
1. **Create the Notebook:**
   - In the left-hand menu, click Workspace.
   - Click the dropdown arrow next to your name/workspace, and choose Create > Notebook.
   - Name your notebook Nightly-ETL-Process.
   - Ensure the default language is set to Python**.**
2. **Developing the ETL Logic in the Notebook**

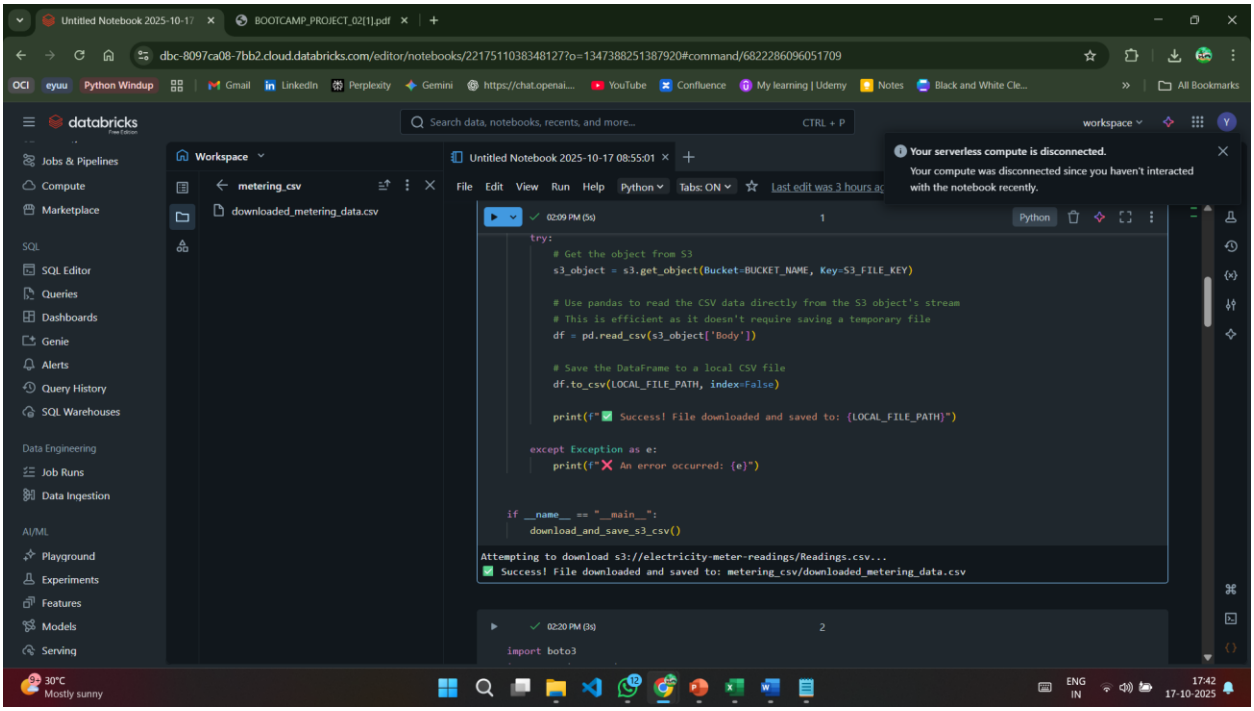
   Here is the PySpark code, broken down into logical cells, that you will add to your Nightly-ETL-Process notebook.

**Cell 1:** Define Variables and Read Raw Data from S3.This cell reads all the CSV files from your raw data bucket.

```
import boto3
import os
import pandas as pd

# --- Configuration ---
BUCKET_NAME = "electricity-meter-readings"
# IMPORTANT: Update this with the full path to your single CSV file in the bucket
S3_FILE_KEY = "Readings.csv"
REGION = "ap-south-1"
OUTPUT_DIR = "metering_csv"
LOCAL_FILE_PATH = os.path.join(OUTPUT_DIR, "downloaded_metering_data.csv")

def download_and_save_s3_csv():
    """
    Connects to S3, reads a specific CSV file into a pandas DataFrame,
    and saves it to a local directory.
    """
    # --- AWS Connection ---
    # Kept credentials as requested for Databricks environment.
    # For production, it's highly recommended to use Databricks secrets
    # to store and access credentials securely.
    s3 = boto3.client(
        's3',
        aws_access_key_id='',
        aws_secret_access_key='',
        region_name=REGION
    )

    # Ensure the output directory exists
    os.makedirs(OUTPUT_DIR, exist_ok=True)

    print(f"Attempting to download s3://{BUCKET_NAME}/{S3_FILE_KEY}...")

    try:
        # Get the object from S3
        s3_object = s3.get_object(Bucket=BUCKET_NAME, Key=S3_FILE_KEY)

        # Use pandas to read the CSV data directly from the S3 object's stream
        # This is efficient as it doesn't require saving a temporary file
        df = pd.read_csv(s3_object['Body'])

        # Save the DataFrame to a local CSV file
        df.to_csv(LOCAL_FILE_PATH, index=False)
```

```
        print(f"✅ Success! File downloaded and saved to: {LOCAL_FILE_PATH}")

    except Exception as e:
        print(f"❌ An error occurred: {e}")

if __name__ == "__main__":
    download_and_save_s3_csv()
```



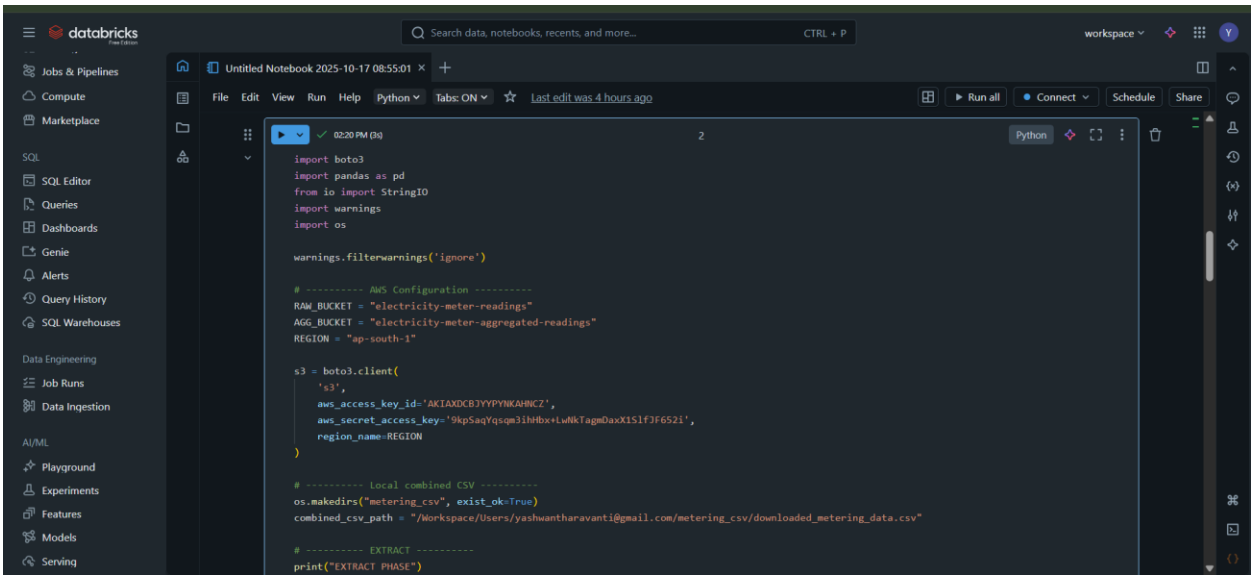



**Cell 2: End-to-End ETL Pipeline: Cleaning, Aggregation, & Billing**

```
import boto3
import pandas as pd
from io import StringIO
import warnings
import os

warnings.filterwarnings('ignore')

# ---------- AWS Configuration ----------
RAW_BUCKET = "electricity-meter-readings"
AGG_BUCKET = "electricity-meter-aggregated-readings"
REGION = "ap-south-1"

s3 = boto3.client(
    's3',
    aws_access_key_id='',
    aws_secret_access_key='',
    region_name=REGION
)

# ---------- Local combined CSV ----------
os.makedirs("metering_csv", exist_ok=True)
combined_csv_path =
"/Workspace/Users/yashwantharavanti@gmail.com/metering_csv/downloaded_metering_data.csv"

# ---------- EXTRACT ----------
print("EXTRACT PHASE")
df = pd.read_csv(combined_csv_path, quotechar='"', on_bad_lines='skip', low_memory=False)
print(f"Combined CSV loaded: {len(df)} records, columns: {df.columns.tolist()}")

# ---------- TRANSFORM: Cleaning ----------
print("Cleaning data")

# Convert Date column to datetime
df['Date'] = pd.to_datetime(df['Date'], dayfirst=True, errors='coerce')
df = df.dropna(subset=['Date'])

# Convert numeric columns
numeric_cols = [
    'Global_active_power', 'Global_reactive_power',
    'Voltage', 'Global_intensity',
    'Sub_metering_1', 'Sub_metering_2', 'Sub_metering_3'
]

for c in numeric_cols:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors='coerce')

# Remove rows where all numeric values are NaN or zero
print("Removing empty or zero-value rows")
df = df.dropna(subset=numeric_cols, how='all')
df = df[~(df[numeric_cols].fillna(0) == 0).all(axis=1)]

print(f"Records after removing empty/zero rows: {len(df)}")

# ---------- TRANSFORM: Daily Aggregation ----------
print("Aggregating daily data")
daily_agg = df.groupby('Date').agg({
    'Global_active_power': 'mean',
    'Global_reactive_power': 'mean',
    'Voltage': ['mean', 'min', 'max'],
```

```
    'Global_intensity': 'mean',
    'Sub_metering_1': 'sum',
    'Sub_metering_2': 'sum',
    'Sub_metering_3': 'sum'
})

# Flatten column names
daily_agg.columns = [
    'avg_Global_active_power', 'avg_Global_reactive_power',
    'avg_Voltage', 'min_Voltage', 'max_Voltage',
    'avg_Global_intensity',
    'total_Sub_metering_1', 'total_Sub_metering_2', 'total_Sub_metering_3'
]
daily_agg = daily_agg.reset_index()

# Average sub-metering
daily_agg['avg_submetering_value'] = (
    daily_agg['total_Sub_metering_1'] +
    daily_agg['total_Sub_metering_2'] +
    daily_agg['total_Sub_metering_3']
) / 3

# Total daily sum
daily_agg['total_daily_sum'] = (
    daily_agg['total_Sub_metering_1'] +
    daily_agg['total_Sub_metering_2'] +
    daily_agg['total_Sub_metering_3']
)

# ---------- Anomaly Detection (3-sigma) ----------
mean_sub = daily_agg['avg_submetering_value'].mean()
std_sub = daily_agg['avg_submetering_value'].std()

daily_agg['anomaly_flag'] = (
    (daily_agg['avg_submetering_value'] > mean_sub + 3 * std_sub) |
    (daily_agg['avg_submetering_value'] < mean_sub - 3 * std_sub)
)

# ---------- NIGHTLY ETL: Peak/Off-Peak Billing ----------
print("Calculating Peak / Off-Peak billing")
if 'Time' in df.columns:
    df['DateTime'] = pd.to_datetime(
        df['Date'].dt.strftime('%Y-%m-%d') + ' ' + df['Time'],
        errors='coerce'
    )
    df['Hour'] = df['DateTime'].dt.hour
    df['period'] = df['Hour'].apply(lambda x: 'peak' if 18 <= x < 22 else 'offpeak')
    df['Global_active_power'] = pd.to_numeric(df['Global_active_power'], errors='coerce')

    daily_power = df.groupby(['Date',
'period'])['Global_active_power'].sum().unstack(fill_value=0).reset_index()

    PEAK_RATE = 8.5
    OFFPEAK_RATE = 5.0

    daily_power['peak_charge'] = daily_power.get('peak', 0) * PEAK_RATE
    daily_power['offpeak_charge'] = daily_power.get('offpeak', 0) * OFFPEAK_RATE
    daily_power['total_charge'] = daily_power['peak_charge'] + daily_power['offpeak_charge']

    # Merge billing with aggregated data
    final_agg = pd.merge(
        daily_agg,
        daily_power[['Date', 'peak_charge', 'offpeak_charge', 'total_charge']],
        on='Date',
        how='left'
```

```
    )
else:
    print("Time column not found, skipping peak/off-peak billing")
    final_agg = daily_agg.copy()

# ---------- LOAD: Upload to S3 ----------
print("Uploading final aggregated ETL file to S3")
csv_buffer = StringIO()
final_agg.to_csv(csv_buffer, index=False)

s3_key = f"data/billing_agg_{pd.Timestamp.now().strftime('%Y-%m-%d_%H-%M-%S')}.csv"

s3.put_object(
    Bucket=AGG_BUCKET,
    Key=s3_key,
    Body=csv_buffer.getvalue(),
    ContentType='text/csv'
)

print(f"☑ ETL Completed! Uploaded to s3://{AGG_BUCKET}/{s3_key}")
print(f"Total records in final output: {len(final_agg)}")
```



**Explanation:**

**1. AWS and Python Setup**

This initial section configures the script's environment.

- It imports all the necessary libraries like **boto3** (for AWS), **pandas** (for data handling), and **os** (for file paths).
- It establishes a connection to your AWS account using the **boto3** client.
- It defines key variables for your S3 buckets and AWS region:
    - RAW_BUCKET: The S3 bucket for your original, untouched data.
    - AGG_BUCKET: The destination S3 bucket for the cleaned and processed data.
    - REGION: Your specified AWS region (e.g., ap-south-1).

**2. Extract Phase (Reading Data)**

This phase focuses on ingesting the source data.

- It reads a local CSV file (metering_csv/combined_metering.csv) into memory.
- All the power consumption data is loaded into a powerful and flexible **Pandas DataFrame**.
- The script is designed to automatically **skip any broken or badly formatted lines** in the CSV to prevent errors.

**3. Clean Phase (Data Cleaning)**

This is where the raw data is tidied up and prepared for calculations.

- It converts the Date column from text into a proper **datetime format**, recognizing the DD-MM-YYYY structure.
- It **removes any rows** that have a missing or invalid date.
- Key measurement columns (Global_active_power, Voltage, etc.) are converted from text to **numeric types**, making them ready for mathematical operations.

**4. Aggregate Phase (Daily Summary)**

The script transforms the minute-by-minute data into a useful daily summary.

- It **groups all the individual readings by Date**.

- For each day, it calculates key statistics:
    - **Mean** of power, voltage, and intensity.
    - **Minimum** and **maximum** voltage recorded.
    - **Total (sum)** of energy consumed by each of the three sub-meters.
- The result is a new, much smaller table containing one row for each day's summary.

**5. Derived Metrics (Feature Engineering)**

Here, new, insightful columns are created from the aggregated data.

- It computes the **average daily consumption** across the three sub-meters.
- It also calculates the **total daily energy consumption** by summing the sub-meters.

**6. Anomaly Detection**

This section automatically identifies unusual patterns in the data.

- It calculates the overall **mean and standard deviation** for the average sub-metering values across all days.
- Using the **3-sigma rule**, it flags any day as an "anomaly" if its consumption is unusually high or low (more than 3 standard deviations from the average).

**7. Peak & Off-Peak Billing**

This is the core business logic, where energy costs are calculated.

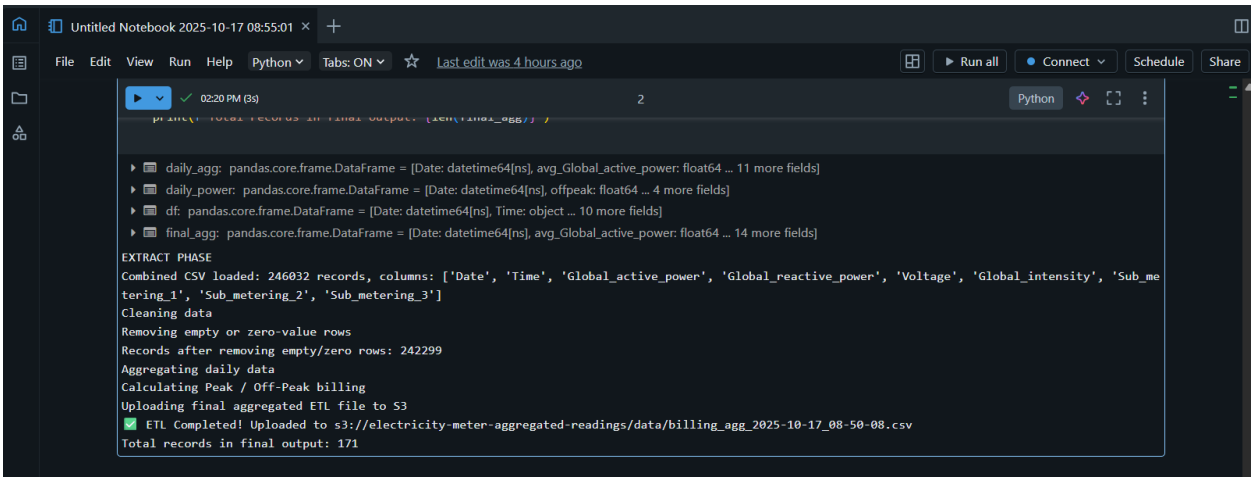
- It combines the Date and Time columns into a single DateTime column.
- It extracts the hour of the day to determine if a reading occurred during:
    - **Peak hours** (6 PM – 10 PM)
    - **Off-peak hours** (all other times)
- It applies different billing rates to the energy consumed in each period:
    - **Peak rate:** ₹8.5 per unit
    - **Off-peak rate:** ₹5.0 per unit
- Finally, it adds peak_charge, offpeak_charge, and total_charge columns to the daily summary.

