

## 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.

# ENERGY CONSUMPTION METERING



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.

**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.

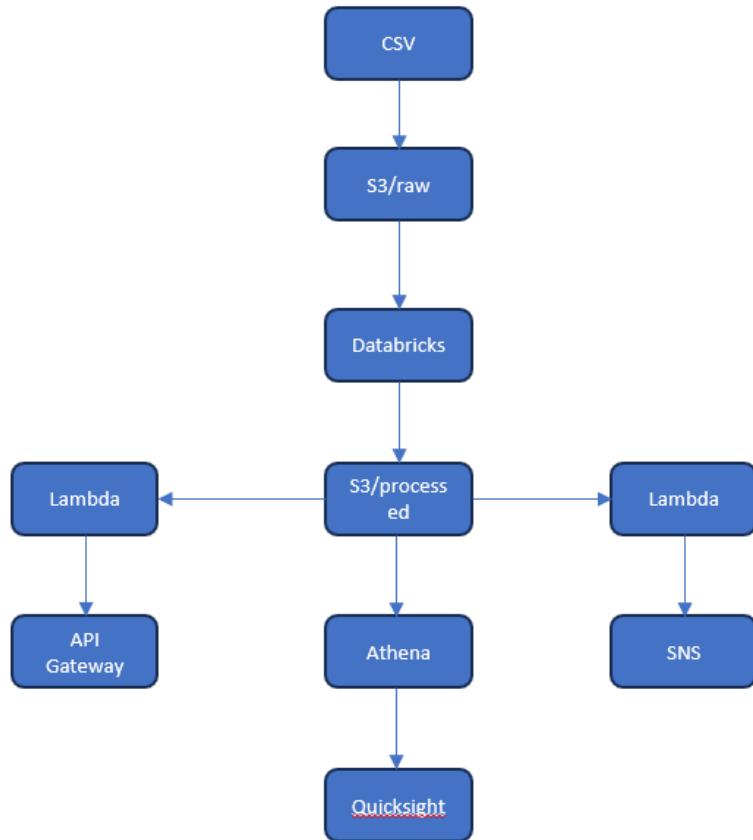
##### 4. Automated Alerting

- 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.

Name	AWS Region	Creation date
<a href="#">electricity-meter-aggregated-readings</a>	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 13:16:06 (UTC+05:30)
<a href="#">electricity-meter-readings</a>	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 13:15:40 (UTC+05:30)

**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	0	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
16-12-2006	17:27:00	5.388	0.502	233.74	23	0	1	17
16-12-2006	17:28:00	3.666	0.528	235.68	15.8	0	1	17
16-12-2006	17:29:00	3.52	0.522	235.02	15	0	2	17
16-12-2006	17:30:00	3.702	0.52	235.09	15.8	0	1	17
16-12-2006	17:31:00	3.7	0.52	235.22	15.8	0	1	17
16-12-2006	17:32:00	3.668	0.51	233.99	15.8	0	1	17
16-12-2006	17:33:00	3.662	0.51	233.86	15.8	0	2	16
16-12-2006	17:34:00	4.448	0.498	232.86	19.6	0	1	17
16-12-2006	17:35:00	5.412	0.47	232.78	23.2	0	1	17
16-12-2006	17:36:00	5.224	0.478	232.99	22.4	0	1	16
16-12-2006	17:37:00	5.268	0.398	232.91	22.6	0	2	17
16-12-2006	17:38:00	4.054	0.422	235.24	17.6	0	1	17
16-12-2006	17:39:00	3.384	0.282	237.14	14.2	0	0	17
16-12-2006	17:40:00	3.27	0.152	236.73	13.8	0	0	17
16-12-2006	17:41:00	3.43	0.156	237.06	14.4	0	0	17
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.

**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
Readings.csv	csv	October 17, 2025, 13:17:05 (UTC+05:30)	11.4 MB	Standard

**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.

**General purpose buckets (2) [Info](#)**

Buckets are containers for data stored in S3.

Name	AWS Region	Creation date
electricity-meter-aggregated-readings	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 13:16:06 (UTC+05:30)
electricity-meter-readings	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 13:15:40 (UTC+05:30)

**Account snapshot [Info](#)**  
View dashboard  
Updated daily  
Storage Lens provides visibility into storage usage and activity trends.

**External access summary - new [Info](#)**  
Updated daily  
External access findings help you identify bucket permissions that allow public access or access from other AWS accounts.

**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)
    
```

## Energy Consumption Metering | Yashwanth aravanti

```
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()
```

This screenshot shows a Databricks workspace with a notebook titled 'Untitled Notebook 2025-10-17 08:55:01'. The notebook contains Python code for downloading a CSV file from an S3 bucket. A tooltip message 'Your serverless compute is disconnected.' is visible in the top right corner.

```
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='AKIAJXOCB7JYPPYNKAHNCZ',
        aws_secret_access_key='9ko5auTasom3ihHbx+LwNkTaemDaxX1S1FjF6521'.
```

This screenshot shows the same Databricks workspace after running the notebook. The output cell shows the success message 'Success! File downloaded and saved to: metering\_csv/downloaded\_metering\_data.csv'. A tooltip message 'Your serverless compute is disconnected.' is visible in the top right corner.

```
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()

Attempting to download s3://electricity-meter-readings/Readings.csv...
✅ Success! File downloaded and saved to: metering_csv/downloaded_metering_data.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()

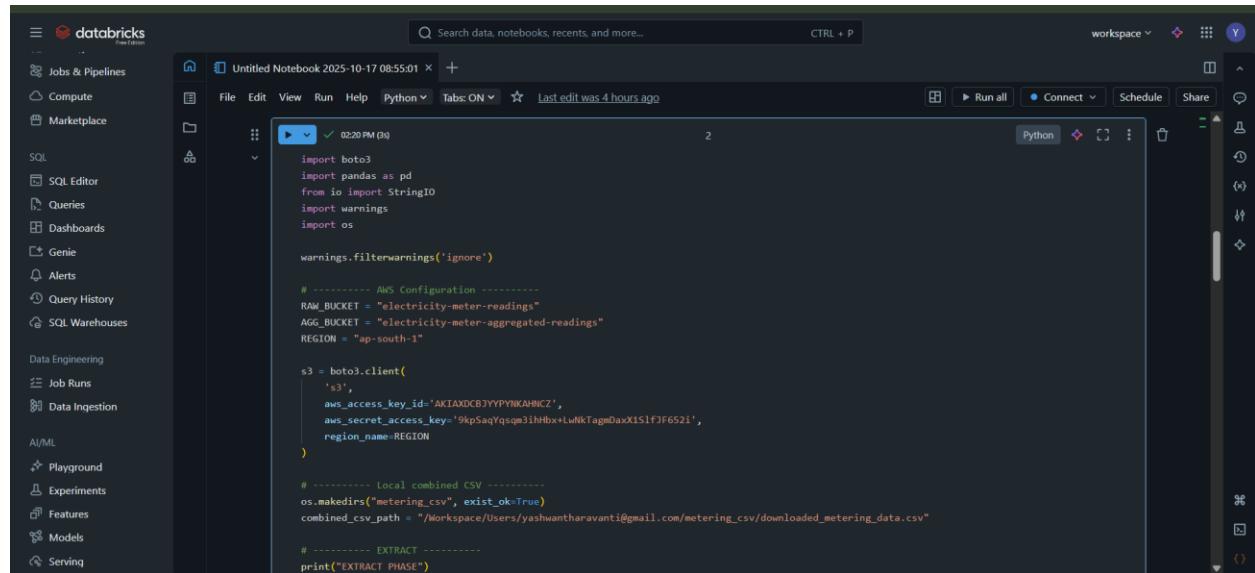
# ----- LOAD: Upload to S3 -----
print("Uploading final aggregated ETL file to S3")
csv_buffer = StringIO()
final_agg.to_csv(csv_buffer, index=False)

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.

