**End-to-End Data Engineering Pipeline for Renewable Energy Data with AWS and Python**

Rajesh. R. Shahu

**Introduction**

The transition to renewable energy sources presents significant opportunities and challenges in modern energy systems. As the deployment of solar farms, wind parks, and other sustainable generation facilities increases, so does the need for robust data-driven infrastructure to monitor, analyze, and ensure operational efficiency. This project presents a scalable, serverless data engineering solution for simulating, ingesting, processing, and visualizing energy generation and consumption data across multiple renewable energy sites.

Using AWS cloud services including S3, Lambda, DynamoDB, and SNS—this system emulates a real-time energy monitoring pipeline that identifies anomalies, stores structured data, and provides analytical insights through interactive dashboards and visualizations. The objective is to demonstrate how cloud-native technologies and modern data engineering practices can be leveraged to build resilient energy analytics pipelines that support anomaly detection, trend analysis, and informed decision-making in renewable energy operations.

The pipeline was developed for a mock renewable energy company and showcases real-world data engineering practices, including:

* Real-time ingestion of JSON-based data using Amazon S3
* Serverless processing with AWS Lambda
* Storing processed data in Amazon DynamoDB
* Anomaly detection and alerting via Amazon SNS
* Providing queryable access through FastAPI
* Visualization using Seaborn, Matplotlib and Plotly
* Logging and monitoring via Amazon CloudWatch

**Install Python**

Before you proceed with AWS CLI or running any scripts in this project, ensure Python is installed on your system. Python is required for:

* Simulating the data feed
* Running the FastAPI server
* Generating visualizations with Matplotlib/Seaborn
* Interacting with AWS via Boto3

**Steps to Install:**

1. **Download Python for Windows:**

* Visit the official Python website: <https://www.python.org/downloads/>
* Click on the latest stable version to download the installer.

1. **Run the Installer:**

* When the installer launches, make sure to check the box that says “Add Python to PATH” before clicking “Install Now”.

1. **Verify Installation:**

Open Command Prompt and run:

*python --version*

*pip --version*

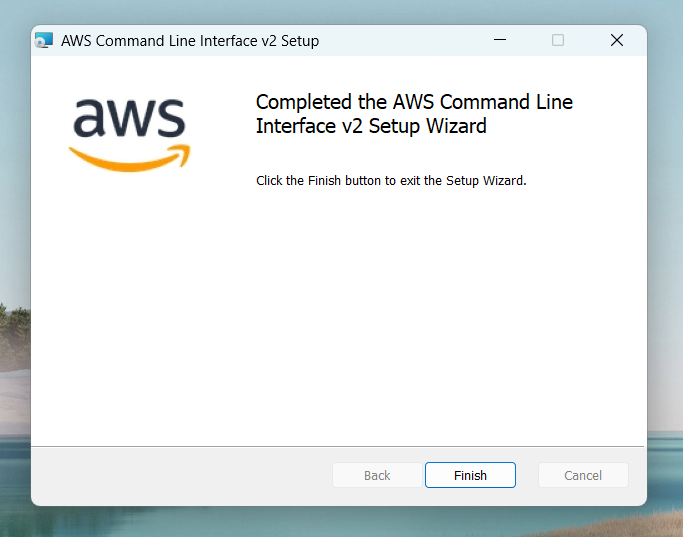
These commands should return the installed Python and pip versions.

If the command is not recognized, try restarting your terminal or checking if Python is added to your system’s environment variables.

**AWS CLI Setup**

To interact with AWS services programmatically and test services like S3, Lambda, DynamoDB, and SNS, I used the AWS CLI (Command Line Interface).

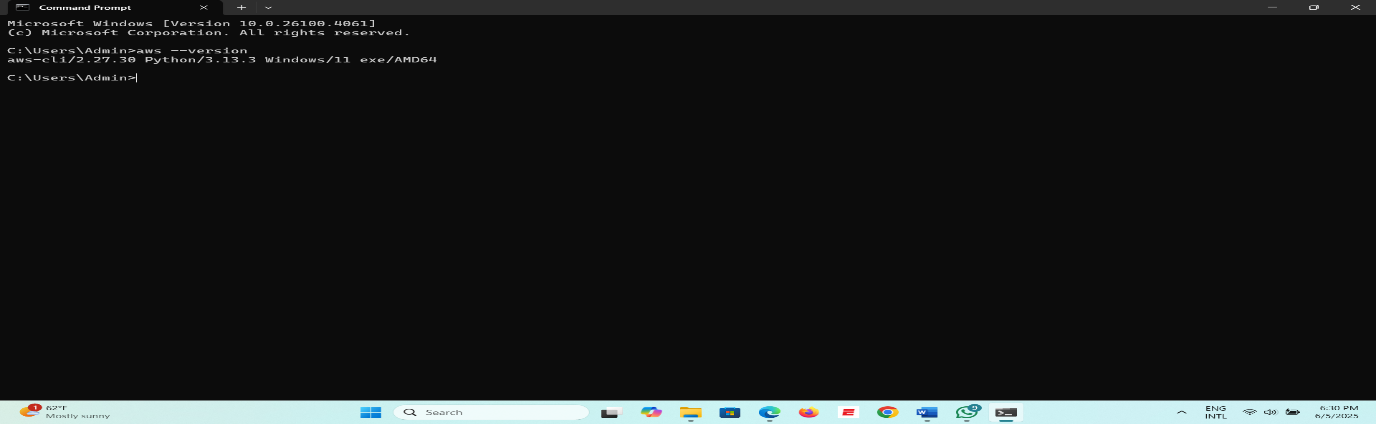
1. **Downloaded and installed AWS CLI:**  
   I followed the official guide for windows: <https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2-windows.html>.