### 8. Load Phase (Upload to S3)

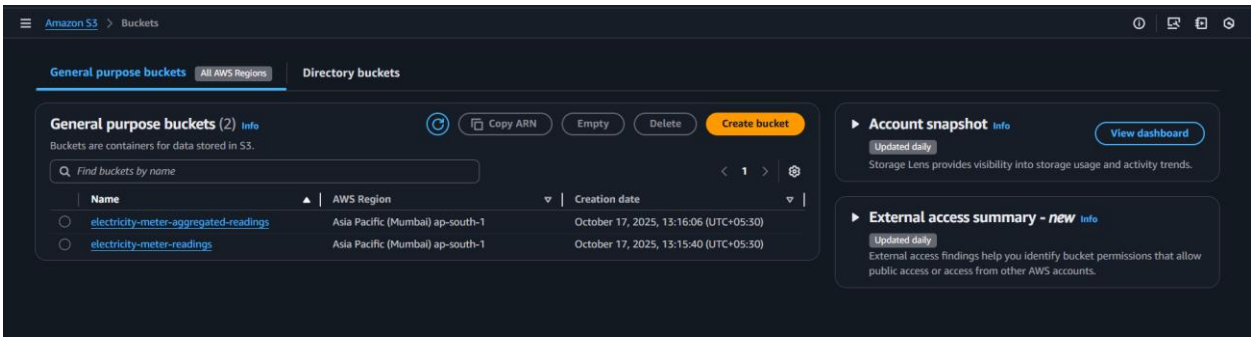
The final phase saves and stores the result of the ETL process.

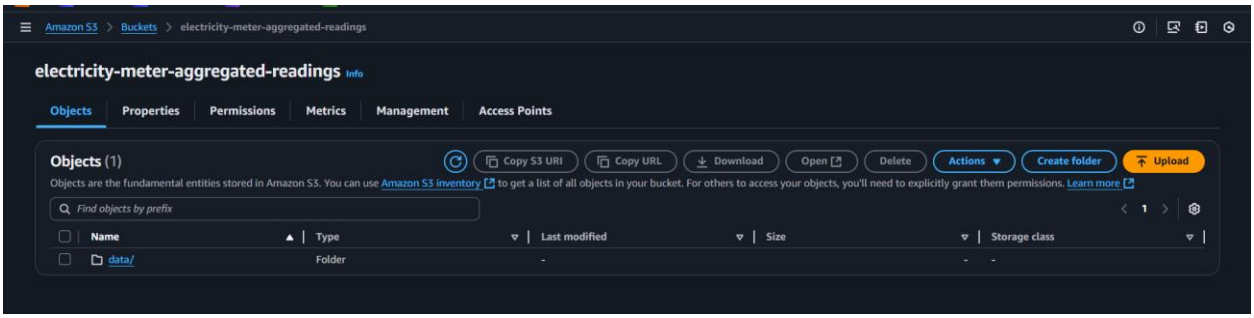
- It converts the final, enriched DataFrame into a CSV format in memory.
- It securely **uploads this file to your aggregated S3 bucket** (s3://meteringbucket1234-aggregated/).
- Each file is saved with a **unique timestamp** in its name (e.g., billing_agg_2025-10-17_17-51-31.csv), which prevents overwriting previous runs and provides a clear version history.

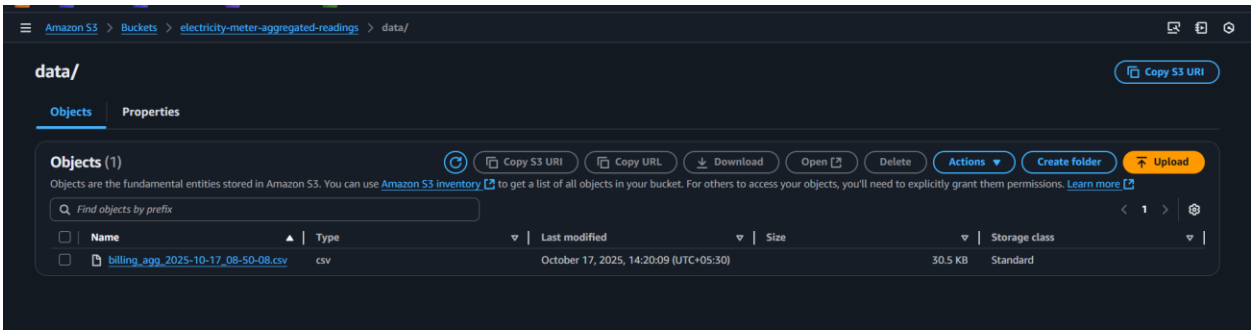
**Output:**

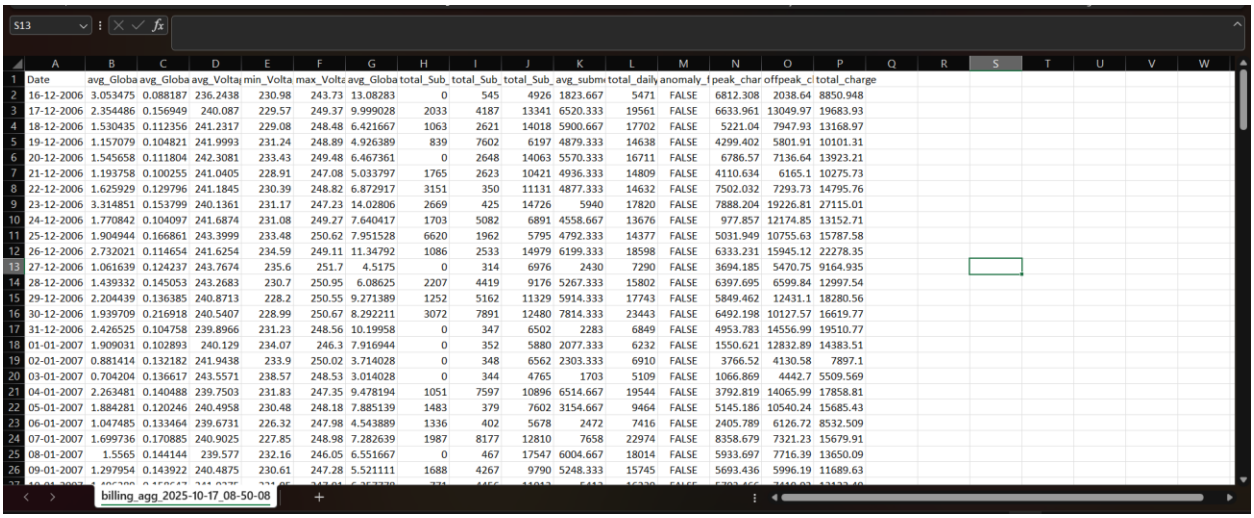



**Storing the Aggregated Data in S3:**



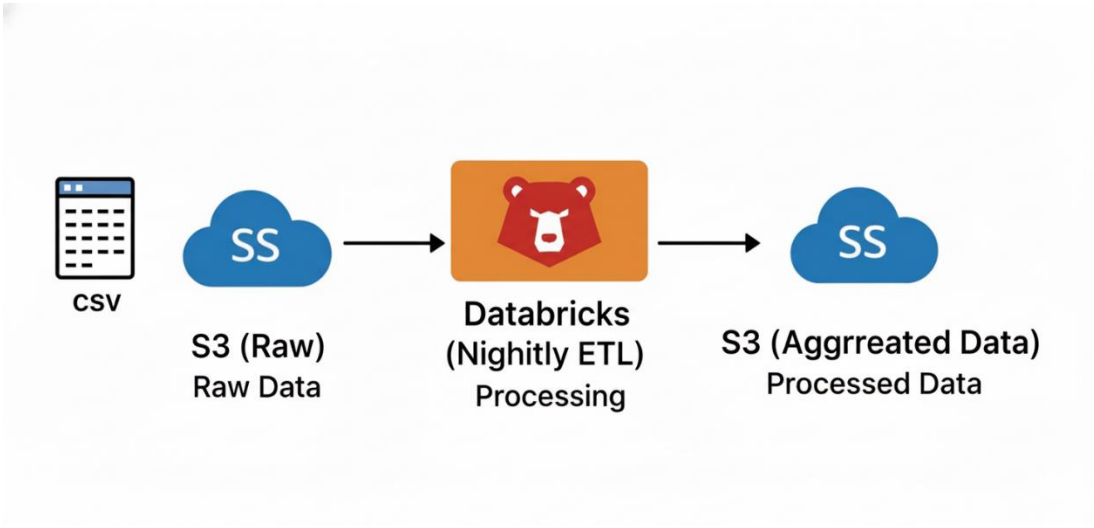





| Date | avg_Globa | avg_Globa | avg_Volta | min_Volta | max_Volta | avg_Globa | total_Sub_ | total_Sub_ | total_Sub_ | avg_subm | total_daily | anomaly_f | peak_char | offpeak_cl | total_charge |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16-12-2006 | 3.053475 | 0.088187 | 236.2438 | 230.98 | 243.73 | 13.08283 | 0 | 545 | 4926 | 1823.667 | 5471 | FALSE | 6812.308 | 2038.64 | 8850.948 |
| 17-12-2006 | 2.354486 | 0.156949 | 240.087 | 229.57 | 249.37 | 9.999028 | 2033 | 4187 | 13341 | 6520.333 | 19561 | FALSE | 6633.961 | 13049.97 | 19683.93 |
| 18-12-2006 | 1.530435 | 0.112356 | 241.2317 | 229.08 | 248.48 | 6.421667 | 1063 | 2621 | 14018 | 5900.667 | 17702 | FALSE | 5221.04 | 7947.93 | 13168.97 |
| 19-12-2006 | 1.157079 | 0.104821 | 241.9993 | 231.24 | 248.89 | 4.926389 | 839 | 7602 | 6197 | 4879.333 | 14638 | FALSE | 4299.402 | 5801.91 | 10101.31 |
| 20-12-2006 | 1.545658 | 0.111804 | 242.3081 | 233.43 | 249.48 | 6.467361 | 0 | 2648 | 14063 | 5570.333 | 16711 | FALSE | 6786.57 | 7136.64 | 13923.21 |
| 21-12-2006 | 1.193758 | 0.100255 | 241.0405 | 228.91 | 247.08 | 5.033797 | 1765 | 2623 | 10421 | 4936.333 | 14809 | FALSE | 4110.634 | 6165.1 | 10275.73 |
| 22-12-2006 | 1.625929 | 0.129796 | 241.1845 | 230.39 | 248.82 | 6.872917 | 3151 | 350 | 11131 | 4877.333 | 14632 | FALSE | 7502.032 | 7293.73 | 14795.76 |
| 23-12-2006 | 3.314851 | 0.153799 | 240.1361 | 231.17 | 247.23 | 14.02806 | 2669 | 425 | 14726 | 5940 | 17820 | FALSE | 7888.204 | 19226.81 | 27115.01 |
| 24-12-2006 | 1.770842 | 0.104097 | 241.6874 | 231.08 | 249.27 | 7.640417 | 1703 | 5082 | 6891 | 4558.667 | 13676 | FALSE | 977.857 | 12174.85 | 13152.71 |
| 25-12-2006 | 1.904944 | 0.166861 | 243.3999 | 233.48 | 250.62 | 7.951528 | 6620 | 1962 | 5795 | 4792.333 | 14377 | FALSE | 5031.949 | 10755.63 | 15787.58 |
| 26-12-2006 | 2.732021 | 0.114654 | 241.6254 | 234.59 | 249.11 | 11.34792 | 1086 | 2533 | 14979 | 6199.333 | 18598 | FALSE | 6333.231 | 15945.12 | 22278.35 |
| 27-12-2006 | 1.061639 | 0.124237 | 243.7674 | 235.6 | 251.7 | 4.5175 | 0 | 314 | 6976 | 2430 | 7290 | FALSE | 3694.185 | 5470.75 | 9164.935 |
| 28-12-2006 | 1.439332 | 0.145053 | 243.2683 | 230.7 | 250.95 | 6.08625 | 2207 | 4419 | 9176 | 5267.333 | 15802 | FALSE | 6397.695 | 6599.84 | 12997.54 |
| 29-12-2006 | 2.204439 | 0.136385 | 240.8713 | 228.2 | 250.55 | 9.271389 | 1252 | 5162 | 11329 | 5914.333 | 17743 | FALSE | 5849.462 | 12431.1 | 18280.56 |
| 30-12-2006 | 1.939709 | 0.216918 | 240.5407 | 228.99 | 250.67 | 8.292211 | 3072 | 7891 | 12480 | 7814.333 | 23443 | FALSE | 6492.198 | 10127.57 | 16619.77 |
| 31-12-2006 | 2.426525 | 0.104758 | 239.8966 | 231.23 | 248.56 | 10.19958 | 0 | 347 | 6502 | 2283 | 6849 | FALSE | 4953.783 | 14556.99 | 19510.77 |
| 01-01-2007 | 1.909031 | 0.102893 | 240.129 | 234.07 | 246.3 | 7.916944 | 0 | 352 | 5880 | 2077.333 | 6232 | FALSE | 1550.621 | 12832.89 | 14383.51 |
| 02-01-2007 | 0.881414 | 0.132182 | 241.9438 | 233.9 | 250.02 | 3.714028 | 0 | 348 | 6562 | 2303.333 | 6910 | FALSE | 3766.52 | 4130.58 | 7897.1 |
| 03-01-2007 | 0.704204 | 0.136617 | 243.5571 | 238.57 | 248.53 | 3.014028 | 0 | 344 | 4765 | 1703 | 5109 | FALSE | 1066.869 | 4442.7 | 5509.569 |
| 04-01-2007 | 2.263481 | 0.140488 | 239.7503 | 231.83 | 247.35 | 9.478194 | 1051 | 7597 | 10896 | 6514.667 | 19544 | FALSE | 3792.819 | 14065.99 | 17858.81 |
| 05-01-2007 | 1.884281 | 0.120246 | 240.4958 | 230.48 | 248.18 | 7.885139 | 1483 | 379 | 7602 | 3154.667 | 9464 | FALSE | 5145.186 | 10540.24 | 15685.43 |
| 06-01-2007 | 1.047485 | 0.133464 | 239.6731 | 226.32 | 247.98 | 4.543889 | 1336 | 402 | 5678 | 2472 | 7416 | FALSE | 2405.789 | 6126.72 | 8532.509 |
| 07-01-2007 | 1.699736 | 0.170885 | 240.9025 | 227.85 | 248.98 | 7.282639 | 1987 | 8177 | 12810 | 7658 | 22974 | FALSE | 8358.679 | 7321.23 | 15679.91 |
| 08-01-2007 | 1.5565 | 0.144144 | 239.577 | 232.16 | 246.05 | 6.551667 | 0 | 467 | 17547 | 6004.667 | 18014 | FALSE | 5933.697 | 7716.39 | 13650.09 |
| 09-01-2007 | 1.297954 | 0.143922 | 240.4875 | 230.61 | 247.28 | 5.521111 | 1688 | 4267 | 9790 | 5248.333 | 15745 | FALSE | 5693.436 | 5996.19 | 11689.63 |

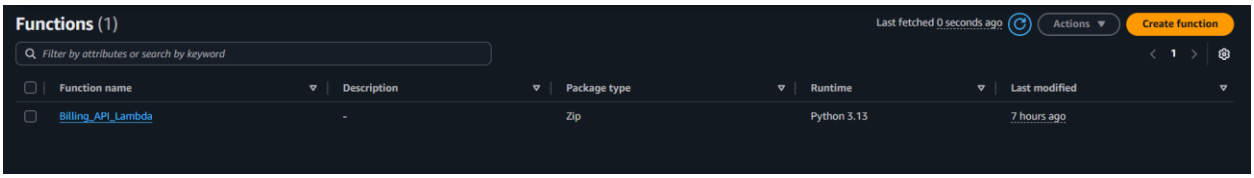




**Provide REST APIs for consumers to fetch monthly bills:**

**Step 5: Create a Lambda Function to trigger the API Gateway to fetch monthly bills**

1. Navigate to the AWS Lambda console and click the Create function button.
2. Select the Author from scratch option.
3. Enter the following basic information for your function:
   - Function name: BillingAPILambda
   - Runtime: Python 3.9
4. Configure the necessary permissions for your Lambda function to access the S3 bucket (meteringbucket1234-aggregated). This is typically done by attaching an IAM role with S3 read permissions.
5. Click Create function.