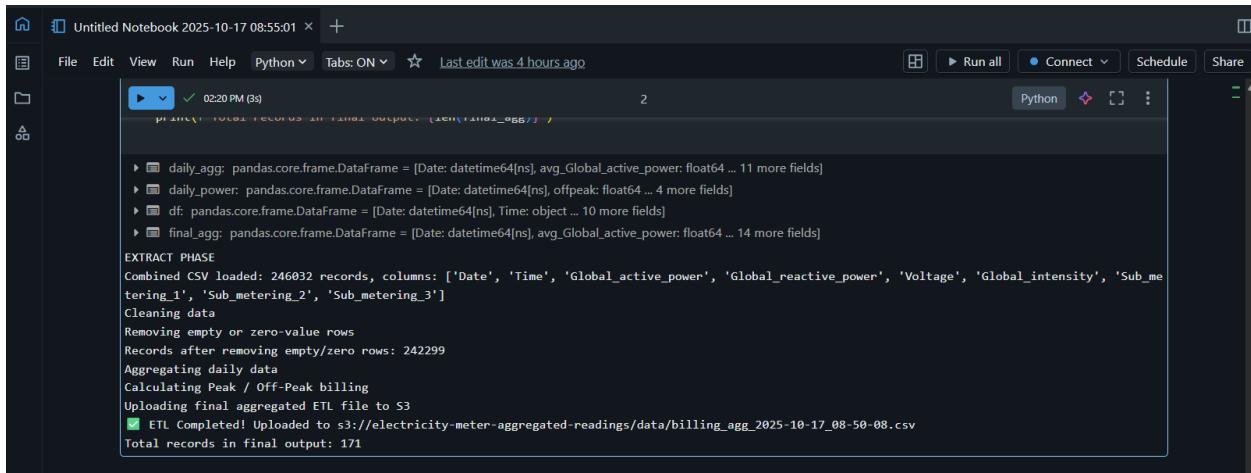
- It combines the Date and Time columns into a single DateTime column.
- It extracts the hour of the day to determine if a reading occurred during:
  - **Peak hours** (6 PM – 10 PM)
  - **Off-peak hours** (all other times)
- It applies different billing rates to the energy consumed in each period:
  - **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:



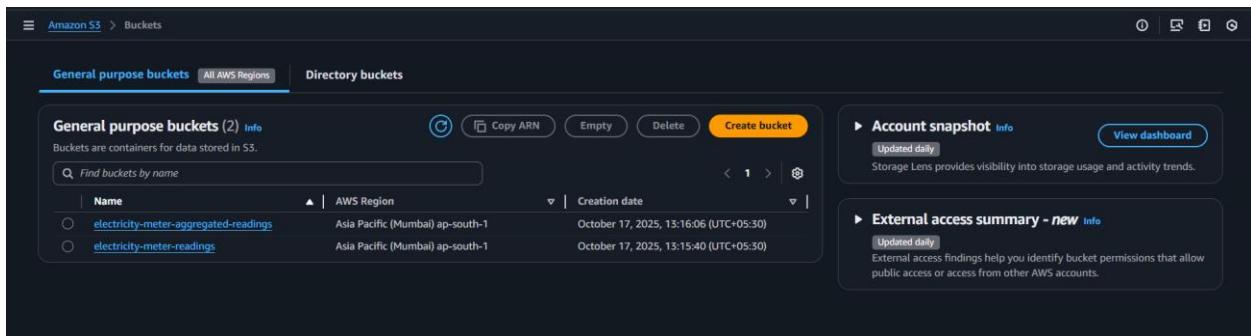
```

[2]: daily_agg: pandas.core.frame.DataFrame = [Date: datetime64[ns], avg_Global_active_power: float64 ... 11 more fields]
[2]: daily_power: pandas.core.frame.DataFrame = [Date: datetime64[ns], offpeak: float64 ... 4 more fields]
[2]: df: pandas.core.frame.DataFrame = [Date: datetime64[ns], Time: object ... 10 more fields]
[2]: final_agg: pandas.core.frame.DataFrame = [Date: datetime64[ns], avg_Global_active_power: float64 ... 14 more fields]

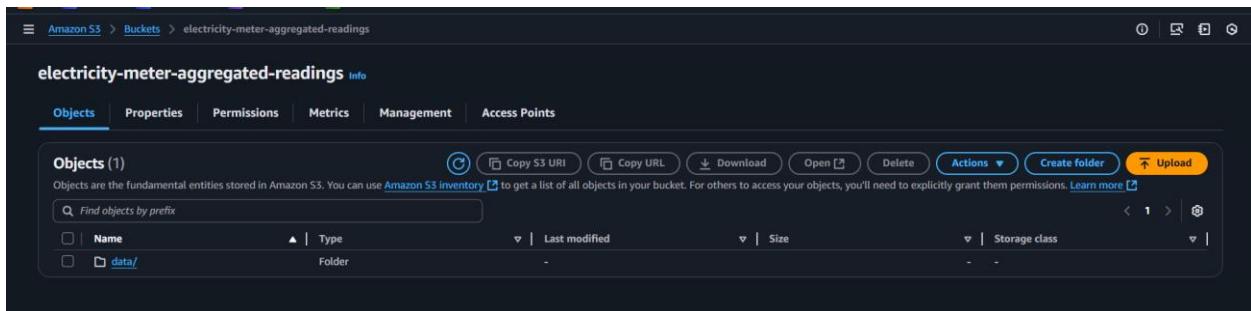
EXTRACT PHASE
Combined CSV loaded: 246032 records, columns: ['Date', 'Time', 'Global_active_power', 'Global_reactive_power', 'Voltage', 'Global_intensity', 'Sub_metering_1', 'Sub_metering_2', 'Sub_metering_3']
Cleaning data
Removing empty or zero-value rows
Records after removing empty/zero rows: 242299
Aggregating daily data
Calculating Peak / Off-Peak billing
Uploading final aggregated ETL file to S3
ETL Completed! Uploaded to s3://electricity-meter-aggregated-readings/data/billing_agg_2025-10-17_08-50-08.csv
Total records in final output: 171

```

### Storing the Aggregated Data in S3:



Name	AWS Region	Creation date
<a href="#">electricity-meter-aggregated-readings</a>	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 15:16:06 (UTC+05:30)
<a href="#">electricity-meter-readings</a>	Asia Pacific (Mumbai) ap-south-1	October 17, 2025, 15:15:40 (UTC+05:30)