1. **Verified the installation using:**

*aws –version*

This command confirmed that AWS CLI was installed correctly and the path was properly set.



This command confirmed that AWS CLI was installed correctly and the path was properly set.

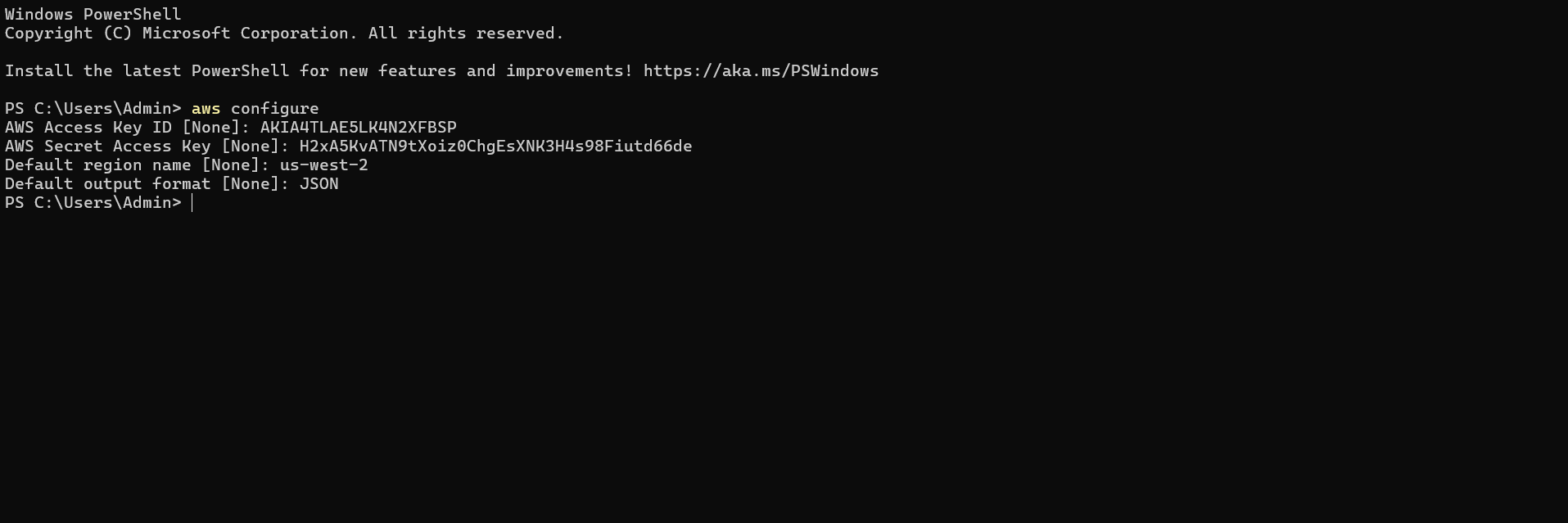
1. **Configured AWS credentials with:**

*aws configure*

I entered the following credentials and configuration:

* + **Access Key ID**:
  + **Secret Access Key**:
  + **Default region name**:
  + **Default output format**:

This step allowed my local system to authenticate and interact with AWS services securely.