**Lambda_function.py:**

```
import json
import boto3
import pandas as pd

# ------------------ S3 Configuration ------------------
BUCKET_NAME = "electricity-meter-aggregated-readings"
S3_PREFIX = "data/billing_agg"
s3 = boto3.client('s3')

# ------------------ Lambda Handler ------------------
def lambda_handler(event, context):
    try:
        # Debug: log incoming event
        print("Event received:", event)

        # Get 'month' parameter (format: YYYY-MM)
        month = (event.get('queryStringParameters') or {}).get('month')
        if not month:
            return {
                "statusCode": 400,
                "body": json.dumps({"error": "Month parameter is required, e.g., ?month=2025-10"})
            }

        # List all billing files in S3
        response = s3.list_objects_v2(Bucket=BUCKET_NAME, Prefix=S3_PREFIX)
        if 'Contents' not in response:
            return {
                "statusCode": 404,
                "body": json.dumps({"error": "No billing files found in S3"})
            }

        # Pick the latest file based on LastModified
        latest_file = max(response['Contents'], key=lambda x: x['LastModified'])['Key']
        print("Latest billing file:", latest_file)
```

```
        # Load CSV into DataFrame
        obj = s3.get_object(Bucket=BUCKET_NAME, Key=latest_file)
        df = pd.read_csv(obj['Body'])

        # Ensure 'Date' column exists and parse it
        if 'Date' not in df.columns:
            return {
                "statusCode": 500,
                "body": json.dumps({"error": "CSV missing 'Date' column"})
            }
        df['Date'] = pd.to_datetime(df['Date'], dayfirst=True, errors='coerce')
        df = df.dropna(subset=['Date'])
        if df.empty:
            return {
                "statusCode": 500,
                "body": json.dumps({"error": "No valid dates found in CSV"})
            }

        # Fill missing billing columns with 0
        billing_cols = [
            'total_Sub_metering_1', 'total_Sub_metering_2', 'total_Sub_metering_3',
            'avg_submetering_value', 'total_daily_sum',
            'peak_charge', 'offpeak_charge', 'total_charge'
        ]
        for col in billing_cols:
            if col not in df.columns:
                df[col] = 0
        df[billing_cols] = df[billing_cols].fillna(0)

        # Filter data for requested month
        monthly_df = df[df['Date'].dt.strftime('%Y-%m') == month]
        if monthly_df.empty:
            return {
                "statusCode": 404,
                "body": json.dumps({"error": f"No billing data found for month {month}"})
            }

        # Select relevant columns and format date
        monthly_df = monthly_df[['Date'] + billing_cols]
        monthly_df['Date'] = monthly_df['Date'].dt.strftime('%d-%m-%Y')  # Format for JSON

        # Return JSON response
        return {
            "statusCode": 200,
            "body": monthly_df.to_json(orient='records')
        }

    except Exception as e:
        print("Error:", str(e))
        return {
            "statusCode": 500,
            "body": json.dumps({"error": "Internal server error", "details": str(e)})
        }
```

**Purpose:** This Lambda function serves as the serverless backend for your API, containing the logic to read processed data from S3, filter it by a specific month, and return a formatted bill on-demand.

**Explanation: AWS Lambda Function: Monthly Billing Report**

**1. Input and Event Handling**

- **Trigger:** The function is initiated by an HTTP event, managed via Amazon API Gateway.
- **Parameter Validation:** It reads the month query parameter from the request, expecting a YYYY-MM format.
- **Error Handling:** If the month parameter is missing or improperly formatted, the function returns an HTTP 400 Bad Request error.

**2. Fetch Latest Billing File from S3**

- **File Discovery:** The function lists all objects in the designated S3 bucket that match the prefix billing_agg_.
- **File Selection:** It identifies and selects the most recently modified CSV file from the list to process the latest available data.

**3. Load and Clean Data**

- **Data Ingestion:** The selected CSV file is loaded into a Pandas DataFrame.
- **Data Cleansing:**
    - The Date column is converted to a standard datetime format; rows that cannot be converted are removed.
    - The function verifies that all expected billing columns exist. If any are missing, they are created and filled with a default value of 0.

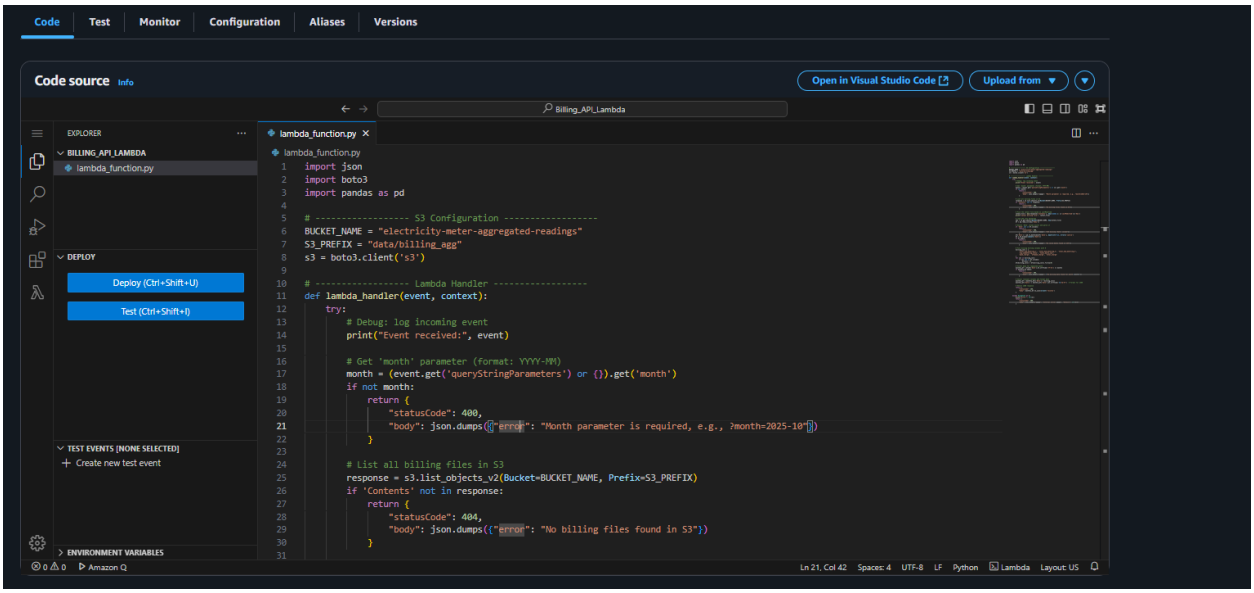
**4. Filter by Month**

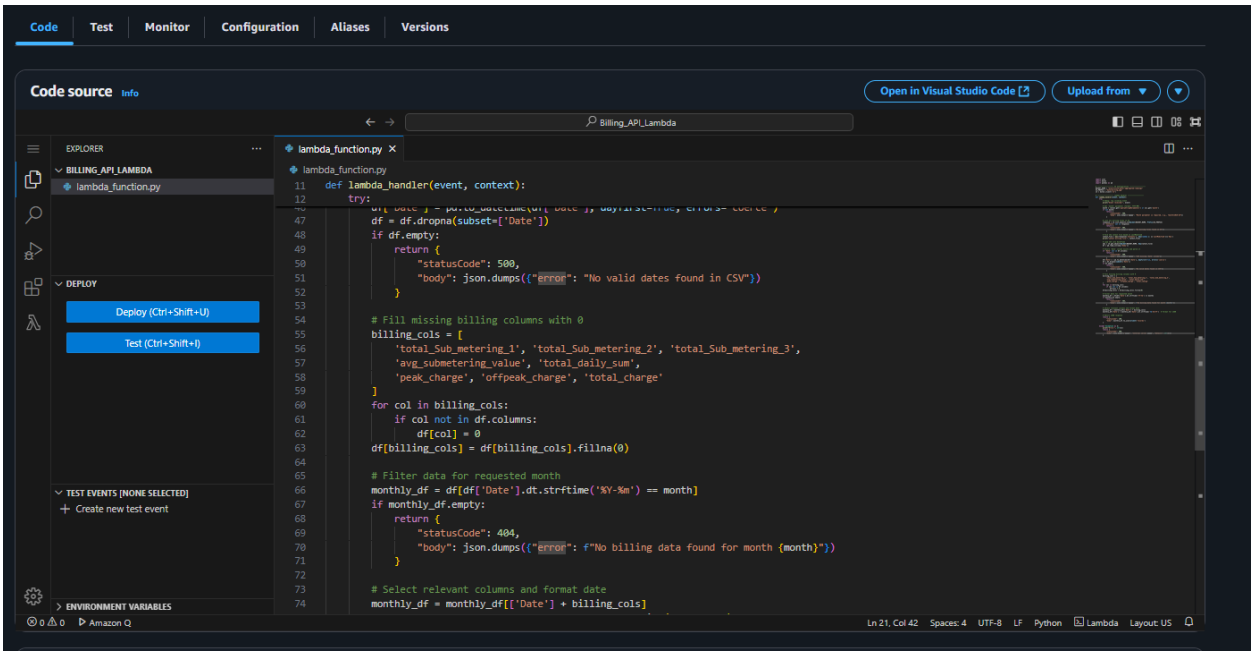
- **Data Filtering:** The DataFrame is filtered to include only the rows that fall within the requested month.
- **Error Handling:** If no data exists for the specified month after filtering, the function returns an HTTP 404 Not Found error.

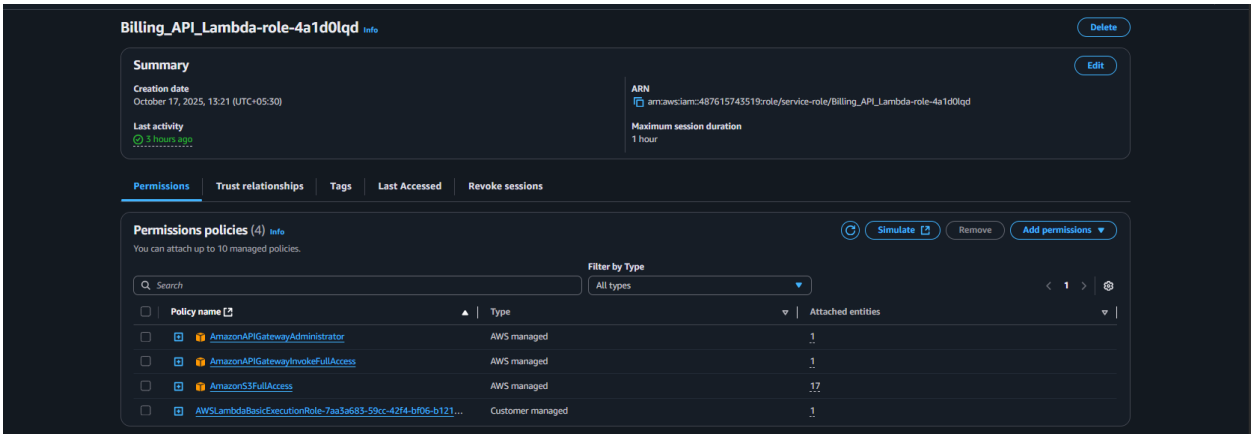
**5. Format and Return JSON Response**

- **Output Formatting:**
    - The Date column is formatted into a DD-MM-YYYY string.
    - Only the relevant billing columns are selected for the final output.
- **Successful Response:** The processed data is converted into a JSON object and returned with an HTTP 200 OK status.

- **Exception Handling:** A global try-catch block is implemented to handle any unexpected runtime errors. If an exception occurs, the function returns an HTTP 500 Internal Server Error with a JSON payload containing the error details.






**Step 6: Create a Rest API**

**1. Create a New REST API**

First, let's create the basic API container.

1. Navigate to the **API Gateway** service in your AWS Console.
2. Find the **REST API** box (not the private or WebSocket one) and click **Build**.
3. On the next screen, choose **New API**.
4. Enter MYAPI for the **API name**.
5. Leave the other settings as default and click **Create API**.

---

**2. Enable CORS and Create a Method**

Now, we'll enable Cross-Origin Resource Sharing (CORS) and create the GET endpoint.

1. With your new API selected, click the **Actions** dropdown menu and select **Enable CORS**.
2. On the CORS screen, just confirm the default settings (which handle OPTIONS methods and default responses like 4XX and 5XX) by clicking the **"Enable CORS and replace existing CORS headers"** button.
3. Back on the main screen, click the **Actions** menu again and select **Create Method**.
4. From the small dropdown that appears under the / resource, select **GET** and click the checkmark.

---

**3. Integrate with Lambda and Configure the Request**

This is where you connect your API endpoint to your Lambda function and tell it what inputs to expect.

1. For the **Integration type**, select **Lambda Function**.
2. Check the box for **Use Lambda Proxy integration**. This simplifies things by passing the entire request to your Lambda function.
3. In the **Lambda Function** field, paste the **ARN** (Amazon Resource Name) of your billing Lambda function.
4. Click **Save**. A pop-up will ask for permission to invoke your Lambda function; click **OK**.
5. You will now see the "Method Execution" diagram. Click on **Method Request**.
6. Expand the **URL Query String Parameters** section.
7. Click **Add query string**.
8. Enter month for the name and **check the box** next to it to make it **Required**. Click the checkmark to save.

---

**4. Deploy and Test Your API**

The final step is to make your API live and test it.

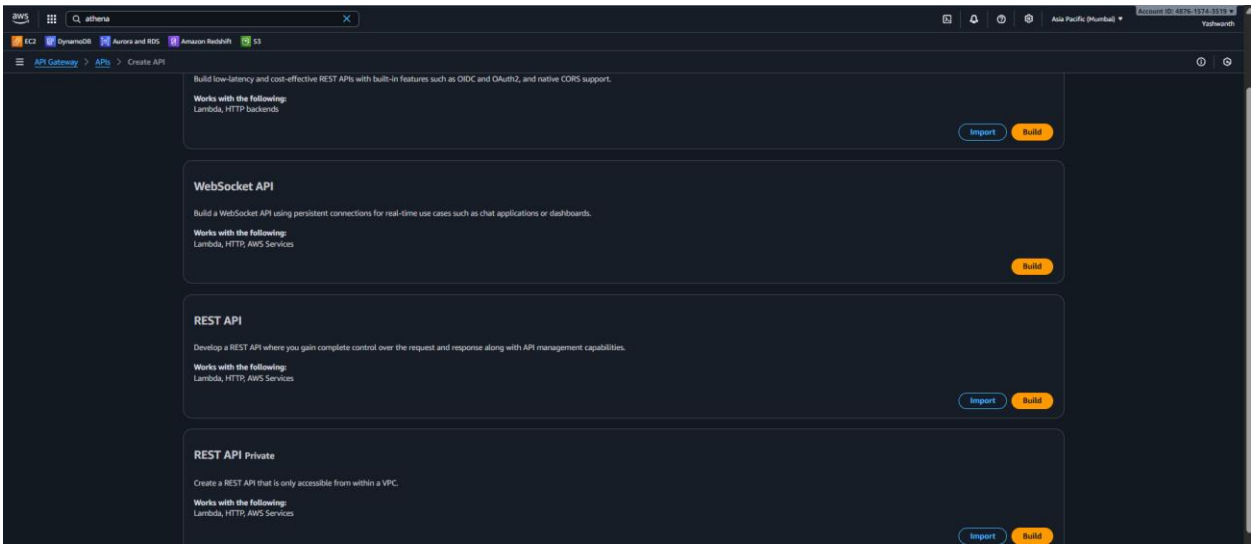
1. Click the **Actions** menu one last time and select **Deploy API**.
2. In the pop-up window, select **[New Stage]** for the **Deployment stage**.
3. Enter bills as the **Stage name**.
4. Click **Deploy**.
5. You'll be taken to the Stage Editor. In the navigation tree on the left, click on the **GET** method under your / resource.
6. The **Invoke URL** will be displayed at the top. It will look something like this: https://abcdef123.execute-api.us-east-1.amazonaws.com/bills.
7. **Copy this URL** and paste it into your web browser or an API client like Postman.
8. To test it, add your required query string parameter to the end of the URL: **?month=2007-04**.