Name	Type	Last modified	Size	Storage class
<a href="#">data/</a>	Folder	-	-	-

## Energy Consumption Metering | Yashwanth aravanti

Amazon S3 > Buckets > electricity-meter-aggregated-readings > data/

**data/**

**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](#)

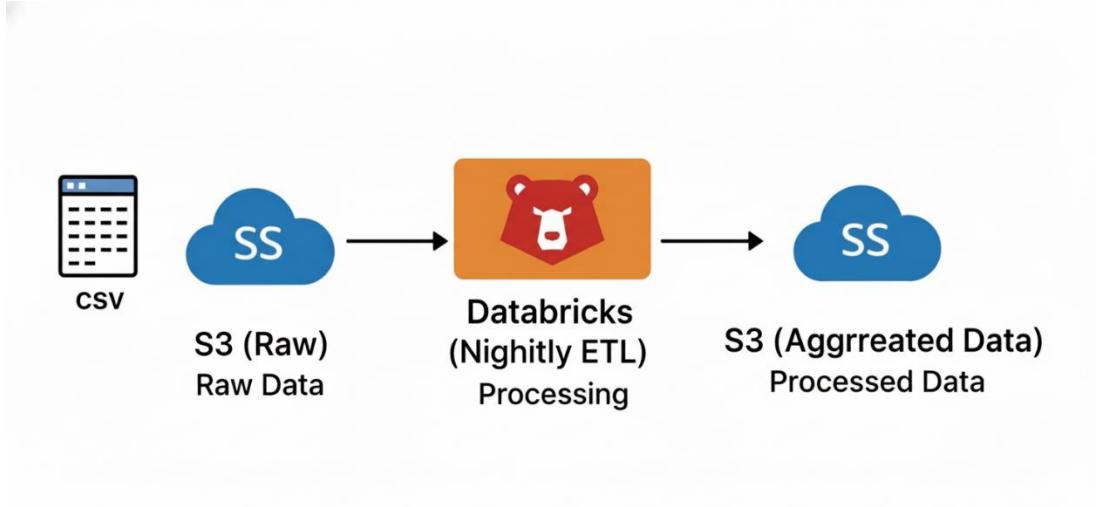
**Find objects by prefix**

Name	Type	Last modified	Size	Storage class
<a href="#">billing_agg_2025-10-17_08-50-08.csv</a> csv		October 17, 2025, 14:20:09 (UTC+05:30)	30.5 KB	Standard

S13

**billing\_agg\_2025-10-17\_08-50-08.csv**

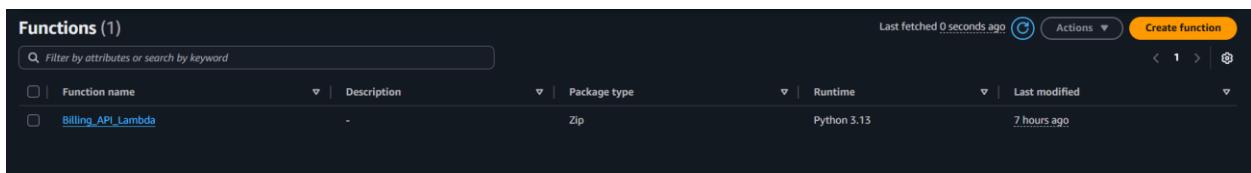
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1	Date	avg_Globa	avg_Volta	avg_Volta	min_Volta	max_Volta	total_Sub	total_charge														
2	16-12-2006	3.053475	0.08187	236.2438	230.98	243.73	13.08283	0	545	4926	1823.667	5471	FALSE	6812.308	2038.64	8850.948						
3	17-12-2006	2.354486	0.156949	240.087	229.57	249.37	9.999028	2033	4187	13341	6520.333	19561	FALSE	6633.961	13049.97	19683.93						
4	18-12-2006	3.50435	0.112356	241.2317	229.08	248.48	6.421667	1063	2621	14018	5900.667	17702	FALSE	5221.04	7947.93	13168.97						
5	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						
6	20-12-2006	1.545658	0.111804	242.3083	233.43	249.48	6.467361	0	2648	14065	5570.333	16711	FALSE	6786.57	7136.64	13923.21						
7	21-12-2006	1.193758	0.100255	241.0403	228.91	247.08	5.033797	1765	2623	10421	4936.333	14809	FALSE	4110.634	6165.1	10275.73						
8	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						
9	23-12-2006	3.314851	0.153799	240.1361	231.17	247.23	14.02806	2669	425	14726	5940	17820	FALSE	7888.202	19226.81	27115.01						
10	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						
11	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						
12	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						
13	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						
14	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						
15	29-12-2006	2.204439	0.136385	240.8713	228.2	250.55	9.271389	1252	5162	11329	5914.333	17743	FALSE	5849.461	12431.1	18280.56						
16	30-12-2006	1.939709	0.216918	240.5407	228.99	250.67	8.292211	3072	7891	12480	7814.333	23443	FALSE	6492.196	10127.57	16619.77						
17	31-12-2006	2.426525	0.104758	239.8966	231.23	248.56	10.199598	0	347	6502	2283	6849	FALSE	4953.781	14556.99	19510.77						
18	01-01-2007	1.909031	0.102893	240.129	234.07	246.3	7.916944	0	352	5884	2077.333	6232	FALSE	1550.621	12832.89	14383.51						
19	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						
20	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						
21	04-01-2007	2.263481	0.140488	239.503	231.83	247.35	9.478194	1051	7597	10896	6514.667	19544	FALSE	3792.819	14065.99	17858.81						
22	05-01-2007	1.884281	0.120246	240.4958	230.48	248.18	7.885139	1483	379	7602	3154.667	9464	FALSE	5145.181	10540.24	15685.43						
23	06-01-2007	1.047485	0.133464	239.6731	226.32	247.98	4.543889	1336	402	5678	2472	7416	FALSE	2405.781	6126.72	8532.509						
24	07-01-2007	1.699736	0.170885	240.9025	227.85	248.98	7.282639	1987	8177	12810	7656	22974	FALSE	8358.679	7321.23	15679.91						
25	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						
26	09-01-2007	1.297954	0.143922	240.4875	230.61	247.28	5.521111	1688	4267	979	5248.333	15745	FALSE	5693.436	5996.19	11689.63						
27	10-01-2007	1.628801	0.162657	241.9274	231.95	247.01	6.361129	2770	4465	11012	54103	16230	FALSE	6792.465	7110.02	13103.49						



**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.



The screenshot shows the AWS Lambda console interface. At the top, there's a header with 'Last fetched 0 seconds ago' and a 'Create function' button. Below the header is a search bar labeled 'Filter by attributes or search by keyword'. A table lists one function: 'Billing\_APILambda'. The table has columns for 'Function name', 'Description', 'Package type', 'Runtime', and 'Last modified'. The 'Function name' column shows 'Billing\_APILambda', 'Description' is '-', 'Package type' is 'Zip', 'Runtime' is 'Python 3.13', and 'Last modified' is '7 hours ago'. There are also 'Actions' and a refresh icon in the top right of the table area.

Functions (1)					
Function name	Description	Package type	Runtime	Last modified	Actions
Billing_APILambda	-	Zip	Python 3.13	7 hours ago	

**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)
    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"error": str(e)})
        }
    else:
        return {
            "statusCode": 200,
            "body": json.dumps({"month": month, "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.