1. **Installed Boto3 for AWS SDK integration in Python:**

*pip install boto3*

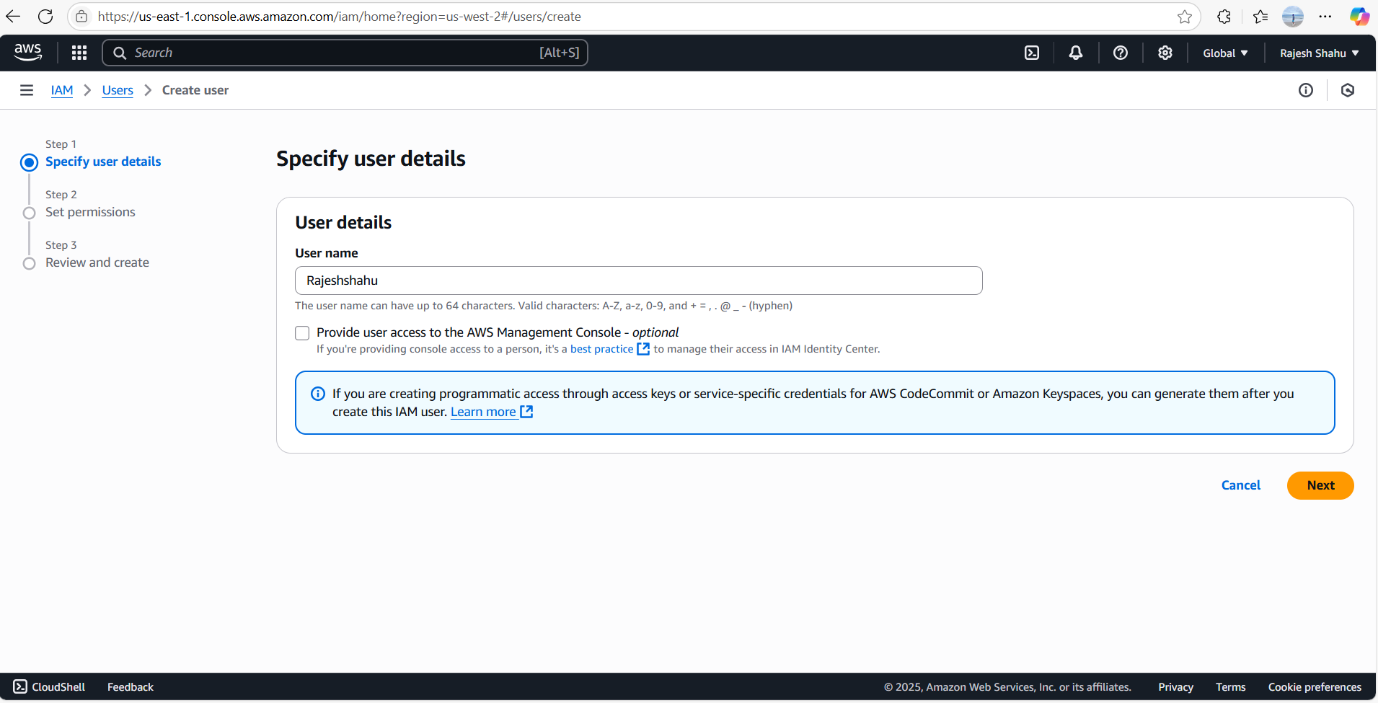
Boto3 is essential for interacting with AWS services like S3, Lambda, and DynamoDB directly from Python scripts.