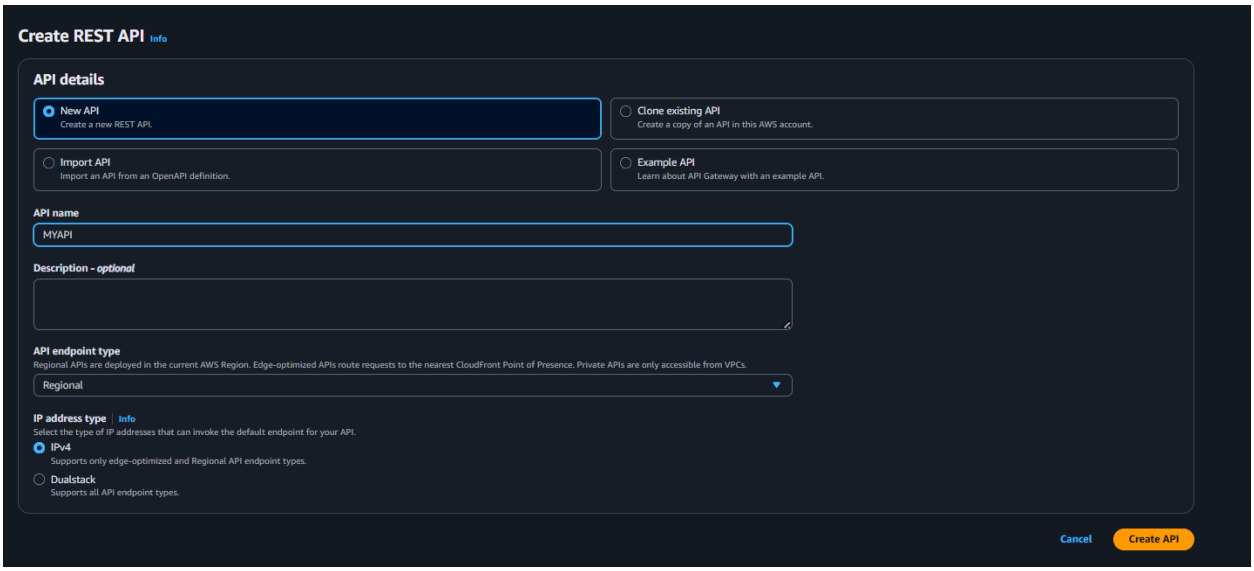
Your final test URL should look like this: https://abcdef123.execute-api.us-east-1.amazonaws.com/bills?month=2007-04

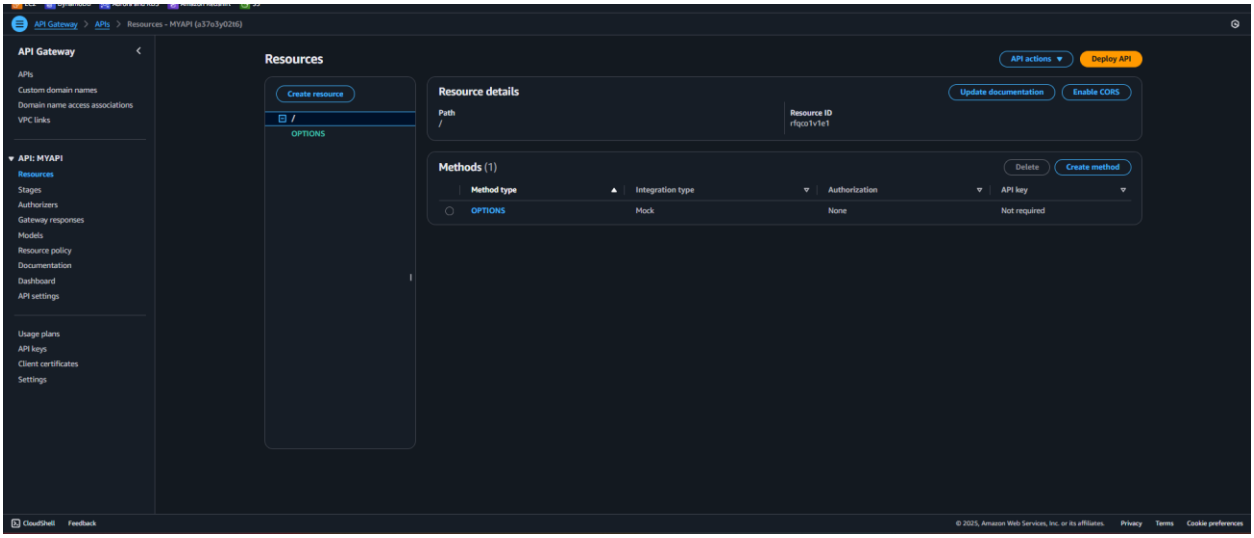
Pressing Enter should now trigger your Lambda function and return the billing data for April 2007.

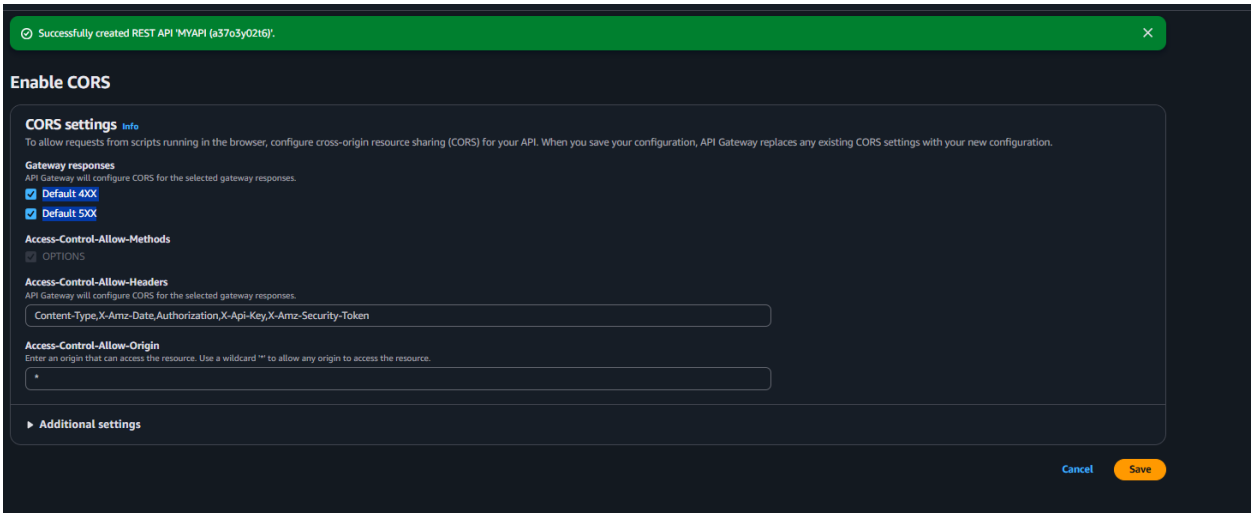
https://s8bb871sqb.execute-api.ap-south-1.amazonaws.com/new_satge/bills?month=2007-05

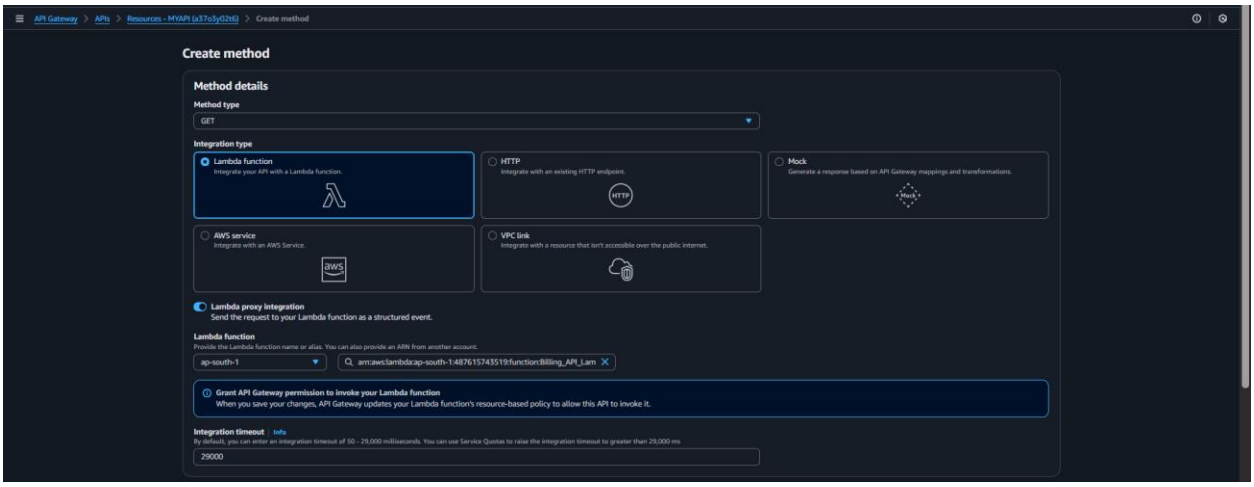
**Purpose:** This step uses API Gateway to create a secure and public REST API endpoint, effectively exposing your Lambda function to the internet so users can fetch billing data on-demand by calling a simple URL.

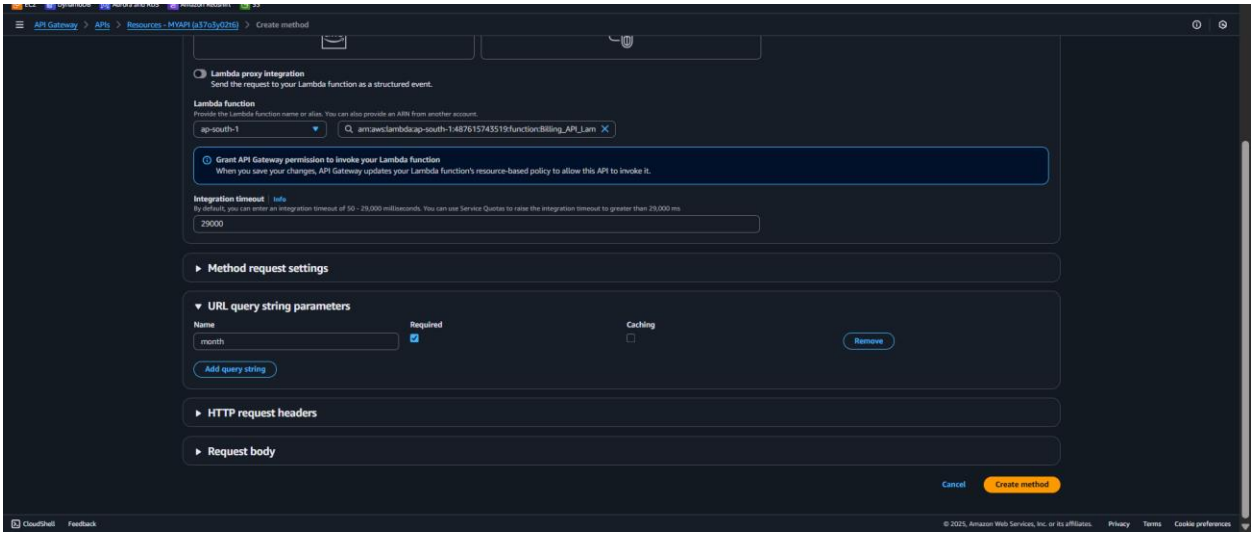


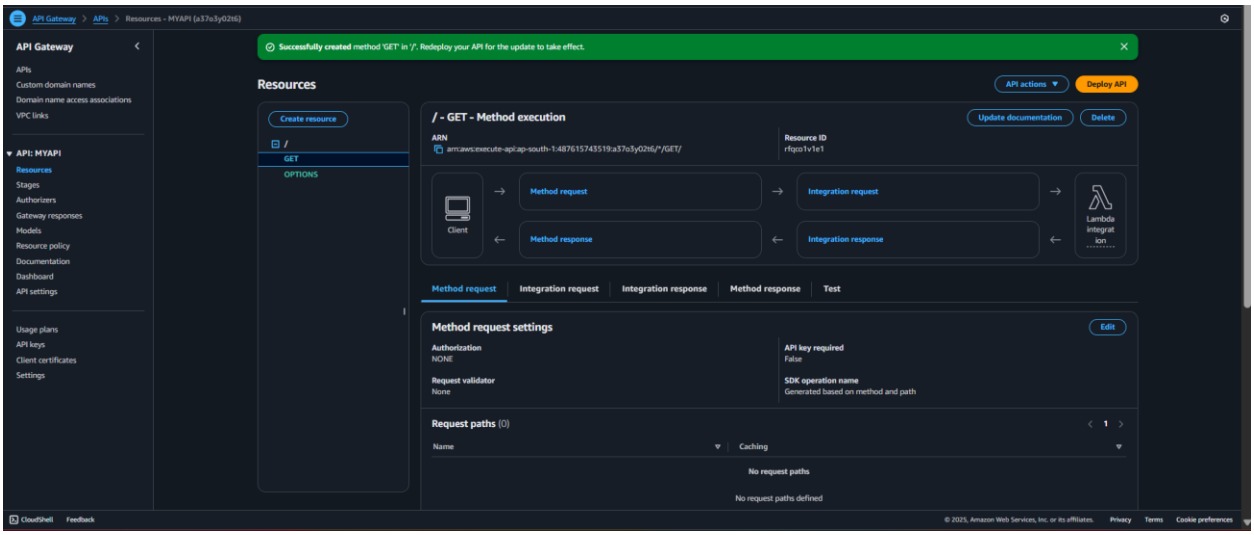



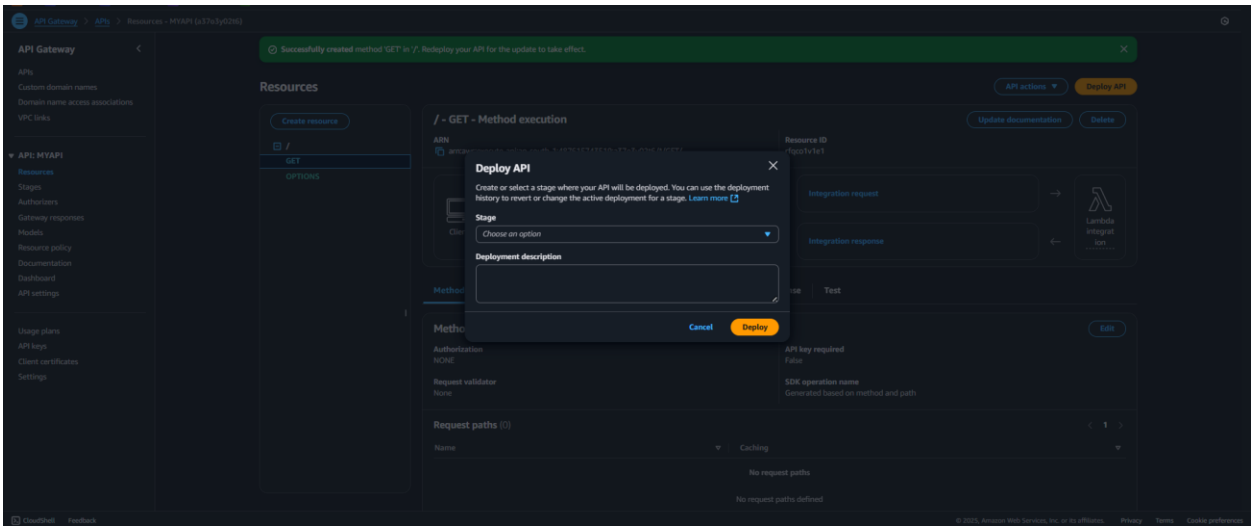

API Gateway
APIs
Resources - MYAPI (a37o3y02t6)
Resources
Create resource
/
OPTIONS
Resource details
Path
/
Resource ID
rfqco1v1e1
API actions
Deploy API
Update documentation
Enable CORS
Methods (1)
Delete
Create method
Method type
Integration type
Authorization
API key
OPTIONS
Mock
None
Not required
CloudShell
Feedback

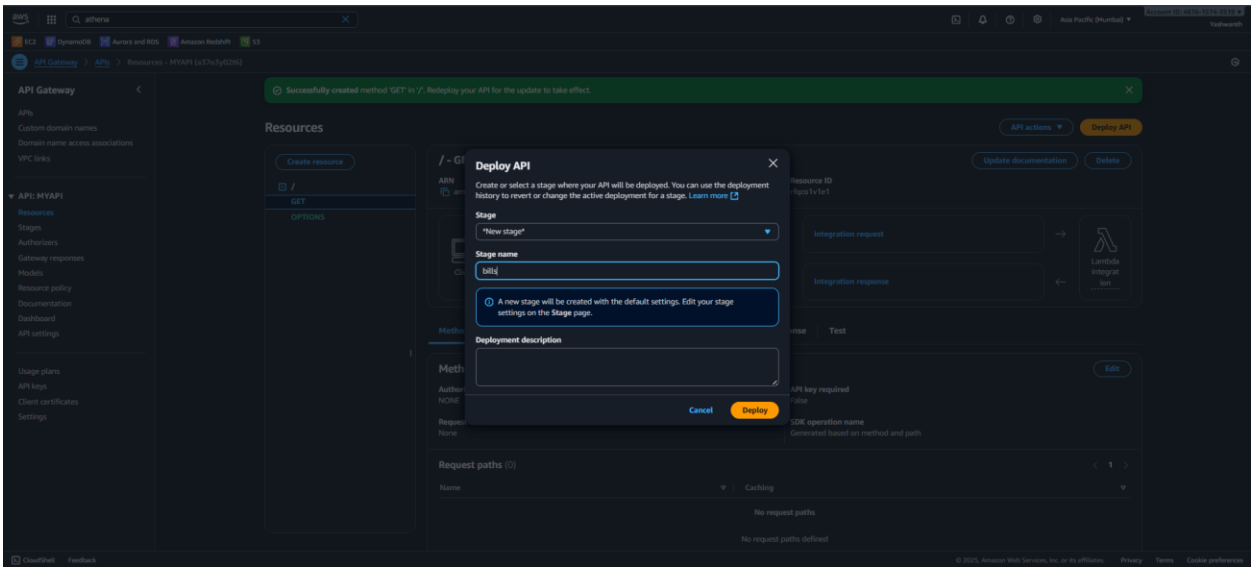

Successfully created REST API 'MYAPI (a37o3y02t6)'.
Enable CORS
CORS settings Info
To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API. When you save your configuration, API Gateway replaces any existing CORS settings with your new configuration.
Gateway responses
API Gateway will configure CORS for the selected gateway responses.
Default 4XX
Default 5XX
Access-Control-Allow-Methods
OPTIONS
Access-Control-Allow-Headers
API Gateway will configure CORS for the selected gateway responses.
Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token
Access-Control-Allow-Origin
Enter an origin that can access the resource. Use a wildcard '*' to allow any origin to access the resource.
*
Additional settings
Cancel
Save