Code | Test | Monitor | Configuration | Aliases | Versions

Code source Info

EXPLORER

- BILLING\_API\_LAMBDA
  - lambda\_function.py

DEPLOY

- Deploy (Ctrl+Shift+U)
- Test (Ctrl+Shift+I)

TEST EVENTS [NONE SELECTED]

- + Create new test event

ENVIRONMENT VARIABLES

Open in Visual Studio Code ▾ Upload from ▾

```

lambda_function.py

# ----- S3 Configuration -----
BUCKET_NAME = "electricity-meter-aggregated-readings"
S3_PREFIX = "data/billing_egg"
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"})
            }
    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"error": str(e)})
        }

```

Ln 21, Col 42 Spaces: 4 UTF-8 LF Python Lambda Layout: US

Code | Test | Monitor | Configuration | Aliases | Versions

Code source Info

EXPLORER

- BILLING\_API\_LAMBDA
  - lambda\_function.py

DEPLOY

- Deploy (Ctrl+Shift+U)
- Test (Ctrl+Shift+I)

TEST EVENTS [NONE SELECTED]

- + Create new test event

ENVIRONMENT VARIABLES

Open in Visual Studio Code ▾ Upload from ▾

```

lambda_function.py

# ----- Lambda Handler -----
def lambda_handler(event, context):
    try:
        # Parse date from query string
        month = (event.get('queryStringParameters') or {}).get('month')
        df = pd.read_csv(monthly_df, parse_dates=['Date'], dayfirst=True, errors='coerce')
        if df.empty:
            return {
                "statusCode": 500,
                "body": json.dumps({"error": "No valid dates found in CSV"})
            }

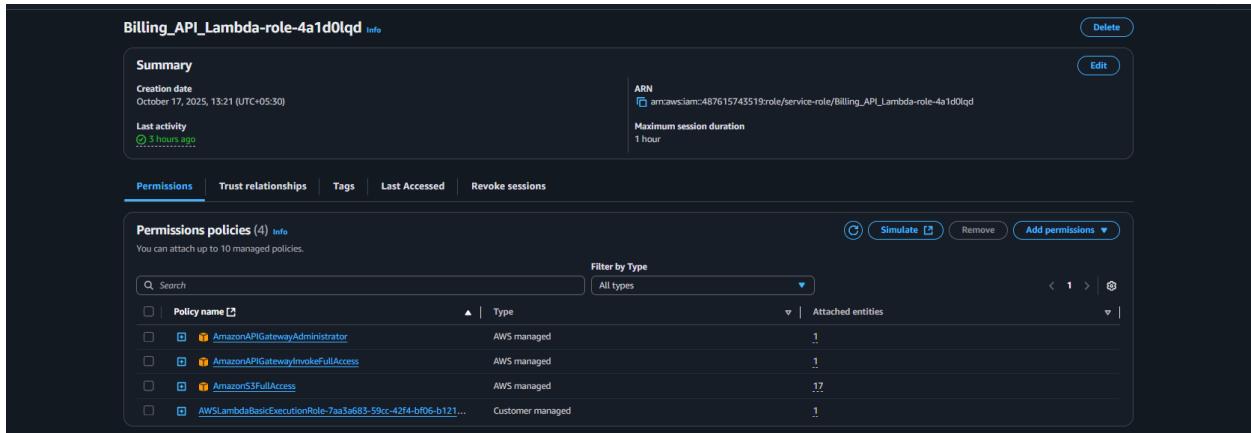
        # Fill missing billing columns with 0
        billing_cols = [
            'total_Sub_metering_1', 'total_Sub_metering_2', 'total_Sub_metering_3',
            'avg_submetering_value', 'total_daily_sum',
            'peak_charge', 'offpeak_charge', 'total_charge'
        ]
        for col in billing_cols:
            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]
    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"error": str(e)})
        }

```

Ln 21, Col 42 Spaces: 4 UTF-8 LF Python Lambda Layout: US



## 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.

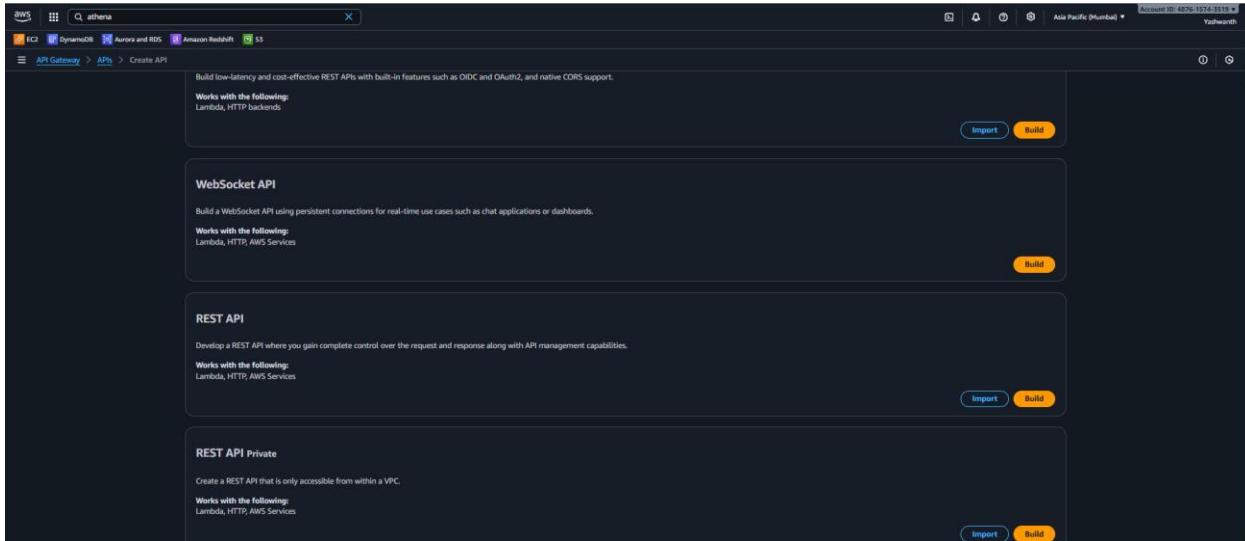
1. Click the **Actions** menu one last time and select **Deploy API**.
2. In the pop-up window, select **[New Stage]** for the **Deployment stage**.
3. Enter bills as the **Stage name**.
4. Click **Deploy**.
5. You'll be taken to the Stage Editor. In the navigation tree on the left, click on the **GET** 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](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.



### Create REST API Info

**API details**

**New API**  
Create a new REST API.

**Clone existing API**  
Create a copy of an API in this AWS account.

**Import API**  
Import an API from an OpenAPI definition.

**Example API**  
Learn about API Gateway with an example API.

**API name**  
MYAPI

**Description - optional**

**API endpoint type**  
Regional APIs are deployed in the current AWS Region. Edge-optimized APIs route requests to the nearest CloudFront Point of Presence. Private APIs are only accessible from VPCs.

Regional

**IP address type** | Info  
Select the type of IP addresses that can invoke the default endpoint for your API.

**IPv4**  
Supports only edge-optimized and Regional API endpoint types.

**Dualstack**  
Supports all API endpoint types.

**Cancel** **Create API**

# Energy Consumption Metering | Yashwanth aravanti

The screenshot shows the AWS API Gateway interface. On the left, the navigation bar includes 'APIs', 'Custom domain names', 'Domain name access associations', 'VPC links', and a section for 'API: MYAPI' which contains 'Resources', 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy documentation', 'Dashboard', and 'API settings'. Below these are sections for 'Usage plans', 'API keys', 'Client certificates', and 'Settings'. The main panel displays 'Resources' with a 'Create resource' button and a table for 'Method details'. The table shows a single entry for 'OPTIONS' with a path of '/'. The 'Integration type' is set to 'Mock' and 'Authorization' is 'None'. Buttons for 'Update documentation' and 'Enable CORS' are at the top right, along with 'API actions' and 'Deploy API' dropdowns.

A green success message at the top states: 'Successfully created REST API 'MYAPI' (a37c03y02t6)'. The dialog is titled 'Enable CORS' and contains the following sections:

- CORS settings**: A note explaining CORS allows requests from scripts running in the browser. It includes a 'Gateway responses' section where 'Default 4XX' and 'Default 5XX' are checked.
- Access-Control-Allow-Methods**: Shows 'OPTIONS' selected.
- Access-Control-Allow-Headers**: Shows 'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token' selected.
- Access-Control-Allow-Origin**: Shows a field containing '\*'.
- Additional settings**: A section for granting API Gateway permission to invoke a Lambda function.

At the bottom are 'Cancel' and 'Save' buttons.