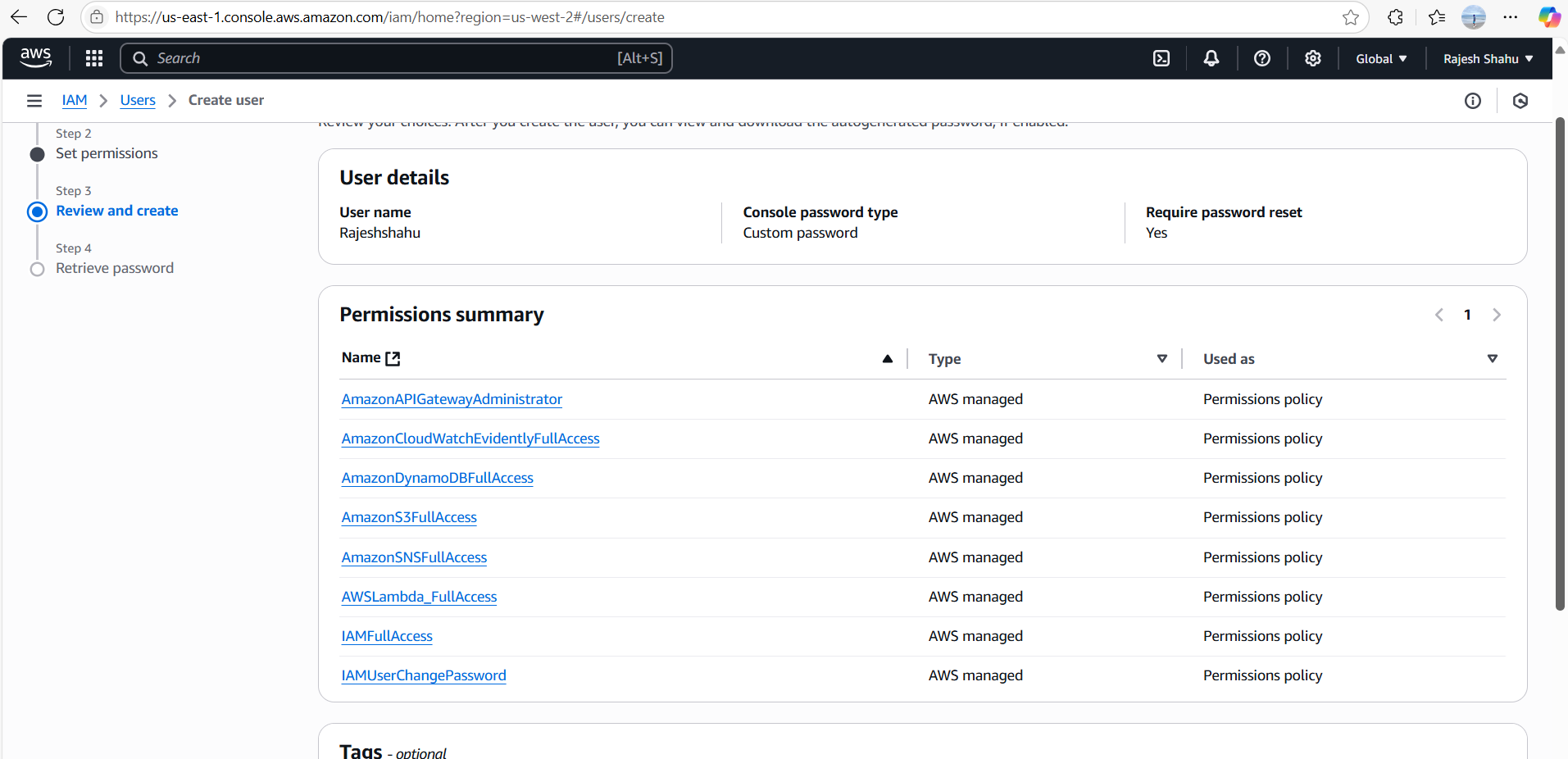
**IAM User and Permissions Setup**

To securely access AWS services via CLI and SDK, I created a dedicated IAM user.

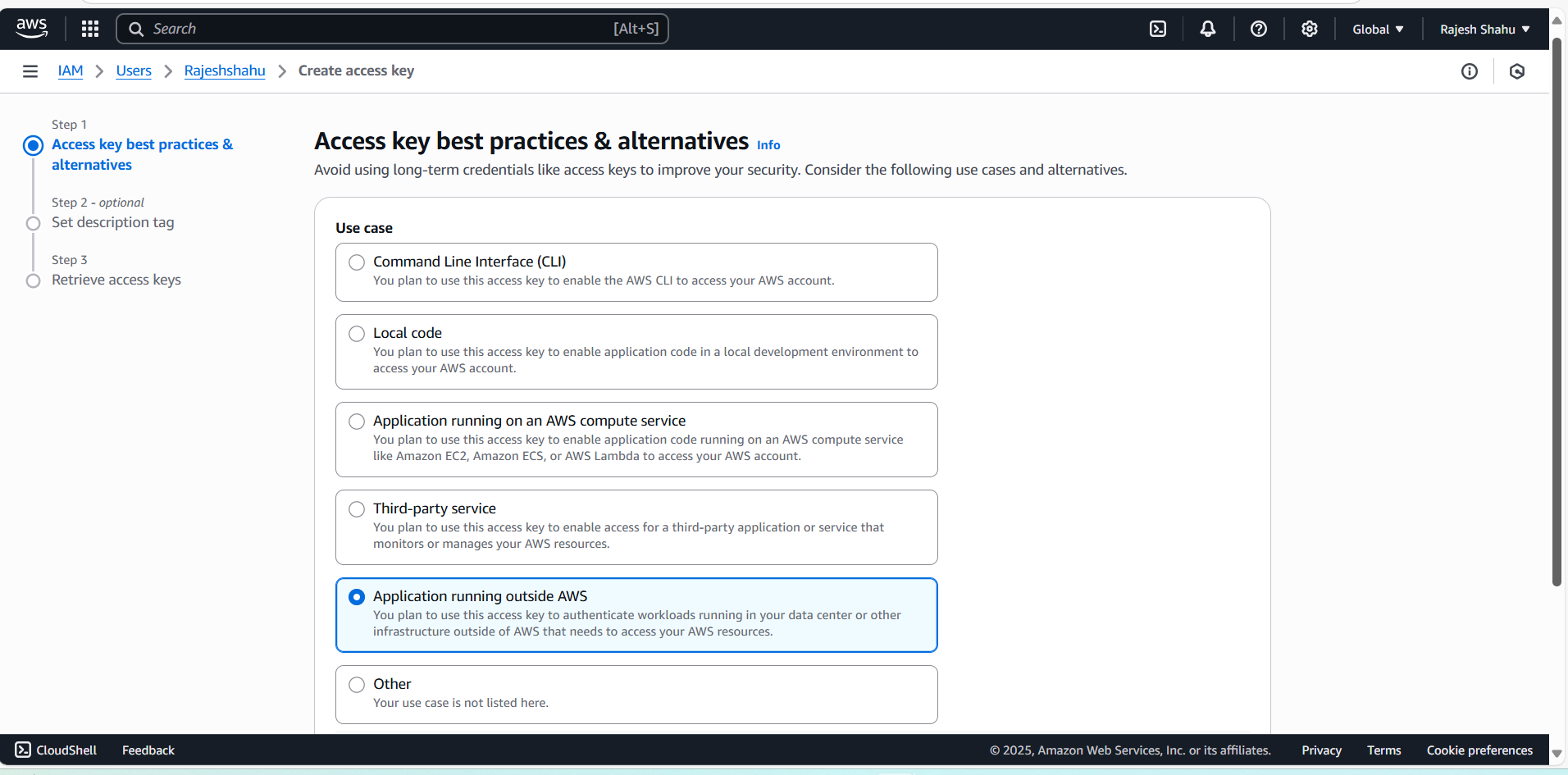
1. **Created a new IAM user:**
   * Username:
   * Access type: Programmatic access only