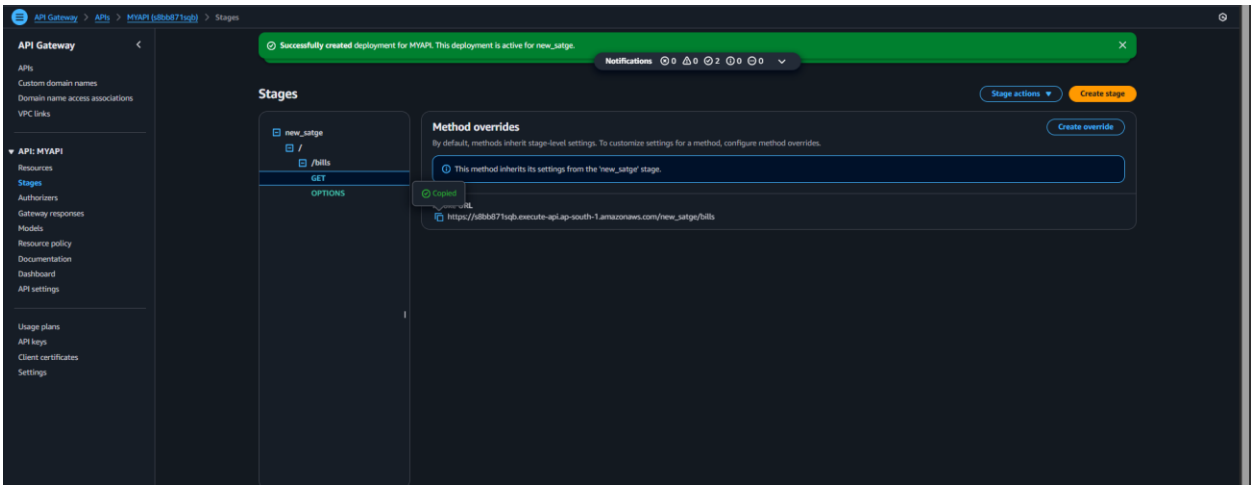

API Gateway
APIs
Resources - MYAPI (a37o3y02t6)
Create method
Create method
Method details
Method type
GET
Integration type
Lambda function
Integrate your API with a Lambda function.
HTTP
Integrate with an existing HTTP endpoint.
Mock
Generate a response based on API Gateway mappings and transformations.
AWS service
Integrate with an AWS Service.
VPC link
Integrate with a resource that isn't accessible over the public internet.
Lambda proxy integration
Send the request to your Lambda function as a structured event.
Lambda function
ap-south-1
Grant API Gateway permission to invoke your Lambda function
When you save your changes, API Gateway updates your Lambda function's resource-based policy to allow this API to invoke it.
Integration timeout Info
29000

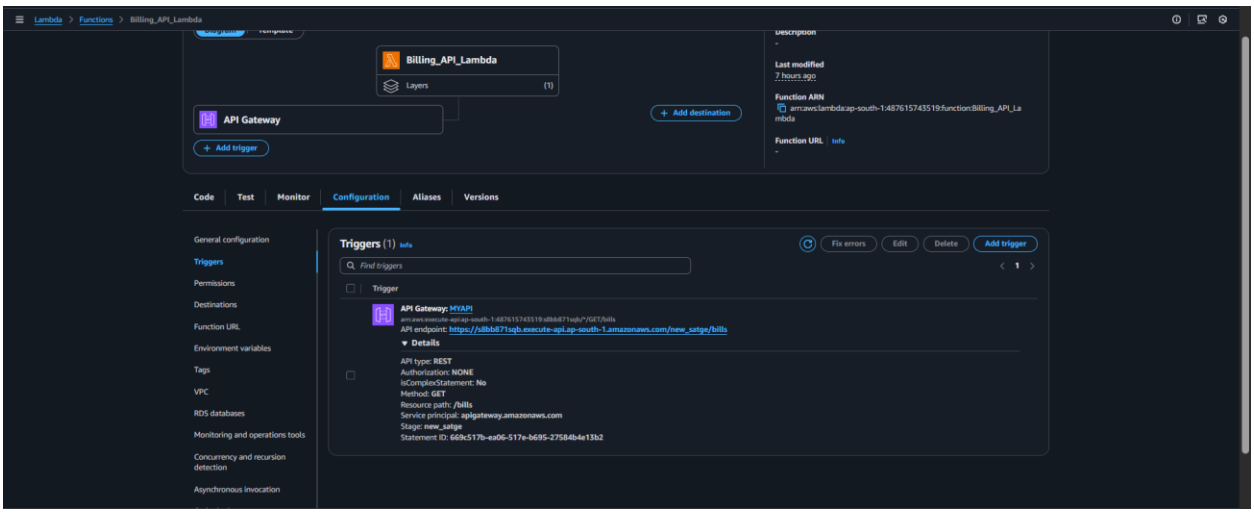

Lambda proxy integration
Send the request to your Lambda function as a structured event.
Lambda function
ap-south-1
Grant API Gateway permission to invoke your Lambda function
When you save your changes, API Gateway updates your Lambda function's resource-based policy to allow this API to invoke it.
Integration timeout
29000
Method request settings
URL query string parameters
Name
Required
Caching
month
Remove
Add query string
HTTP request headers
Request body
Cancel
Create method

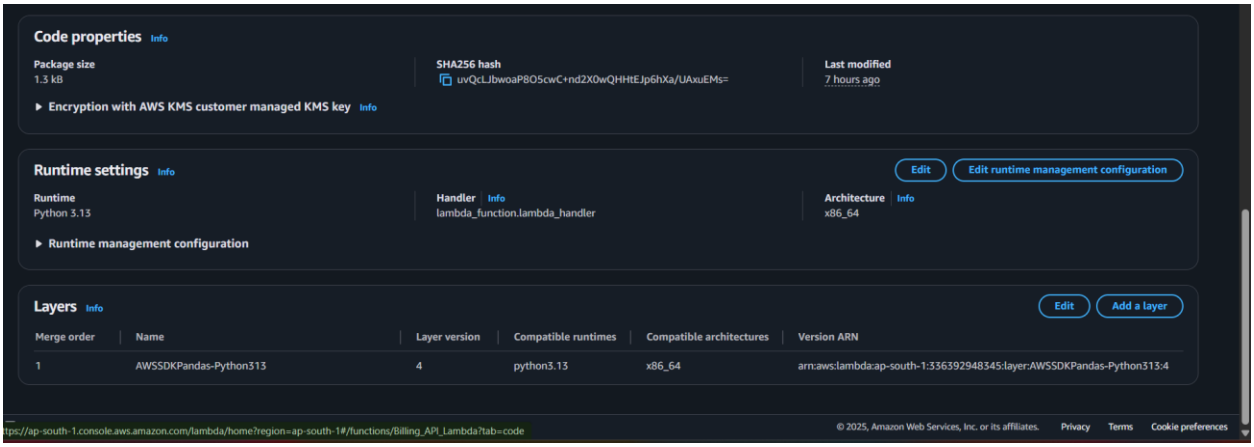

API Gateway
Successfully created method 'GET' in '/'. Redeploy your API for the update to take effect.
Resources
API actions
Deploy API
Create resource
/ - GET - Method execution
Update documentation
Delete
ARN
Resource ID
Client
Method request
Integration request
Method response
Integration response
Lambda integration
Method request
Integration request
Integration response
Method response
Test
Method request settings
Edit
Authorization
NONE
API key required
False
Request validator
None
SDK operation name
Generated based on method and path
Request paths (0)
Name
Caching
No request paths
No request paths defined

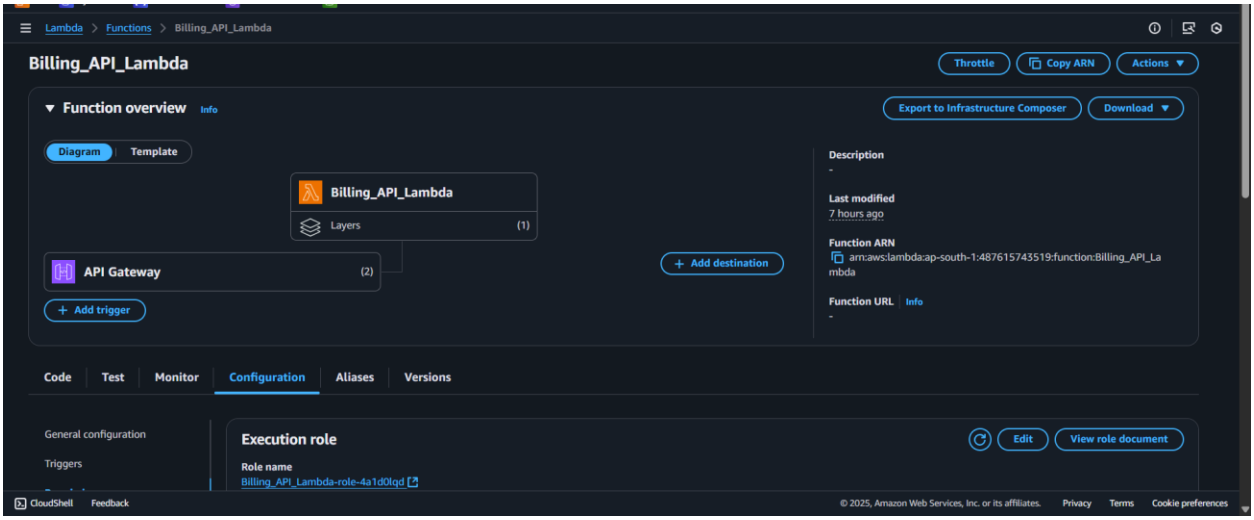

Deploy API
Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. Learn more
Stage
Choose an option
Deployment description
Cancel
Deploy


Deploy API
Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. Learn more
Stage
*New stage*
Stage name
bills
A new stage will be created with the default settings. Edit your stage settings on the Stage page.
Deployment description
Cancel
Deploy
Resources
Request paths (0)
No request paths
No request paths defined

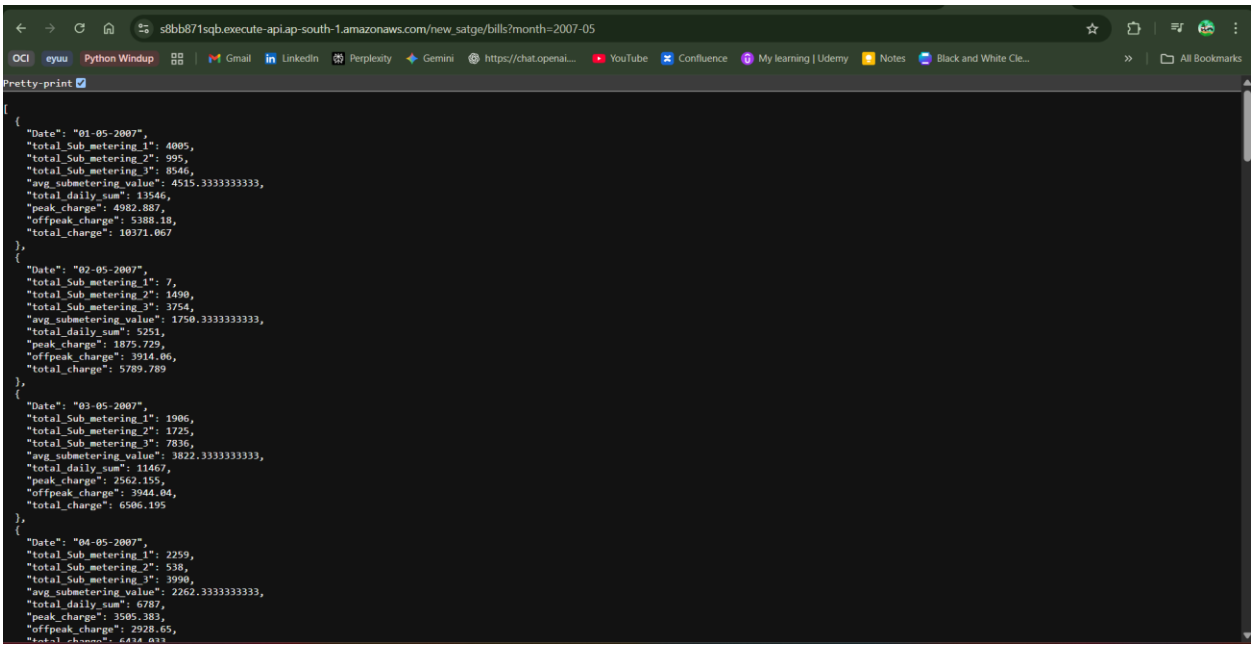

Successfully created deployment for MYAPI. This deployment is active for new_satge.
Stages
Stage actions
Create stage
new_satge
/bills
GET
OPTIONS
Method overrides
Create override
This method inherits its settings from the 'new_satge' stage.
Copied
https://s8bb871sqb.execute-api.ap-south-1.amazonaws.com/new_satge/bills


Billing_API_Lambda
Layers
(1)
API Gateway
Add trigger
Add destination
Last modified
7 hours ago
Function ARN
Function URL
Code
Test
Monitor
Configuration
Aliases
Versions
Triggers (1)
Fix errors
Edit
Delete
Add trigger
API Gateway: MYAPI
API endpoint: https://s8bb871sqb.execute-api.ap-south-1.amazonaws.com/new_satge/bills
Details
API type: REST
Authorization: NONE
isComplexStatement: No
Method: GET
Resource path: /bills
Service principal: apigateway.amazonaws.com
Stage: new_satge