The dialog is titled 'Create method' and shows 'Method details' for a 'GET' request. The 'Integration type' is set to 'Lambda function'. Other options shown include 'HTTP', 'Mock', 'AWS service', 'Lambda proxy integration', and 'VPC link'. Under 'Lambda function', the ARN is listed as 'arn:aws:lambda:ap-south-1:148761574351:function:Billing\_API\_Lam'. A note says 'Grant API Gateway permission to invoke your Lambda function'. At the bottom, 'Integration timeout' is set to '29000' milliseconds.

# Energy Consumption Metering | Yashwanth aravanti

The screenshot shows the 'Create method' dialog for an API named 'MYAPI'. The 'Lambda proxy integration' option is selected. A dropdown menu shows 'ap-south-1' and a search bar with 'arn:aws:lambda:ap-south-1:487615743519:function:Billing\_API\_Lam'. Below this, a button 'Grant API Gateway permission to invoke your Lambda function' is visible. An 'Integration timeout' field is set to 29000 ms. The 'Method request settings' section includes a 'URL query string parameters' table with one entry: 'month' (Required). The 'HTTP request headers' and 'Request body' sections are collapsed. At the bottom right are 'Cancel' and 'Create method' buttons.

The screenshot shows the 'Resources' page for the 'MYAPI' API. A green success message 'Successfully created method 'GET' in '/'. Redeploy your API for the update to take effect.' is displayed. The 'GET' method under resource '/' is selected. The 'Method execution' diagram shows a flow from 'Client' to 'Method request' to 'Integration request' to 'Lambda integration', and back from 'Integration response' to 'Method response' to 'Client'. Below the diagram, tabs for 'Method request', 'Integration request', 'Integration response', 'Method response', and 'Test' are shown. The 'Method request settings' section indicates 'NONE' for Authorization and 'False' for API key required. The 'Request paths' section shows a single path with 'Name' and 'Caching' options. At the top right, there are 'API actions' and 'Deploy API' buttons.

The screenshot shows the 'Resources' page for the 'MYAPI' API. A green success message 'Successfully created method 'GET' in '/'. Redeploy your API for the update to take effect.' is displayed. The 'GET' method under resource '/' is selected. A modal dialog titled 'Deploy API' is open, prompting to choose a deployment stage ('Choose an option') and provide a 'Deployment description'. The 'Method execution' diagram and 'Method request settings' section are visible in the background. The 'Request paths' section shows a single path with 'Name' and 'Caching' options. At the top right, there are 'API actions' and 'Deploy API' buttons.

## Energy Consumption Metering | Yashwanth aravanti

The screenshot shows the AWS API Gateway interface. On the left, the navigation bar includes 'APIs', 'Custom domain names', 'Domain name access associations', and 'VPC links'. Under 'APIs', 'API: MYAPI' is expanded, showing 'Resources', 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings'. The 'Resources' section lists a single resource: '/ - GET' and 'OPTIONS'. A modal window titled 'Deploy API' is open, prompting the user to '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.' A dropdown menu shows 'New stage' selected, with 'Stage name' set to 'bill'. Below this, there's a note: 'A new stage will be created with the default settings. Edit your stage settings on the Stage page.' A 'Deployment description' field is present, though empty. The 'Method' dropdown shows 'Method: / - GET' and 'Auth: NONE'. The 'Request paths' section is empty. At the bottom right of the modal is a yellow 'Deploy' button. The status bar at the bottom indicates 'Successfully created method GET in /; redeploy your API for the update to take effect.'

This screenshot shows the 'Stages' section of the AWS API Gateway. It displays a tree structure with 'new\_stage' as the root node, which contains a single child node '/ - GET'. A green notification bar at the top states 'Successfully created deployment for MYAPI. This deployment is active for new\_stage.' On the right side of the screen, a modal window titled 'Method overrides' is open, stating 'By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.' It shows a note: 'This method inherits its settings from the 'new\_stage' stage.' Below this, a 'Copied' message is shown with a link to the Lambda function: 'Copied https://s8bb871sgb.execute-api.ap-south-1.amazonaws.com/new\_stage/bills'. The status bar at the bottom indicates 'Successfully created deployment for MYAPI. This deployment is active for new\_stage.'

This screenshot shows the 'Configuration' tab for the 'Billing\_API\_Lambda' function in the AWS Lambda console. The left sidebar lists 'Code', 'Test', 'Monitor', 'Configuration', 'Aliases', and 'Versions'. The 'Configuration' tab is active. In the main area, the 'Triggers (1)' section is visible, showing a single trigger named 'API Gateway:MYAPI'. The details for this trigger include: 'API type: REST', 'Authorization: NONE', 'Scalability: No', 'Method: GET', 'Resource path: /bills', 'Service principal: apigateway.amazonaws.com', 'Stage: new\_stage', and 'Statement ID: 669c517b-ea06-517e-bd95-27584b4e13b2'. The status bar at the bottom indicates 'Successfully created deployment for MYAPI. This deployment is active for new\_stage'.

# Energy Consumption Metering | Yashwanth aravanti

**Code properties** [Info](#)

Package size  
1.3 kB

Encryption with AWS KMS customer managed KMS key [Info](#)

**Runtime settings** [Info](#)

Runtime  
Python 3.15

Handler [Info](#)  
lambda\_function.lambda\_handler

Architecture [Info](#)  
x86\_64

**Layers** [Info](#)

Merge order	Name	Layer version	Compatible runtimes	Compatible architectures	Version ARN
1	AWSSDKPandas-Python313	4	python3.13	x86_64	arn:aws:lambda:ap-south-1:336392948345:layer:AWSSDKPandas-Python313:4

[https://ap-south-1.console.aws.amazon.com/lambda/home?region=ap-south-1#/functions/Billing\\_API\\_Lambda?tab=code](https://ap-south-1.console.aws.amazon.com/lambda/home?region=ap-south-1#/functions/Billing_API_Lambda?tab=code) © 2025, Amazon Web Services, Inc. or its affiliates. [Privacy](#) [Terms](#) [Cookie preferences](#)

**Billing\_API\_Lambda**

**Function overview** [Info](#)

[Diagram](#) [Template](#)

**Description**  
Last modified 7 hours ago

**Function ARN**  
arn:aws:lambda:ap-south-1:487615743519:function:Billing\_API\_Lambda

**Function URL** [Info](#)

**Configuration** [Edit](#) [View role document](#)

**Execution role**  
Role name Billing\_API\_Lambda-role-4a1d0lqd

[CloudShell](#) [Feedback](#) © 2025, Amazon Web Services, Inc. or its affiliates. [Privacy](#) [Terms](#) [Cookie preferences](#)

Month=05, Year=2007

```

{s
  "Date": "01-05-2007",
  "total_Sub_metering_1": 4005,
  "total_Sub_metering_2": 995,
  "total_Sub_metering_3": 854,
  "avg_submetering_value": 4515.3333333333,
  "total_daily_sum": 13546,
  "peak_charge": 4982.887,
  "offpeak_charge": 5388.18,
  "total_charge": 10371.067
},
{
  "Date": "02-05-2007",
  "total_Sub_metering_1": 7,
  "total_Sub_metering_2": 1490,
  "total_Sub_metering_3": 3754,
  "avg_submetering_value": 1750.3333333333,
  "total_daily_sum": 5251,
  "peak_charge": 1875.729,
  "offpeak_charge": 3914.86,
  "total_charge": 5789.789
},
{
  "Date": "03-05-2007",
  "total_Sub_metering_1": 1906,
  "total_Sub_metering_2": 1725,
  "total_Sub_metering_3": 7836,
  "avg_submetering_value": 3822.3333333333,
  "total_daily_sum": 5251,
  "peak_charge": 2562.155,
  "offpeak_charge": 3944.04,
  "total_charge": 6506.195
},
{
  "Date": "04-05-2007",
  "total_Sub_metering_1": 2259,
  "total_Sub_metering_2": 538,
  "total_Sub_metering_3": 3990,
  "avg_submetering_value": 2262.3333333333,
  "total_daily_sum": 6787
  "peak_charge": 3505.383,
  "offpeak_charge": 2928.65,
  "total_charge": 6438.033
}