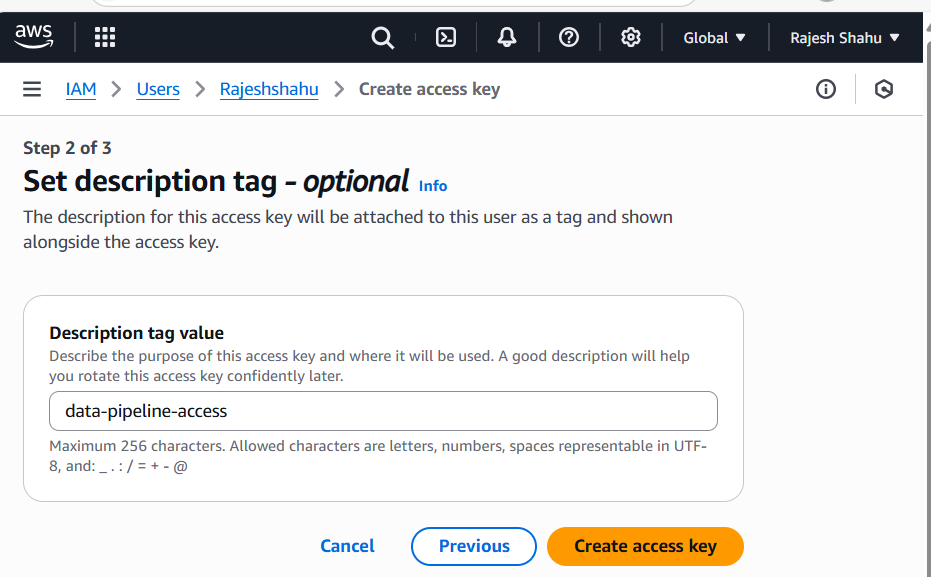
1. **Assigned permissions using AWS managed policies:**
   * AmazonS3FullAccess
   * AmazonDynamoDBFullAccess
   * AWSLambda\_FullAccess
   * AmazonSNSFullAccess
   * AmazonCloudWatchEvidentlyFullAccess
   * IAMFullAccess



1. **Use case setting for access key:**
   * Selected "Application running outside AWS" to allow Python scripts running locally to authenticate via boto3.



1. **Generated and described access key:**
   * Tag: data-pipeline-access to track key usage



This IAM setup provided secure, permissioned access for all CLI and SDK operations used in the project.

**Create Amazon S3 Bucket**

To ingest and store simulated energy data, I created an Amazon S3 bucket. This bucket serves as the landing zone where incoming .json files are uploaded from the simulated feed script.

**Why S3?**

* Acts as the trigger source for AWS Lambda
* Provides serverless, scalable storage
* Compatible with AWS CLI, Boto3, and real-time event-driven architecture