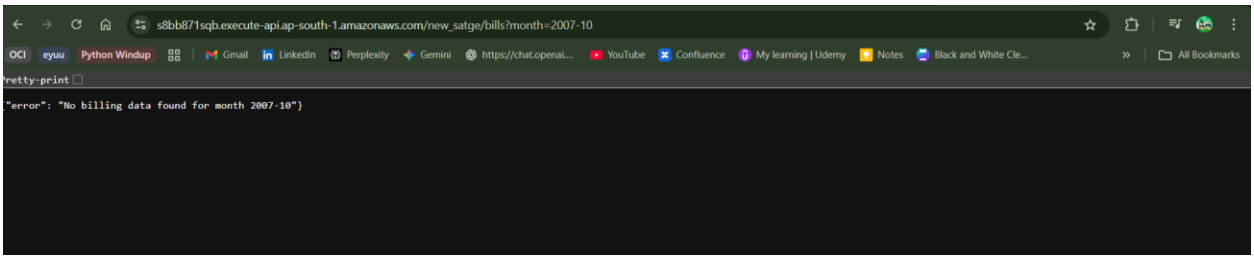


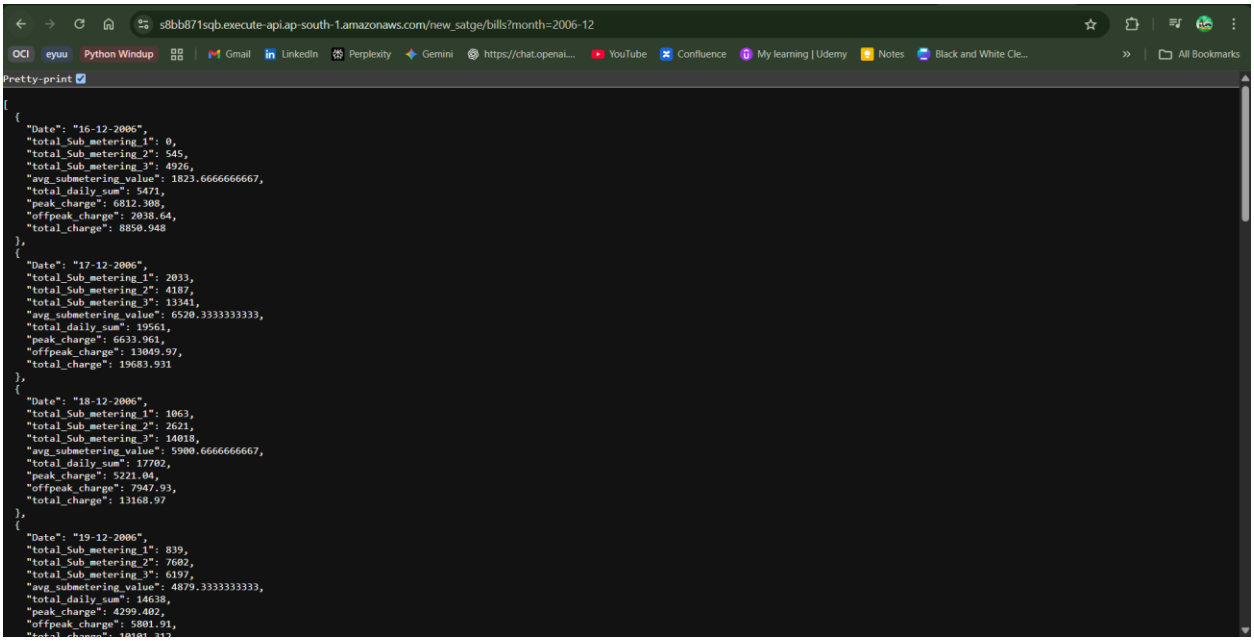

Month=05, Year=2007



Month=10, Year=2007




Month=12, Year=2006



**Anomaly detection via rule-based thresholds and alerts:**

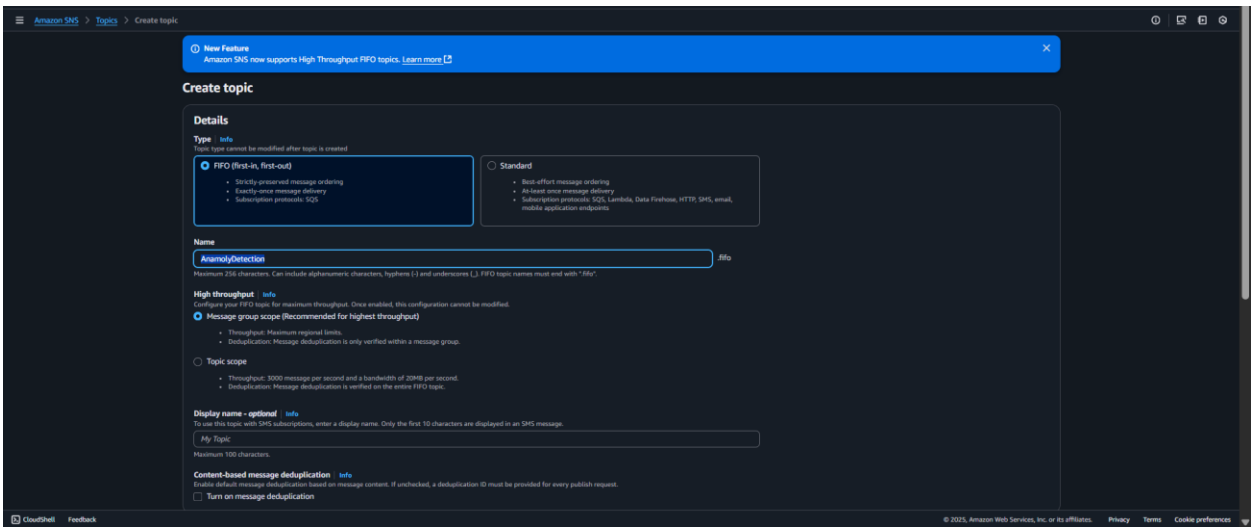
**Flow:**

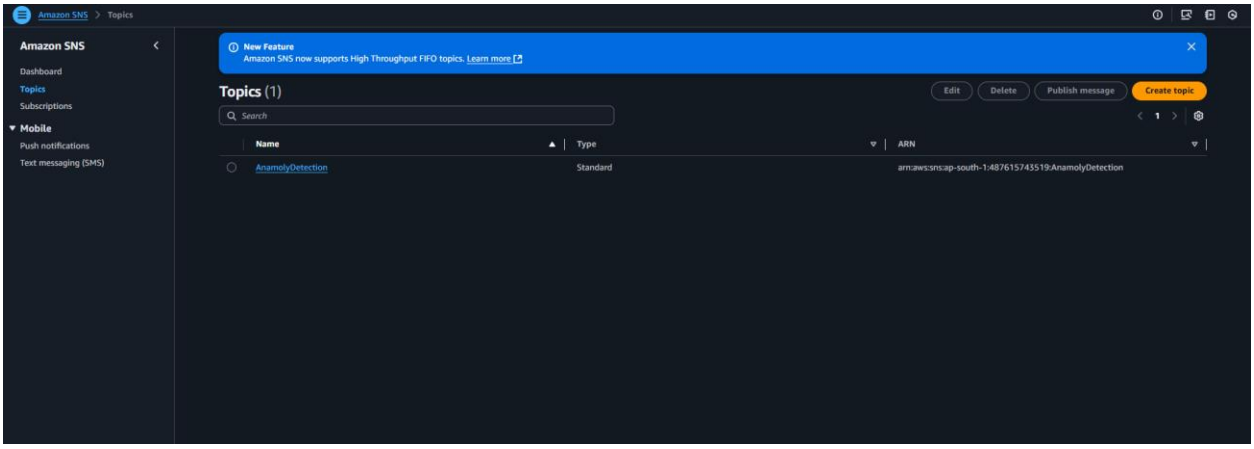
Data source (S3)→ Lambda → Rule-based check → SNS alert → Subscriber (email/SMS)

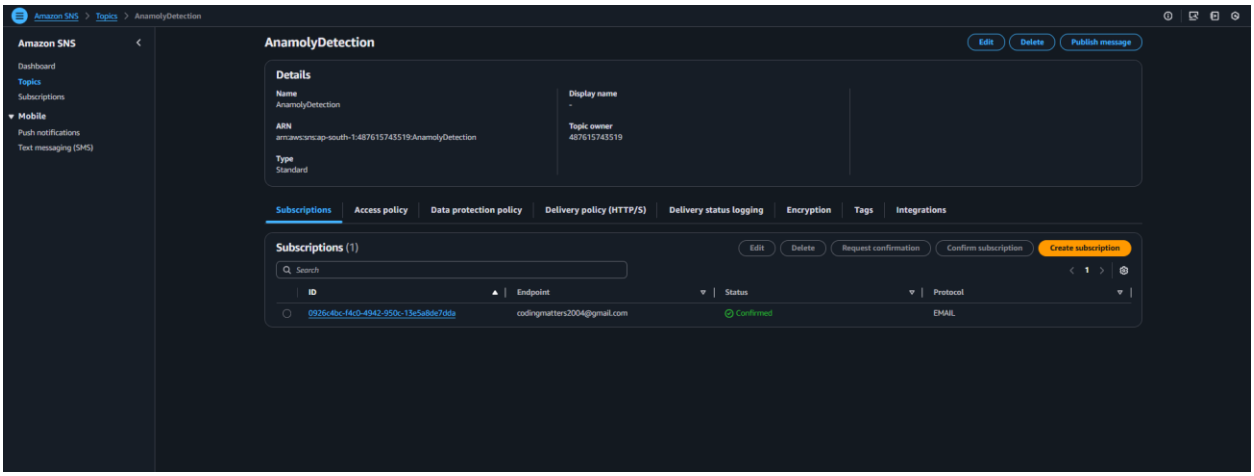
**Step 7: Create an SNS Topic**

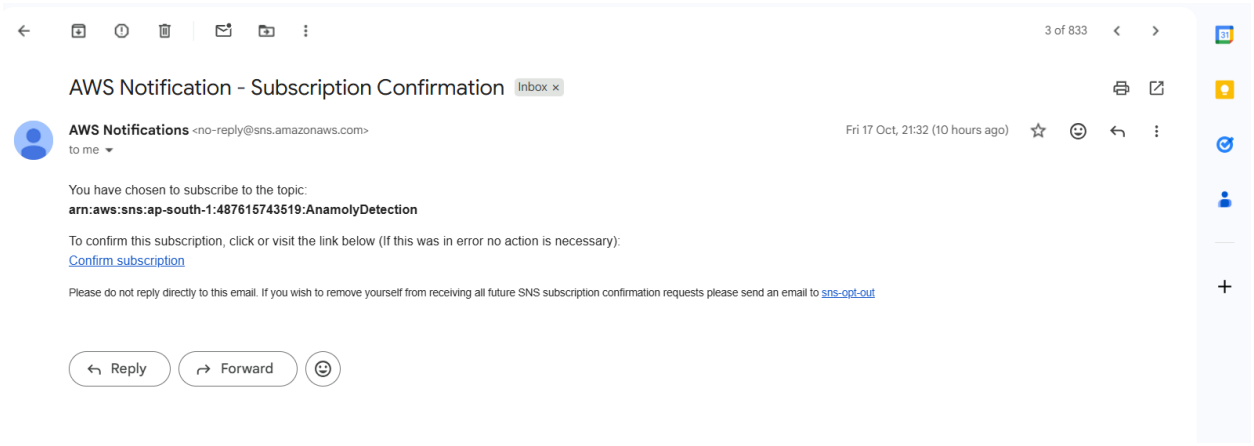
1. Go to AWS Console → Services → Simple Notification Service (SNS).
2. Click Topics → Create topic.
3. Select Standard topic.
4. Give it a name, e.g., AnamolyDetection
5. Click Create topic.
6. Add a subscription:
    - Protocol: Email
    - Endpoint: Your email address (or SMS)
7. Click Create subscription.
8. Check your email and confirm subscription.

**Purpose:** This step creates a centralized SNS topic that acts as a notification channel, allowing your Lambda function to publish a single alert message that is instantly broadcast to all subscribed users.





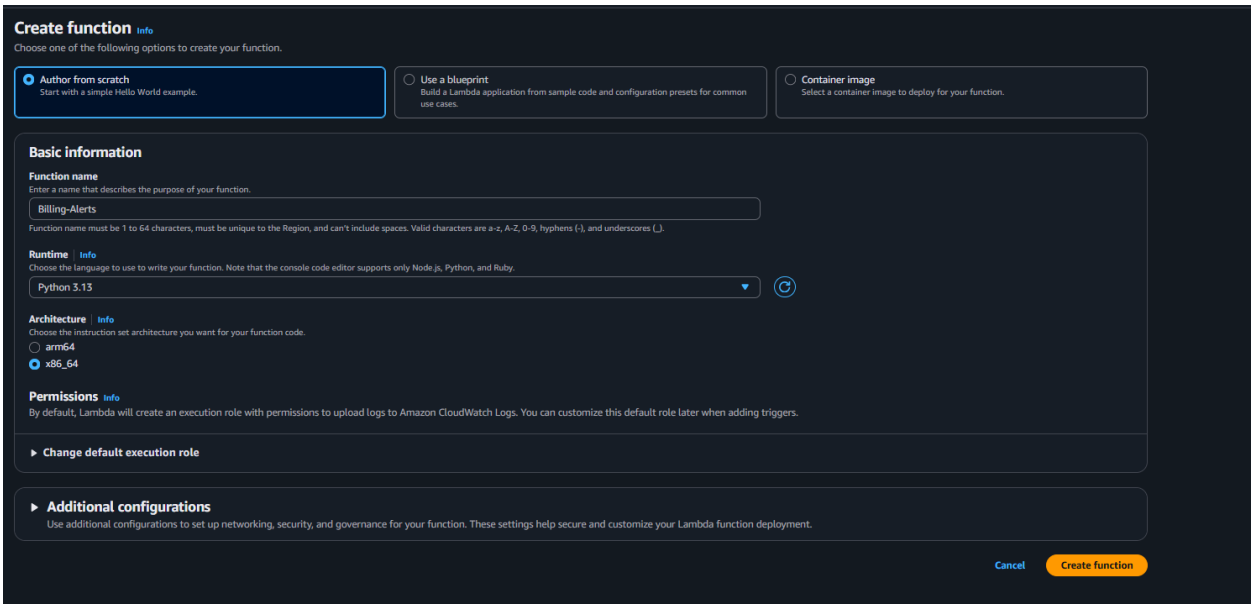

**Step 8: Create a Lambda Function and add a trigger to SNS**

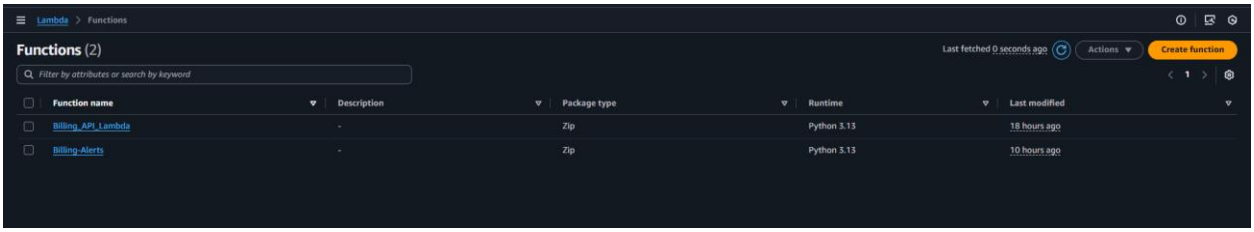
Create a Lambda Function

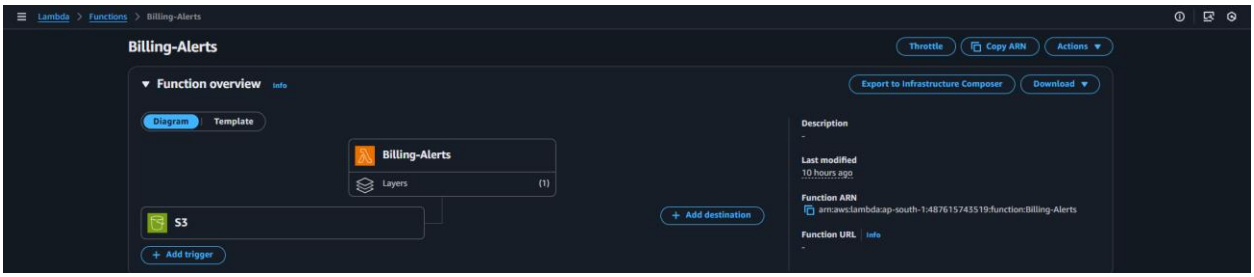
1. Go to AWS Console → Services → Lambda → Create function.
2. Choose Author from scratch.
    - Function name: Billing-Alerts
    - Runtime: Python 3.11 (or Node.js if you prefer)

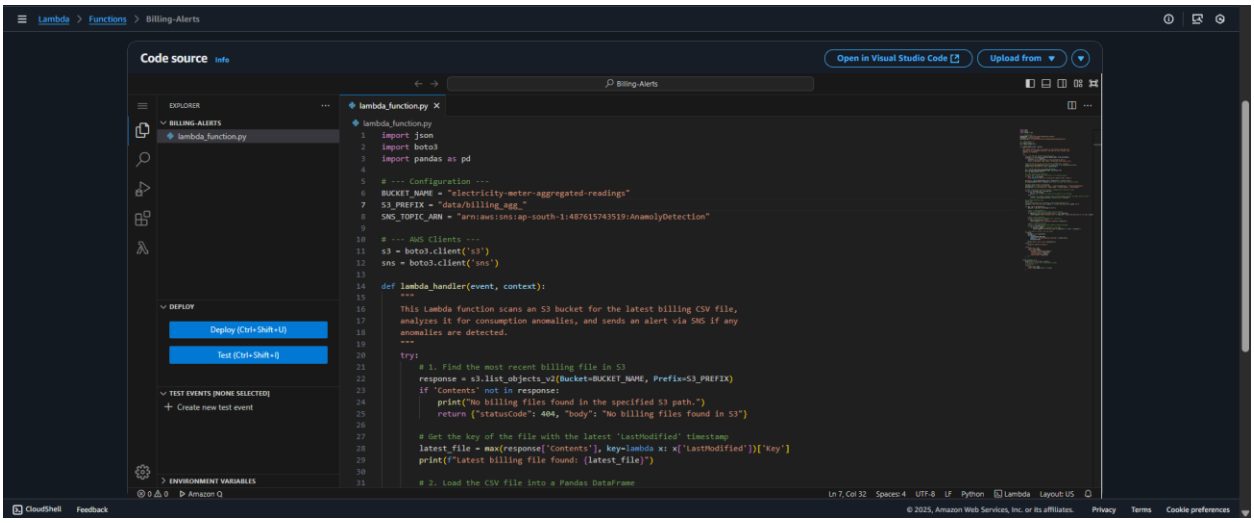
3. Click Create function.
4. Add S3 as trigger

**Purpose:** This Lambda function serves as the automated monitoring engine, containing the core logic to analyze the latest billing data from S3 and publish an alert to your SNS topic whenever an anomaly is detected.