```

Month=10, Year=2007

```
pretty-print
{
  "error": "No billing data found for month 2007-10"
}
```

Month=12, Year=2006

```
pretty-print
[ {
  "Date": "16-12-2006",
  "total_Sub_metering_1": 0,
  "total_Sub_metering_2": 545,
  "total_Sub_metering_3": 4926,
  "avg_submetering_value": 1823.6666666667,
  "total_daily_sum": 5471,
  "peak_charge": 6812.308,
  "offpeak_charge": 2058.64,
  "total_charge": 8850.948
},
{
  "Date": "17-12-2006",
  "total_Sub_metering_1": 2033,
  "total_Sub_metering_2": 4187,
  "total_Sub_metering_3": 13341,
  "avg_submetering_value": 6520.3333333333,
  "total_daily_sum": 19561,
  "peak_charge": 6633.961,
  "offpeak_charge": 13049.97,
  "total_charge": 19683.931
},
{
  "Date": "18-12-2006",
  "total_Sub_metering_1": 1063,
  "total_Sub_metering_2": 2021,
  "total_Sub_metering_3": 14019,
  "avg_submetering_value": 5900.6666666667,
  "total_daily_sum": 17702,
  "peak_charge": 5221.04,
  "offpeak_charge": 7947.93,
  "total_charge": 13168.97
},
{
  "Date": "19-12-2006",
  "total_Sub_metering_1": 839,
  "total_Sub_metering_2": 7602,
  "total_Sub_metering_3": 6197,
  "avg_submetering_value": 4879.3333333333,
  "total_daily_sum": 14638,
  "peak_charge": 4299.402,
  "offpeak_charge": 5801.91,
  "total_charge": 10100.312
}]
```

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

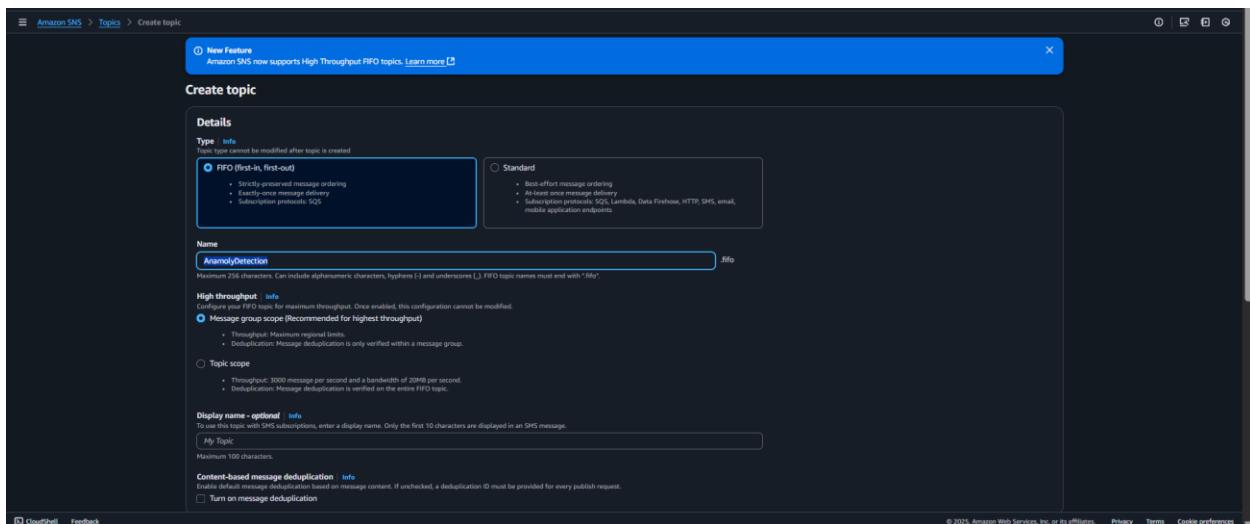
#### Flow:

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.



The screenshot shows the AWS SNS Topics page. On the left, there's a sidebar with 'Amazon SNS' navigation: Dashboard, Topics (selected), Subscriptions, Mobile (Push notifications, Text messaging (SMS)). The main area has a blue banner at the top stating 'New Feature: Amazon SNS now supports High Throughput FIFO topics. Learn more'. Below it, a table titled 'Topics (1)' lists one item: 'Name: AnomalyDetection', 'Type: Standard', and 'ARN: arn:aws:sns:ap-south-1:487615743519:AnomalyDetection'. There are buttons for 'Edit', 'Delete', 'Publish message', and 'Create topic'.

This screenshot shows the 'AnomalyDetection' topic details page. It includes sections for 'Details' (Name: AnomalyDetection, ARN: arn:aws:sns:ap-south-1:487615743519:AnomalyDetection, Type: Standard) and 'Display name' (Topic owner: 487615743519). Below these are tabs for 'Subscriptions', 'Access policy', 'Data protection policy', 'Delivery policy (HTTP/S)', 'Delivery status logging', 'Encryption', 'Tags', and 'Integrations'. The 'Subscriptions' tab shows one entry: 'ID: 0926c4bc-f4c0-4942-950c-13e5a8be7dd8', 'Endpoint: codingmatters2004@gmail.com', 'Status: Confirmed', and 'Protocol: EMAIL'. Buttons for 'Edit', 'Delete', 'Request confirmation', 'Confirm subscription', and 'Create subscription' are present.

The screenshot shows an email from 'AWS Notifications <no-reply@sns.amazonaws.com>' to 'me' dated 'Fri 17 Oct, 21:32 (10 hours ago)'. The subject is 'AWS Notification - Subscription Confirmation'. The body of the email contains the following text:

You have chosen to subscribe to the topic:  
**arn:aws:sns:ap-south-1:487615743519:AnomalyDetection**

To confirm this subscription, click or visit the link below (if this was in error no action is necessary):  
[Confirm subscription](#)

Please do not reply directly to this email. If you wish to remove yourself from receiving all future SNS subscription confirmation requests please send an email to [sns-opt-out](#)

At the bottom are buttons for 'Reply', 'Forward', and an emoji.

## 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.

**Create function** Info

Choose one of the following options to create your function.

- Author from scratch Start with a simple Hello World example.
- Use a blueprint Build a Lambda application from sample code and configuration presets for common use cases.
- Container image Select a container image to deploy for your function.

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.  
**Billing-Alerts**

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (\_).

**Runtime** Info  
Choose the language to use for writing your function. Note that the console code editor supports only Node.js, Python, and Ruby.  
**Python 3.13**

**Architecture** Info  
Choose the instruction set architecture you want for your function code.  
 arm64  
 x86\_64

**Permissions** Info  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

**Additional configurations**  
Use additional configurations to set up networking, security, and governance for your function. These settings help secure and customize your Lambda function deployment.