**Steps to follow:**

1. **Navigate to S3:**

* Open the AWS Management Console → Search for S3
* Click Create bucket

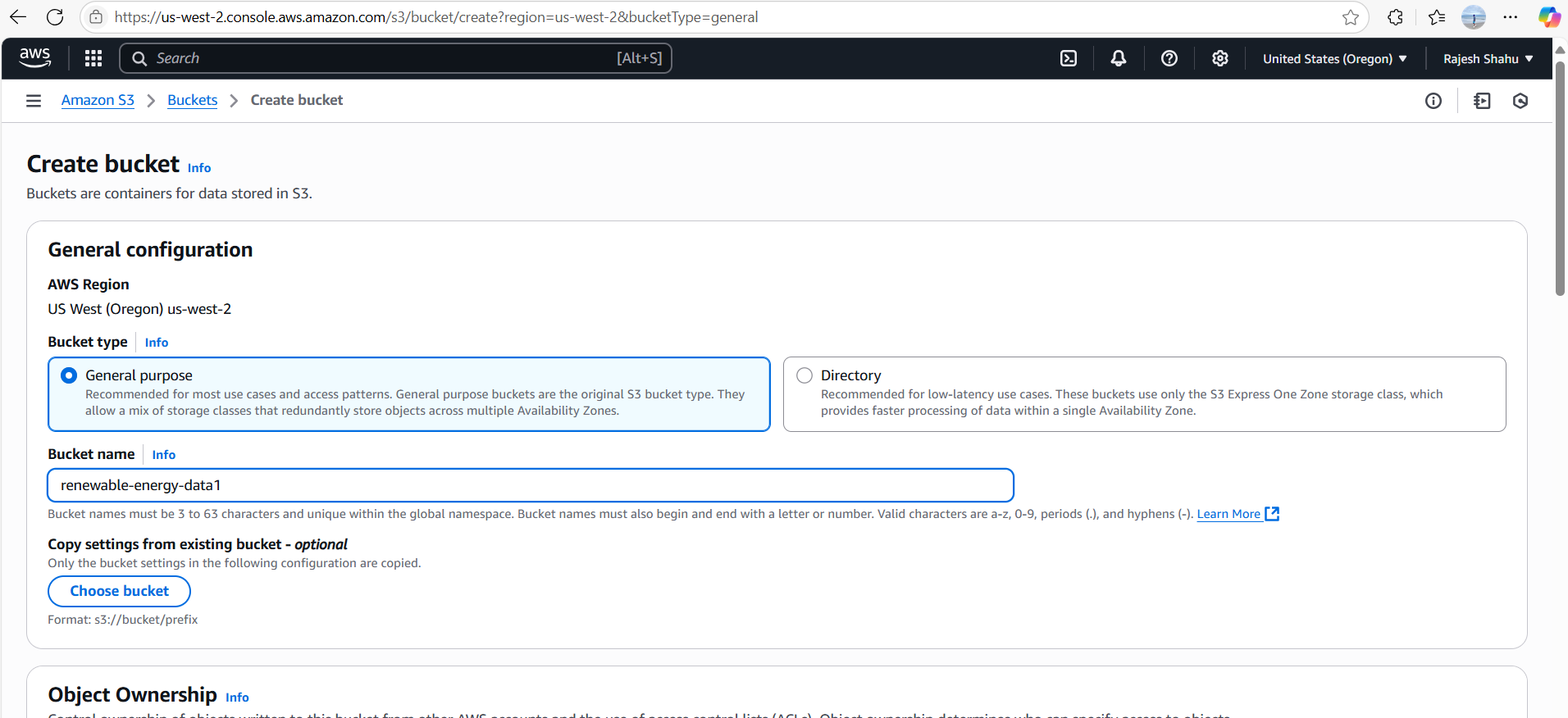
1. **Set Bucket Name & Region:**

* Bucket name: renewable-energy-data1 (make sure it's globally unique)
* Region: us-west-2 (same as configured via aws configure)

1. **Leave Other Settings as Default:**

* Versioning: Disabled (optional)
* Block Public Access: Enabled
* Encryption: None (optional for this project)
* Click Create Bucket

This bucket will now receive .json files simulated from Python scripts and act as an event trigger for AWS Lambda



**Simulated Data Feed with Python**

To simulate real-time energy generation and consumption data from multiple renewable sites, I created a Python script that generates random data and uploads it to **Amazon S3** every 5 minutes. This simulates continuous ingestion and triggers AWS Lambda for downstream processing.

**What this Script does:**

 Initializes the S3 client using boto3 to communicate with AWS S3.

 Defines a list of 5 fictitious energy sites such as "SolarFarm\_AZ\_001" and "WindPark\_CA\_002".