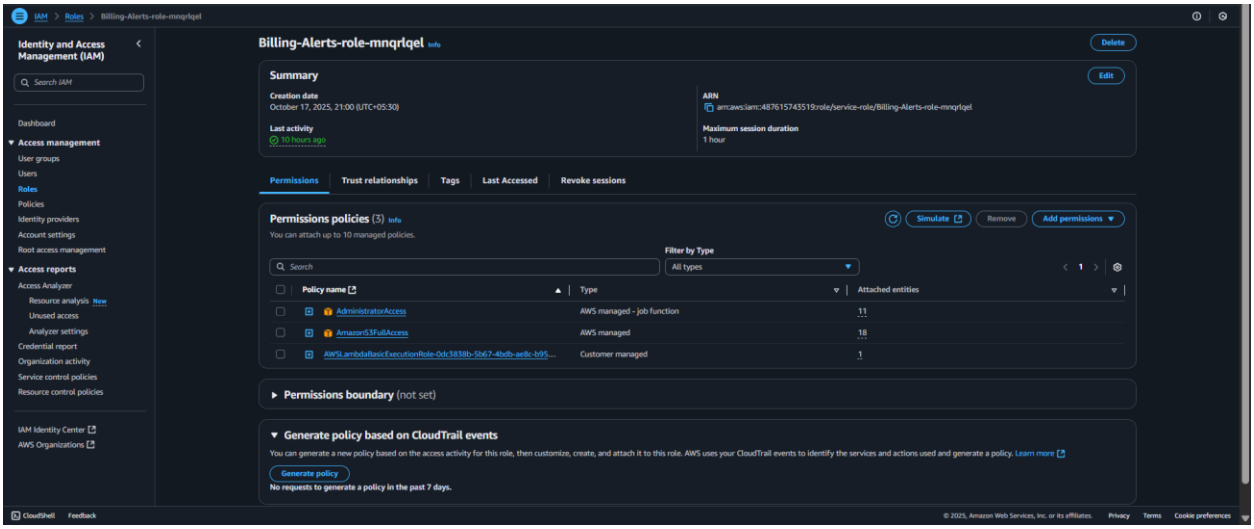

```python
import json
import boto3
import pandas as pd

# --- Configuration ---
BUCKET_NAME = "electricity-meter-aggregated-readings"
S3_PREFIX = "data/billing_agg_"
SNS_TOPIC_ARN = "arn:aws:sns:ap-south-1:487615743519:AnamolyDetection"

# --- AWS Clients ---
s3 = boto3.client('s3')
sns = boto3.client('sns')

def lambda_handler(event, context):
    """
    This Lambda function scans an S3 bucket for the latest billing CSV file,
    analyzes it for consumption anomalies, and sends an alert via SNS if any
    anomalies are detected.
    """
    try:
        # 1. Find the most recent billing file in S3
        response = s3.list_objects_v2(Bucket=BUCKET_NAME, Prefix=S3_PREFIX)
        if 'Contents' not in response:
            print("No billing files found in the specified S3 path.")
            return {"statusCode": 404, "body": "No billing files found in S3"}

        # Get the key of the file with the latest 'LastModified' timestamp
        latest_file = max(response['Contents'], key=lambda x: x['LastModified'])['Key']
        print(f"Latest billing file found: {latest_file}")

        # 2. Load the CSV file into a Pandas DataFrame
        obj = s3.get_object(Bucket=BUCKET_NAME, Key=latest_file)
        df = pd.read_csv(obj['Body'])

        # 3. Data Cleaning and Preparation
        # Ensure 'Date' column exists and convert it to datetime objects
        if 'Date' not in df.columns:
            raise ValueError("CSV file is missing the required 'Date' column.")

        df['Date'] = pd.to_datetime(df['Date'], dayfirst=True, errors='coerce')
        df.dropna(subset=['Date'], inplace=True) # Drop rows where date conversion failed

        # Define column groups for processing
        sub_meter_cols = ['total_Sub_metering_1', 'total_Sub_metering_2', 'total_Sub_metering_3']
```

```
        billing_cols = ['total_daily_sum', 'peak_charge', 'offpeak_charge', 'total_charge']

        # Safely convert data columns to numeric, creating them if they don't exist
        for col in sub_meter_cols + billing_cols:
            if col not in df.columns:
                df[col] = 0  # Create the column with a default of 0 if missing
            else:
                # Convert to numeric, forcing errors (like non-numeric strings) into NaN, then fill
with 0
                df[col] = pd.to_numeric(df[col], errors='coerce').fillna(0)

        # 4. Anomaly Detection Logic
        alerts = []
        # Calculate average daily consumption, handle empty DataFrame case
        avg_daily = df['total_daily_sum'].mean() if not df['total_daily_sum'].empty else 0

        for idx, row in df.iterrows():
            date_str = row['Date'].strftime('%d-%m-%Y')

            # Rule 1: High Consumption
            # Alert if consumption is 50% higher than the average
            if avg_daily > 0 and row['total_daily_sum'] > 1.5 * avg_daily:
                alerts.append(f"High consumption alert on {date_str}: {row['total_daily_sum']:.2f}
(Avg: {avg_daily:.2f})")

            # Rule 2: Zero Consumption
            # Alert if there was no consumption for a given day
            if row['total_daily_sum'] == 0:
                alerts.append(f"Zero consumption detected on {date_str}")

            # Rule 3: Sub-metering Spike
            # Alert if any sub-metering value exceeds a defined threshold
            for col in sub_meter_cols:
                if row[col] > 10000:  # Threshold is adjustable
                    alerts.append(f"Sub-metering spike on {date_str} in '{col}': {row[col]}")

        # 5. Send Alerts via SNS if any were found
        if alerts:
            message = "\n".join(alerts)
            sns.publish(
                TopicArn=SNS_TOPIC_ARN,
                Subject=f"⚠ Billing Anomalies Detected in {latest_file}",
                Message=message
            )
            print(f"Alerts sent to SNS: {len(alerts)}")
        else:
            print("No anomalies detected.")

        return {
            "statusCode": 200,
            "body": json.dumps({
                "message": "Processing complete.",
                "file_processed": latest_file,
                "records_analyzed": len(df),
                "alerts_found": len(alerts)
            })
        }

    except Exception as e:
        print(f"An error occurred: {str(e)}")
        # Optionally, send an error notification via SNS
        # sns.publish(...)
        return {
            "statusCode": 500,
```

```
            "body": json.dumps({"error": str(e)})
        }
```

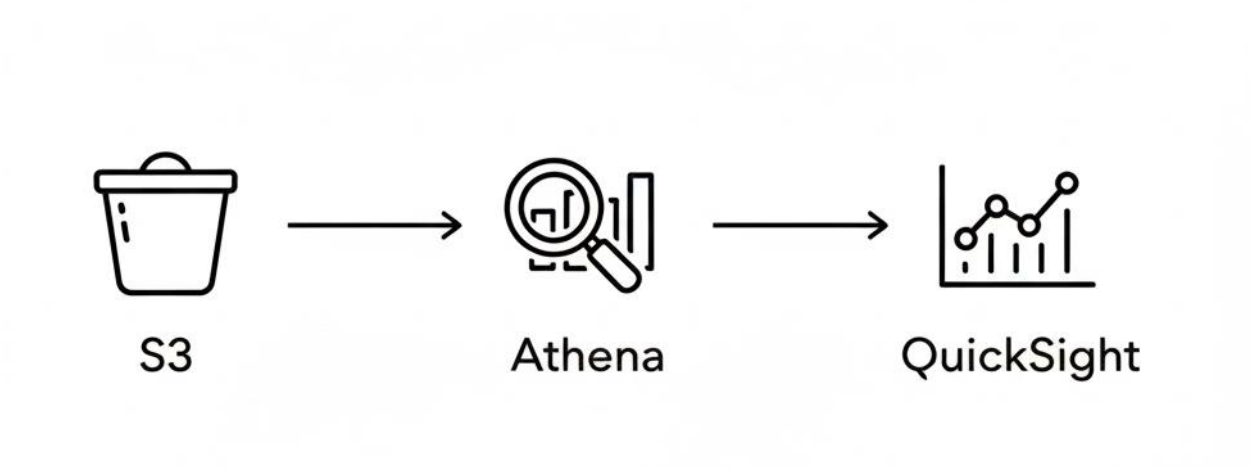
**Explanation:**

1. Read latest S3 file – Loads the newest billing CSV from the S3 bucket.
2. Clean data – Ensures ‘Date’ column exists and converts numeric columns safely.
3. Detect anomalies – Checks for high, zero, or spiking sub-metering values.
4. Send alerts – Publishes detected anomalies to the SNS topic.
5. Report results – Logs processed records, alerts sent, and handles errors.




**Data partitioning strategy to support fast queries and low cost:**

Flow Process: Data Store(S3) -> Billing_agg_->Athena→QuickSight/PowerBI




**Step 9: Use the Aggregated Data stored in S3 for processing the Athena.**

- Go to S3: Select the Bucket which you stored as meteringbucket1234-aggregated
- Select the aggregated dataset (i.e billing_agg_)
- Or create a folder as raw/ → upload your billing_agg_.csv file , easy for Athena process.
- Create a folder called clean → This folder is used to store the aggregated data file with datatype as double.

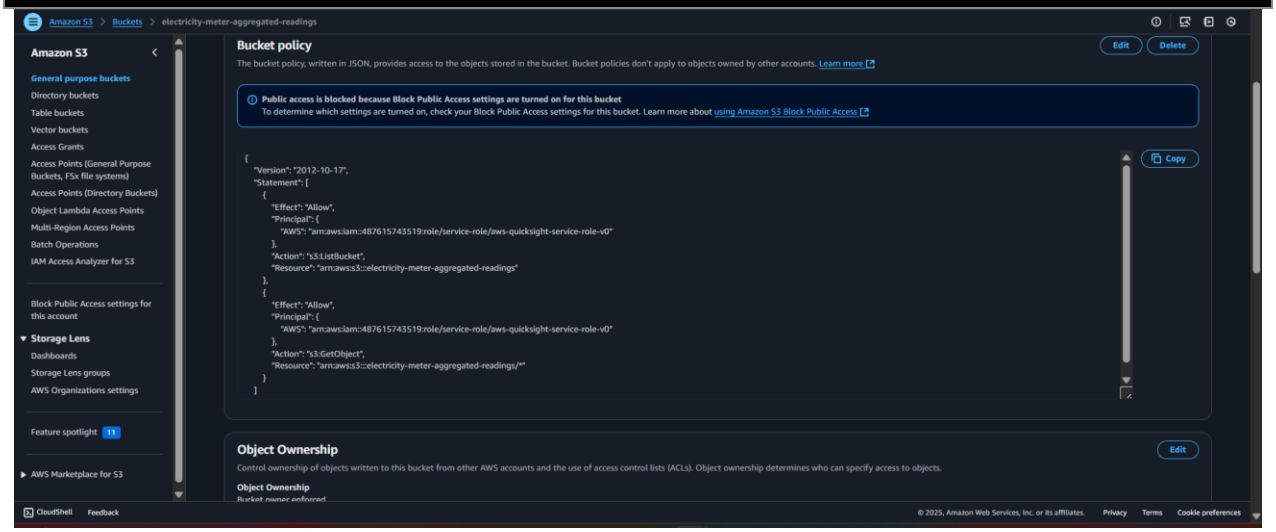
**Note:** The stored aggregated data file format is a string Data Type. Unable to perform operations in Athena.

**Step 10: Select Athena from the AWS console**

- Use Default workspace (permission error if we use new workspace in quicksight)
- Go to the query Editor. Choose a workspace Bucket.
- Choose the S3 bucket Location: To store the metadata information (Database information, Table schema, query information).
- Create a database: metering_db
- Create a table “energy_data_raw”, same as the aggregated data set. (Use data type format as string)

**Purpose:** The purpose is to use Athena as a translator, allowing Quicksight to use standard SQL to query and visualize your raw data files directly from Amazon S3.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::487615743519:role/service-role/aws-quicksight-service-role-v0"
            },
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::electricity-meter-aggregated-readings"
        },
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::487615743519:role/service-role/aws-quicksight-service-role-v0"
            },
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::electricity-meter-aggregated-readings/*"
        }
    ]
}
```

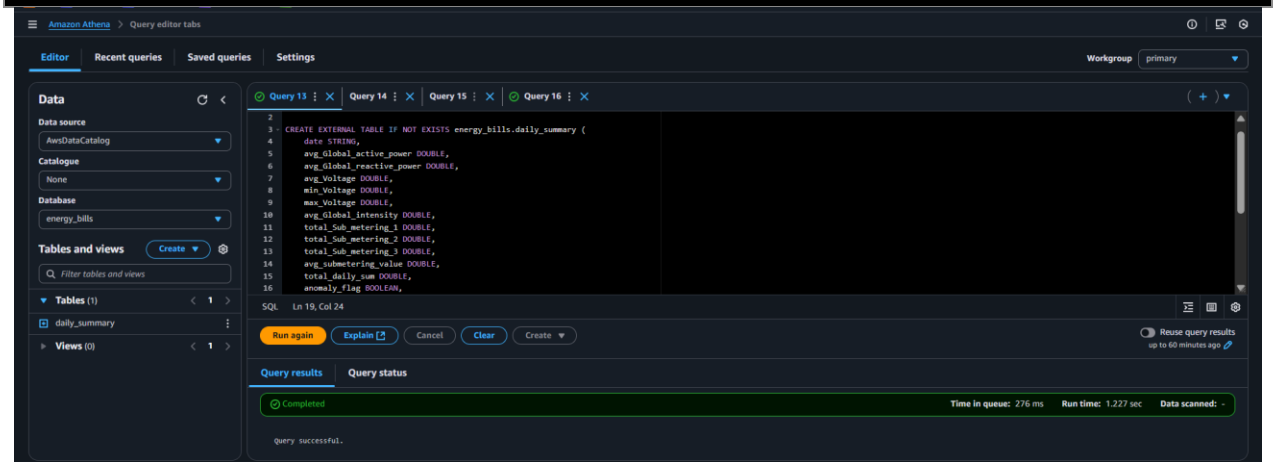



**Create a Database as “energy_bills”:**

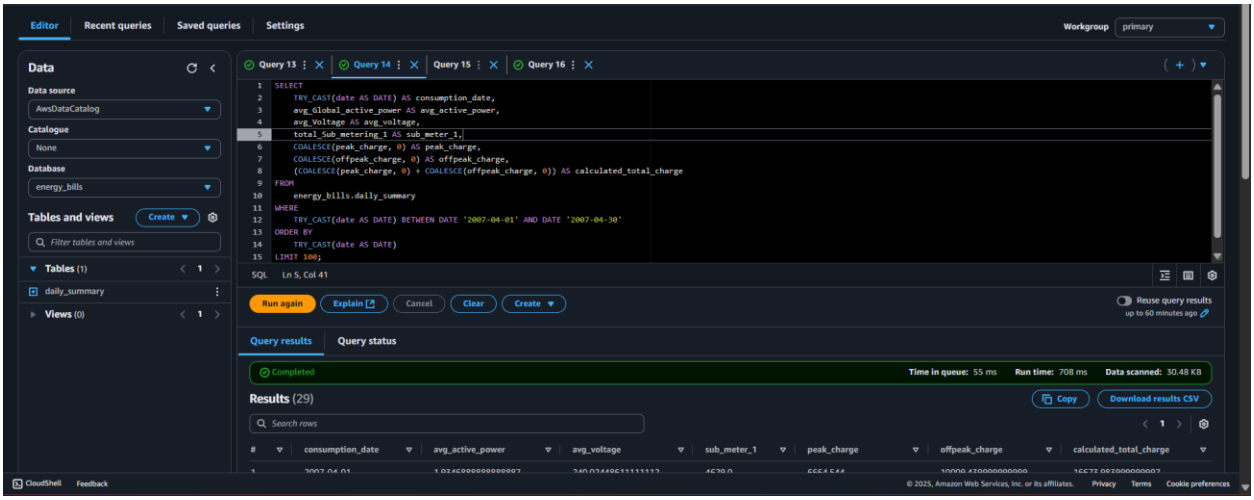
DROP TABLE IF EXISTS energy_bills.daily_summary;

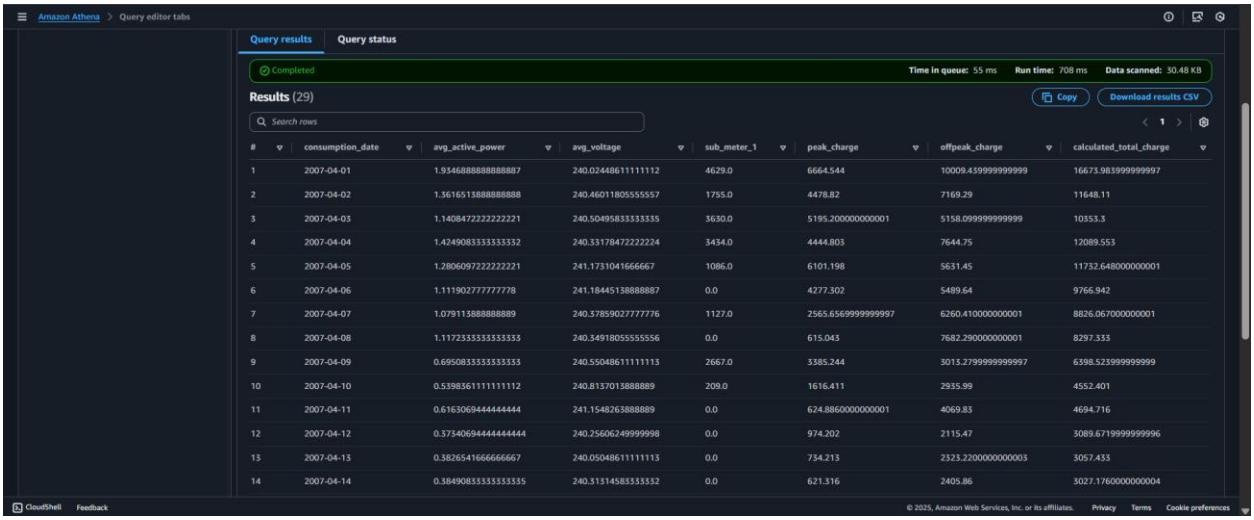
**Create Table daily_summary:**