**Create function**

**Functions (2)**

Function name	Description	Package type	Runtime	Last modified
<a href="#">Billing_API_Lambda</a>	-	Zip	Python 3.15	18 hours ago
<a href="#">Billing-Alerts</a>	-	Zip	Python 3.15	10 hours ago

**Billing-Alerts**

**Function overview** Info

**Diagram** **Template**

**Billing-Alerts**

**S3**

**Add destination**

**Throttle** **Copy ARN** **Actions**

**Description**

**Last modified**  
10 hours ago

**Function ARN**  
[arn:aws:lambda:ap-south-1:487615743519:function:Billing-Alerts](#)

**Function URL** Info

```

    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:AnomalyDetection"

    # --- 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"}

            # Set 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']

            # Analyze data for anomalies
            # ...
        except Exception as e:
            print(f"An error occurred: {e}")
            sns.publish(TopicArn=SNS_TOPIC_ARN, Message=f"An error occurred: {e}")
            return {"statusCode": 500, "body": "An error occurred while processing the data."}
    
```

```

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:AnomalyDetection"

# --- 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"}

        # Set 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']

        # Analyze data for anomalies
        # ...
    except Exception as e:
        print(f"An error occurred: {e}")
        sns.publish(TopicArn=SNS_TOPIC_ARN, Message=f"An error occurred: {e}")
        return {"statusCode": 500, "body": "An error occurred while processing the data."}
    
```

```

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.

The screenshot shows the AWS IAM Roles page. The role 'Billing-Alerts-role-mnqrqlqel' is selected. It has a creation date of October 17, 2025, at 21:00 (UTC+05:30). The maximum session duration is set to 1 hour. Three managed policies are attached: 'AdministratorAccess', 'AmazonS3FullAccess', and 'AWSLambdaBasicExecutionRole'. The 'Permissions' tab is active, showing the attached policies and their types (AWS managed and Customer managed).

The screenshot shows the CloudWatch Logs for a Lambda function. The logs output details about detected anomalies in the billing data. Key entries include:

- Sub-metering spike on 17-12-2006 in 'total\_Sub\_metering\_3': 13341.0
- Sub-metering spike on 18-12-2006 in 'total\_Sub\_metering\_3': 14018.0
- Sub-metering spike on 20-12-2006 in 'total\_Sub\_metering\_3': 14063.0
- Sub-metering spike on 21-12-2006 in 'total\_Sub\_metering\_3': 10421.0
- Sub-metering spike on 22-12-2006 in 'total\_Sub\_metering\_3': 11131.0
- Sub-metering spike on 23-12-2006 in 'total\_Sub\_metering\_3': 14726.0
- Sub-metering spike on 26-12-2006 in 'total\_Sub\_metering\_3': 14979.0
- Sub-metering spike on 29-12-2006 in 'total\_Sub\_metering\_3': 11329.0
- High consumption alert on 30-12-2006: 23443.00 (Avg: 13130.28)
- Sub-metering spike on 30-12-2006 in 'total\_Sub\_metering\_3': 12480.0
- Sub-metering spike on 04-01-2007 in 'total\_Sub\_metering\_3': 10896.0
- High consumption alert on 07-01-2007: 22974.00 (Avg: 13130.28)
- Sub-metering spike on 07-01-2007 in 'total\_Sub\_metering\_3': 12810.0
- Sub-metering spike on 08-01-2007 in 'total\_Sub\_metering\_3': 17547.0
- Sub-metering spike on 10-01-2007 in 'total\_Sub\_metering\_3': 11012.0
- Sub-metering spike on 11-01-2007 in 'total\_Sub\_metering\_3': 12415.0
- High consumption alert on 13-01-2007: 22122.00 (Avg: 13130.28)
- Sub-metering spike on 13-01-2007 in 'total\_Sub\_metering\_3': 11341.0
- Sub-metering spike on 14-01-2007 in 'total\_Sub\_metering\_3': 12975.0
- Sub-metering spike on 15-01-2007 in 'total\_Sub\_metering\_3': 15163.0
- High consumption alert on 17-01-2007: 25301.00 (Avg: 13130.28)
- Sub-metering spike on 17-01-2007 in 'total\_Sub\_metering\_3': 16064.0
- Sub-metering spike on 18-01-2007 in 'total\_Sub\_metering\_3': 13491.0

**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": "s3GetObject",
      "Resource": "arn:aws:s3:::electricity-meter-aggregated-readings/*"
    }
  ]
}
```

The screenshot shows the AWS S3 Bucket Policy configuration page for the bucket 'electricity-meter-aggregated-readings'. The left sidebar shows various AWS services like Amazon S3, IAM, and CloudWatch Metrics. The main area displays the JSON policy code. A note at the top states: 'Public access is blocked because Block Public Access settings are turned on for this bucket. To determine which settings are turned on, check your Block Public Access settings for this bucket. Learn more about using Amazon S3 Block Public Access.' Below the policy, the 'Object Ownership' section shows 'Bucket owner untrusted'.

### 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');

```

The screenshot shows the Amazon Athena Query Editor interface. The left sidebar displays the 'Data' section with 'Data source' set to 'AwsDataCatalog', 'Catalogue' set to 'None', and 'Database' set to 'energy\_bills'. Under 'Tables and views', there is one table named 'daily\_summary'. The main workspace contains the SQL code for creating the external table. At the bottom, the status bar shows 'Completed' with a timestamp of 'Time in queue: 276 ms' and 'Run time: 1.227 sec'.

```

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;

```

# Energy Consumption Metering | Yashwanth aravanti

The screenshot shows the Amazon Athena Query Editor interface. The left sidebar displays the Data source (AwsDataCatalog), Catalogue (None), and Database (energy\_bills). Under Tables and views, there is one table named 'daily\_summary'. The main area contains a complex SQL query:

```

1 SELECT
2   TRY_CAST(date AS DATE) AS consumption_date,
3   avg_global_active_power AS avg_active_power,
4   avg_voltage AS avg_voltage,
5   total_Sub_metering_1 AS sub_meter_1,
6   COALESCE(peak_charge, 0) AS peak_charge,
7   COALESCE(offpeak_charge, 0) AS offpeak_charge,
8   (COALESCE(peak_charge, 0) + COALESCE(offpeak_charge, 0)) AS calculated_total_charge
9 FROM
10   energy_bills.daily_summary
11 WHERE
12   TRY_CAST(date AS DATE) BETWEEN DATE '2007-04-01' AND DATE '2007-04-30'
13 ORDER BY
14   TRY_CAST(date AS DATE)
15 LIMIT 100;
    