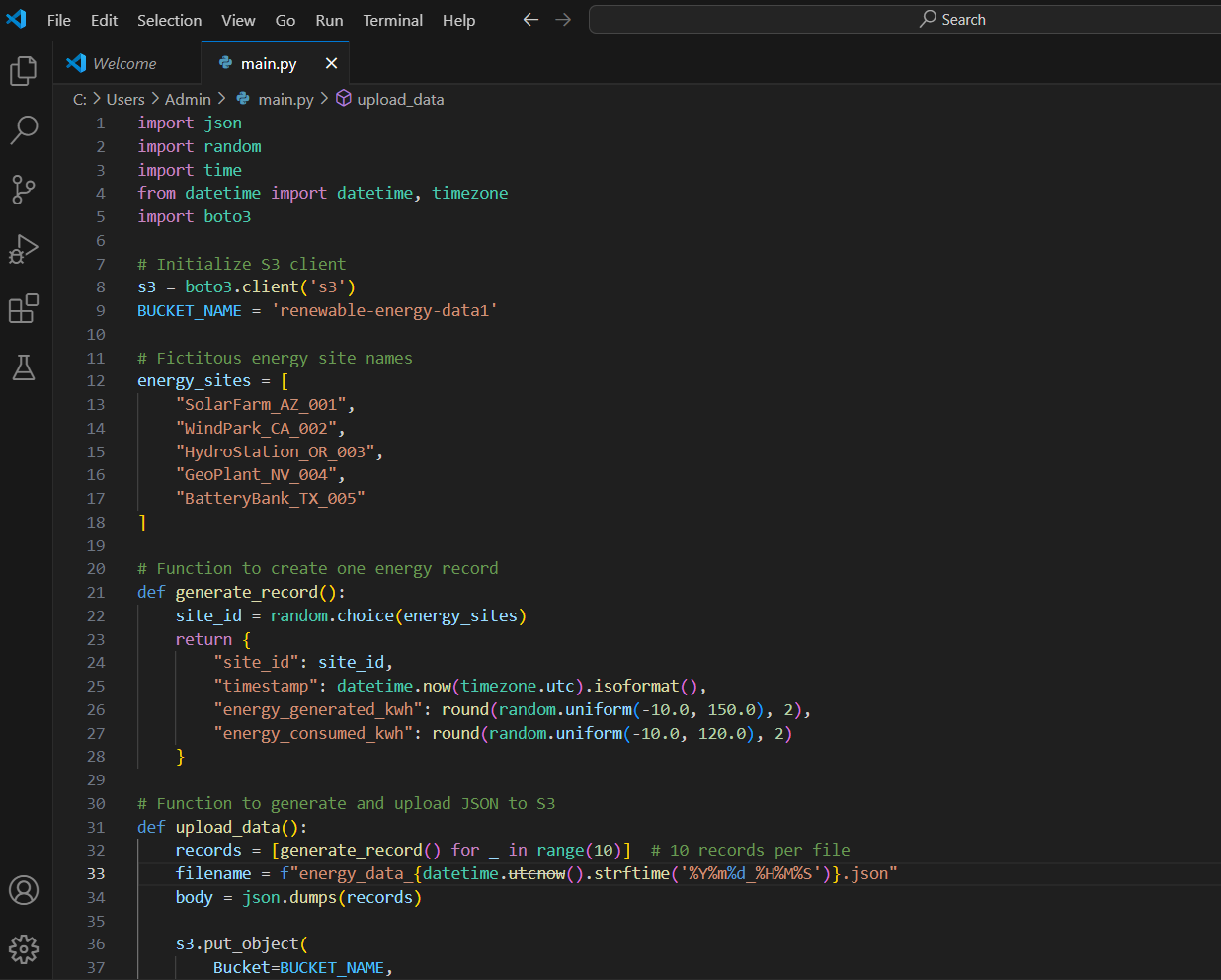
 Generates 10 random records in each iteration with:

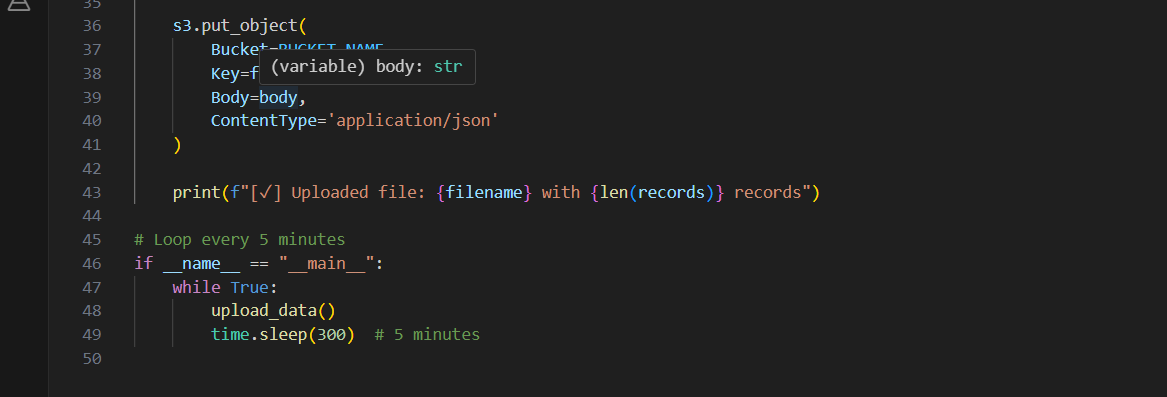
* **site\_id**: randomly chosen from the list
* **timestamp:** current UTC time
* **energy\_generated\_kwh**: random float (can be negative to simulate sensor error)
* **energy\_consumed\_kwh**: random float (can be negative to simulate anomaly)

 Uploads the records to the specified S3 bucket as a .json file with a unique timestamped filename.

 Runs in an infinite loop with a 5-minute sleep interval between uploads to simulate continuous data streaming.