```
CREATE EXTERNAL TABLE IF NOT EXISTS energy_bills.daily_summary (
    date STRING,
    avg_Global_active_power DOUBLE,
    avg_Global_reactive_power DOUBLE,
    avg_Voltage DOUBLE,
    min_Voltage DOUBLE,
    max_Voltage DOUBLE,
    avg_Global_intensity DOUBLE,
    total_Sub_metering_1 DOUBLE,
    total_Sub_metering_2 DOUBLE,
    total_Sub_metering_3 DOUBLE,
    avg_submetering_value DOUBLE,
    total_daily_sum DOUBLE,
    anomaly_flag BOOLEAN,
    peak_charge DOUBLE,
    offpeak_charge DOUBLE,
    total_charge DOUBLE
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar"     = "\""
)
LOCATION 's3://electricity-meter-aggregated-readings/data/'
TBLPROPERTIES ('skip.header.line.count'='1');
```



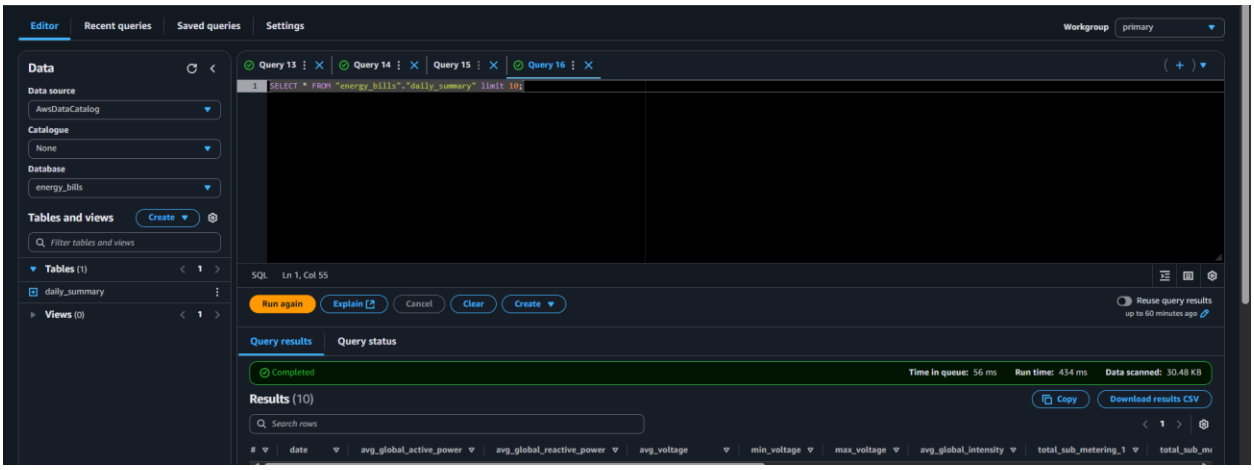

```
SELECT
    TRY_CAST(date AS DATE) AS consumption_date,
    avg_Global_active_power AS avg_active_power,
    avg_Voltage AS avg_voltage,
    total_Sub_metering_1 AS sub_meter_1,
    COALESCE(peak_charge, 0) AS peak_charge,
    COALESCE(offpeak_charge, 0) AS offpeak_charge,
    (COALESCE(peak_charge, 0) + COALESCE(offpeak_charge, 0)) AS calculated_total_charge
FROM
    energy_bills.daily_summary
WHERE
    TRY_CAST(date AS DATE) BETWEEN DATE '2007-04-01' AND DATE '2007-04-30'
ORDER BY
    TRY_CAST(date AS DATE)
LIMIT 100;
```

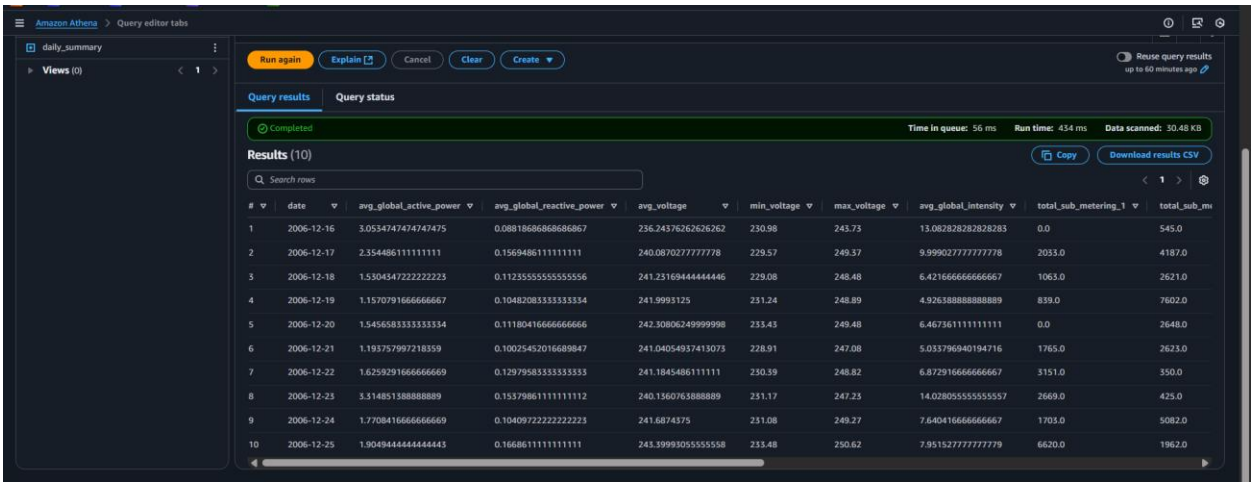



**Select all the items from the table**

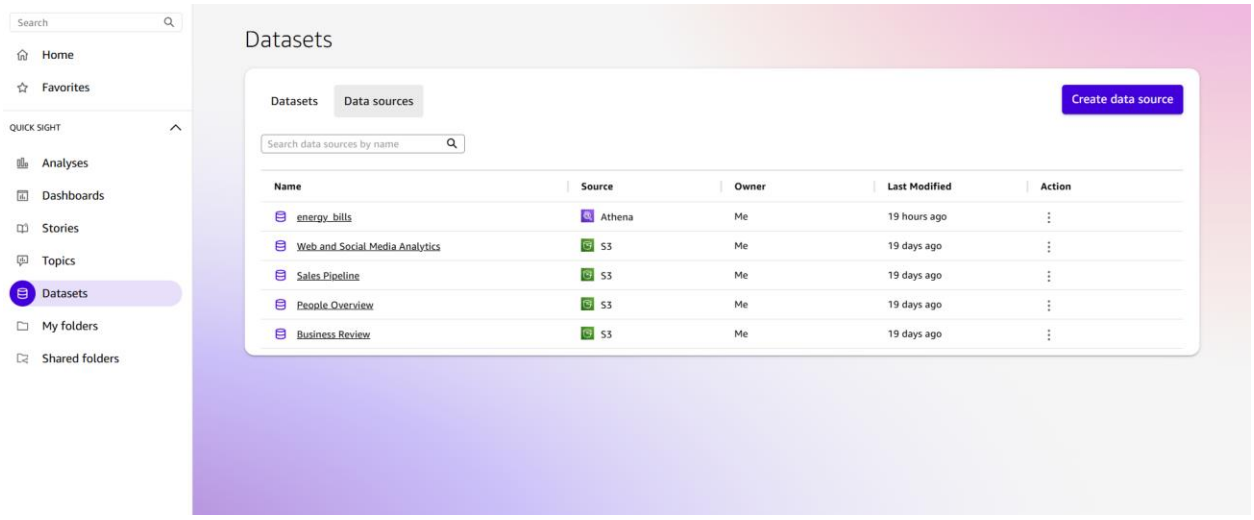
SELECT * FROM "energy_bills"."daily_summary" limit 10;



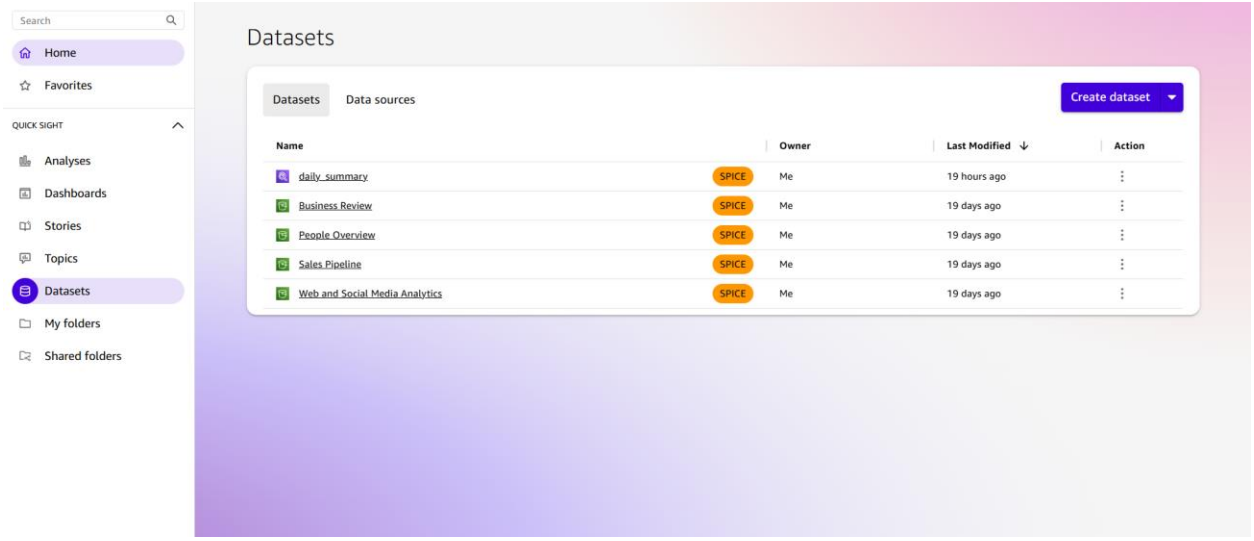



| # | date | avg_global_active_power | avg_global_reactive_power | avg_voltage | min_voltage | max_voltage | avg_global_intensity | total_sub_metering_1 | total_sub_m |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2006-12-16 | 3.0534747474747475 | 0.08818686868686867 | 236.24376262626262 | 230.98 | 243.73 | 13.082828282828283 | 0.0 | 545.0 |
| 2 | 2006-12-17 | 2.354486111111111 | 0.1569486111111111 | 240.0870277777778 | 229.57 | 249.37 | 9.999027777777778 | 2033.0 | 4187.0 |
| 3 | 2006-12-18 | 1.5304347222222223 | 0.11235555555555556 | 241.23169444444446 | 229.08 | 248.48 | 6.421666666666667 | 1063.0 | 2621.0 |
| 4 | 2006-12-19 | 1.1570791666666667 | 0.10482083333333334 | 241.9993125 | 231.24 | 248.89 | 4.926388888888889 | 839.0 | 7602.0 |
| 5 | 2006-12-20 | 1.5456583333333334 | 0.11180416666666666 | 242.30806249999998 | 233.43 | 249.48 | 6.467361111111111 | 0.0 | 2648.0 |
| 6 | 2006-12-21 | 1.193757997218359 | 0.10025452016689847 | 241.04054937413073 | 228.91 | 247.08 | 5.033796940194716 | 1765.0 | 2623.0 |
| 7 | 2006-12-22 | 1.6259291666666669 | 0.12979583333333333 | 241.1845486111111 | 230.39 | 248.82 | 6.872916666666667 | 3151.0 | 350.0 |
| 8 | 2006-12-23 | 3.314851388888889 | 0.15379861111111112 | 240.15607638888889 | 231.17 | 247.23 | 14.028055555555557 | 2669.0 | 425.0 |
| 9 | 2006-12-24 | 1.7708416666666669 | 0.10409722222222223 | 241.6874375 | 231.08 | 249.27 | 7.640416666666667 | 1703.0 | 5082.0 |
| 10 | 2006-12-25 | 1.9049444444444443 | 0.1668611111111111 | 243.39993055555558 | 233.48 | 250.62 | 7.951527777777779 | 6620.0 | 1962.0 |

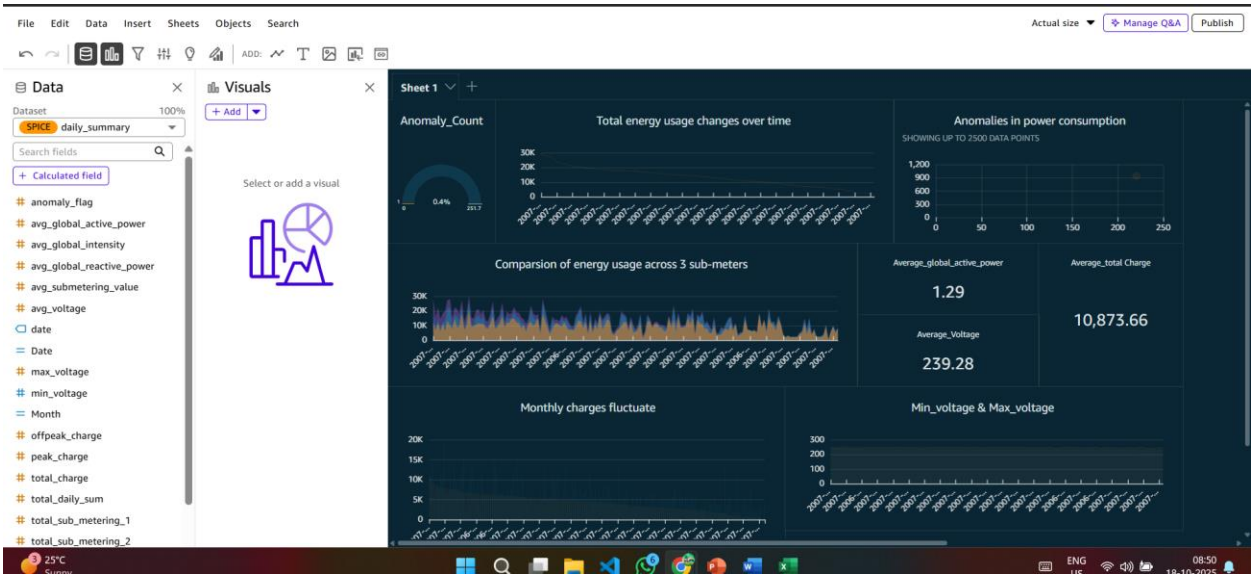
## Step 11: Quicksight Dashboards




| Name | Source | Owner | Last Modified | Action |
|---|---|---|---|---|
| energy_bills | Athena | Me | 19 hours ago | |
| Web and Social Media Analytics | S3 | Me | 19 days ago | |
| Sales Pipeline | S3 | Me | 19 days ago | |
| People Overview | S3 | Me | 19 days ago | |
| Business Review | S3 | Me | 19 days ago | |




| Name | | Owner | Last Modified | Action |
|---|---|---|---|---|
| daily_summary | SPICE | Me | 19 hours ago | |
| Business Review | SPICE | Me | 19 days ago | |
| People Overview | SPICE | Me | 19 days ago | |
| Sales Pipeline | SPICE | Me | 19 days ago | |
| Web and Social Media Analytics | SPICE | Me | 19 days ago | |




**Summary of Project:**

A scalable AWS-based system that automatically ingests, processes, and analyzes energy consumption data to generate bills, detect anomalies, and visualize trends — improving energy management and operational efficiency.

- Batch ingestion of energy meter readings.
- Data aggregation using Databricks.
- REST APIs for monthly bill generation.
- Rule-based anomaly detection and alerting system.
- Scalable and cost-efficient AWS architecture.
- Supports data-driven energy management decisions.