```

The results section shows 29 rows of data with columns: #, consumption\_date, avg\_active\_power, avg\_voltage, sub\_meter\_1, peak\_charge, offpeak\_charge, and calculated\_total\_charge. The first row is highlighted.

The screenshot shows the results of the query from the previous screenshot. The results table has 14 rows of data, each corresponding to a specific date in April 2007. The columns are: #, consumption\_date, avg\_active\_power, avg\_voltage, sub\_meter\_1, peak\_charge, offpeak\_charge, and calculated\_total\_charge. The data scanned is 30.48 KB.

#	consumption_date	avg_active_power	avg_voltage	sub_meter_1	peak_charge	offpeak_charge	calculated_total_charge
1	2007-04-01	1.0240000000000007	260.0744011111113	4629.0	6664.544	10005.459999999999	16673.983999999997
2	2007-04-02	1.3616515888888888	240.460180555557	1755.0	4478.82	7169.29	11648.11
3	2007-04-03	1.1408472222222221	240.5049833333335	3630.0	5191.200000000001	5158.099999999999	10353.1
4	2007-04-04	1.4249083333333332	240.39178472222224	3434.0	4444.803	7644.75	12089.553
5	2007-04-05	1.2806097222222221	241.1731041666667	1086.0	6101.198	5631.45	11732.648000000001
6	2007-04-06	1.1119027777777778	241.18445138888887	0.0	4277.302	5489.64	9766.942
7	2007-04-07	1.079113888888889	240.5785027777776	1127.0	2565.656999999997	6260.410000000001	8626.067000000001
8	2007-04-08	1.1172355353535353	240.3491805555556	0.0	615.043	7682.290000000001	8297.333
9	2007-04-09	0.6950835535353535	240.55048611111113	2667.0	3385.244	3015.279999999997	6398.523999999999
10	2007-04-10	0.5398361111111112	240.8137013888889	209.0	1616.411	2935.99	4552.401
11	2007-04-11	0.6163069444444444	241.1548263888889	0.0	624.88600000000001	4069.83	4694.716
12	2007-04-12	0.3734069444444444	240.2560624999998	0.0	974.202	2115.47	3089.671999999996
13	2007-04-13	0.3826541666666667	240.05046111111113	0.0	734.213	2323.220000000003	3057.433
14	2007-04-14	0.5849083553535355	240.51514583535352	0.0	621.516	2405.86	3022.176000000004

## Select all the items from the table

SELECT \* FROM "energy\_bills"."daily\_summary" limit 10;

The screenshot shows the Amazon Athena Query Editor interface. The left sidebar displays the Data source (AwsDataCatalog), Catalogue (None), and Database (energy\_bills). Under Tables and views, there is one table named 'daily\_summary'. The main area contains a simple SQL query:

```

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

The results section shows 10 rows of data with columns: #, date, avg\_global\_active\_power, avg\_global\_reactive\_power, avg\_voltage, min\_voltage, max\_voltage, avg\_global\_intensity, total\_sub\_metering\_1, and total\_sub\_mi. The first row is highlighted.

## Energy Consumption Metering | Yashwanth aravanti

The screenshot shows the Amazon Athena Query Editor interface. At the top, there are buttons for 'Run again', 'Explain', 'Cancel', 'Clear', and 'Create'. To the right, there is a note about a query being reused up to 60 minutes ago. Below the buttons, tabs for 'Query results' and 'Query status' are visible, with 'Query results' being active. A progress bar indicates the query is 'Completed'. The results table has 10 rows and includes columns such as '#', 'date', 'avg\_global\_active\_power', 'avg\_global\_reactive\_power', 'avg\_voltage', 'min\_voltage', 'max\_voltage', 'avg\_global\_intensity', 'total\_sub\_metering\_1', and 'total\_sub\_mi'. The data is timestamped from December 16, 2006, to December 25, 2006.

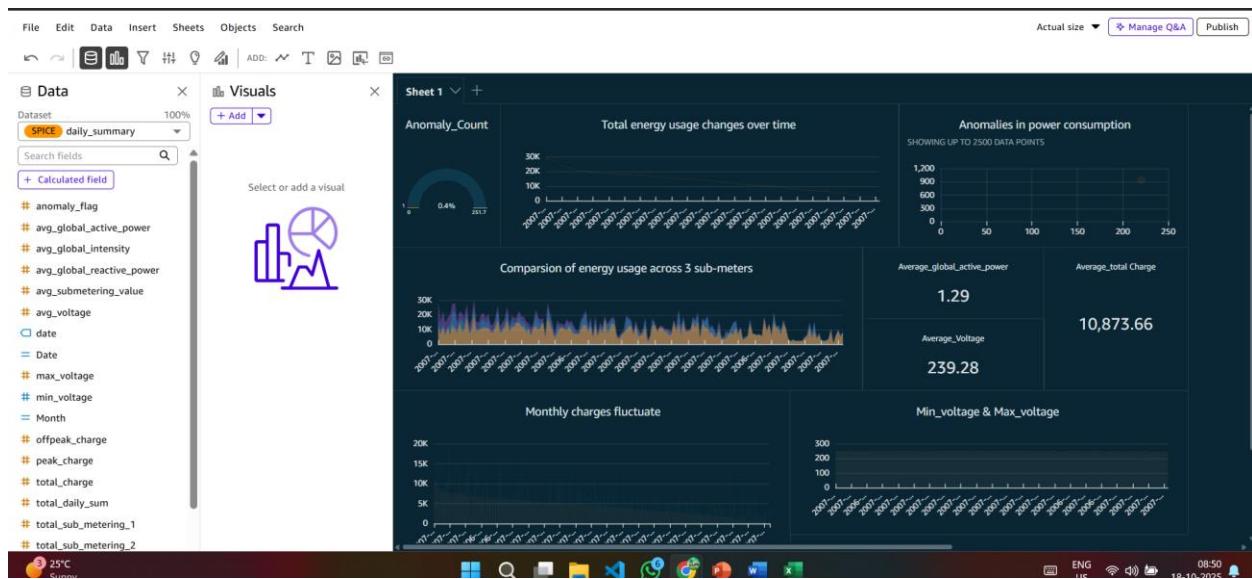
#	date	avg_global_active_power	avg_global_reactive_power	avg_voltage	min_voltage	max_voltage	avg_global_intensity	total_sub_metering_1	total_sub_mi
1	2006-12-16	3.0534747474747475	0.08818686868686867	236.24576262626262	230.98	245.73	13.0872878282828283	0.0	545.0
2	2006-12-17	2.354486111111111	0.1569486111111111	240.0870277777778	229.57	249.37	9.999027777777778	2033.0	4187.0
3	2006-12-18	1.530434722222222	0.12355555555555556	241.23169444444446	229.08	248.48	6.4216666666666657	1065.0	2621.0
4	2006-12-19	1.1570791666666667	0.1048208333333334	241.9993125	231.24	248.89	4.925188888888889	839.0	7602.0
5	2006-12-20	1.5456833333333334	0.11180166666666666	242.3080624999998	233.43	249.48	6.467361111111111	0.0	2648.0
6	2006-12-21	1.193757997218359	0.10025452016689847	241.04054937413073	228.91	247.08	5.0379640194716	1765.0	2623.0
7	2006-12-22	1.6259291666666669	0.1297958333333333	241.1845466111111	230.59	248.82	6.8729166666666667	3151.0	550.0
8	2006-12-23	3.514851588888889	0.1537986111111112	240.1360763888889	231.17	247.23	14.028055555555557	2669.0	425.0
9	2006-12-24	1.7708416666666669	0.1040972222222223	241.6874375	231.08	249.27	7.6404166666666667	1703.0	5082.0
10	2006-12-25	1.9049444444444445	0.1668611111111111	243.3999305555558	233.48	250.62	7.951527777777779	6620.0	1962.0

## Step 11: Quicksight Dashboards

The screenshot shows the AWS QuickSight dataset list. On the left, there is a sidebar with navigation links for Home, Favorites, Analyses, Dashboards, Stories, Topics, Datasets (which is selected), My folders, and Shared folders. The main area displays a table of datasets with columns for Name, Source, Owner, Last Modified, and Action. The datasets listed are: energy\_bills (Athena source, Me owner, 19 hours ago), Web and Social Media Analytics (S3 source, Me owner, 19 days ago), Sales Pipeline (S3 source, Me owner, 19 days ago), People Overview (S3 source, Me owner, 19 days ago), and Business Review (S3 source, Me owner, 19 days ago).

This screenshot shows the AWS QuickSight dataset list, similar to the one above but with different dataset names. The sidebar and dataset table structure are identical, showing datasets named daily\_summary, Business Review, People Overview, Sales Pipeline, and Web and Social Media Analytics, all owned by 'SPICE' and modified 19 days ago.

# Energy Consumption Metering | Yashwanth aravanti



## 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.