**Code Overview:**

****



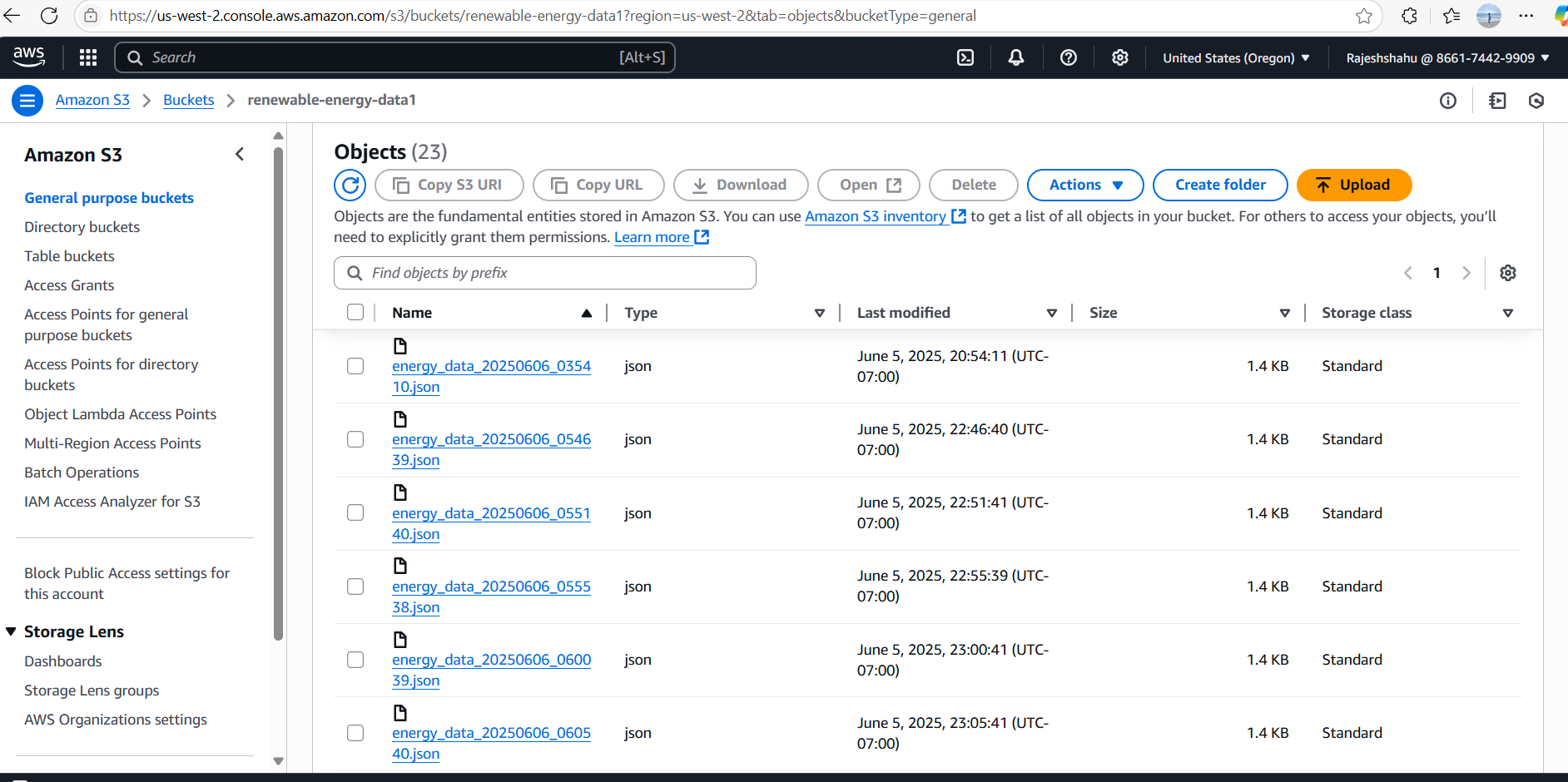
**Setup Instructions**

* Save this code as *main.py*
* Make sure boto3 is installed (pip install boto3)
* Update BUCKET\_NAME to match your actual S3 bucket
* Run the script:

*Python main.py*

**Result:**

After running the script, JSON files are automatically uploaded to the specified S3 bucket every 5 minutes.



**Amazon DynamoDB Table Creation**

to store the processed energy data from AWS Lambda, first I created a DynamoDB table named energy\_data. DynamoDB is a fully managed NoSQL database provided by AWS, ideal for storing time-series data with low latency and high scalability.

**Steps I followed:**

* **Navigated to DynamoDB Console**

Opened the AWS Console and searched for DynamoDB.

* **Clicked on ‘Create Table’**
* **Configured the table as follows:**

Table name: energy\_data

Partition key: site\_id (Type: String)

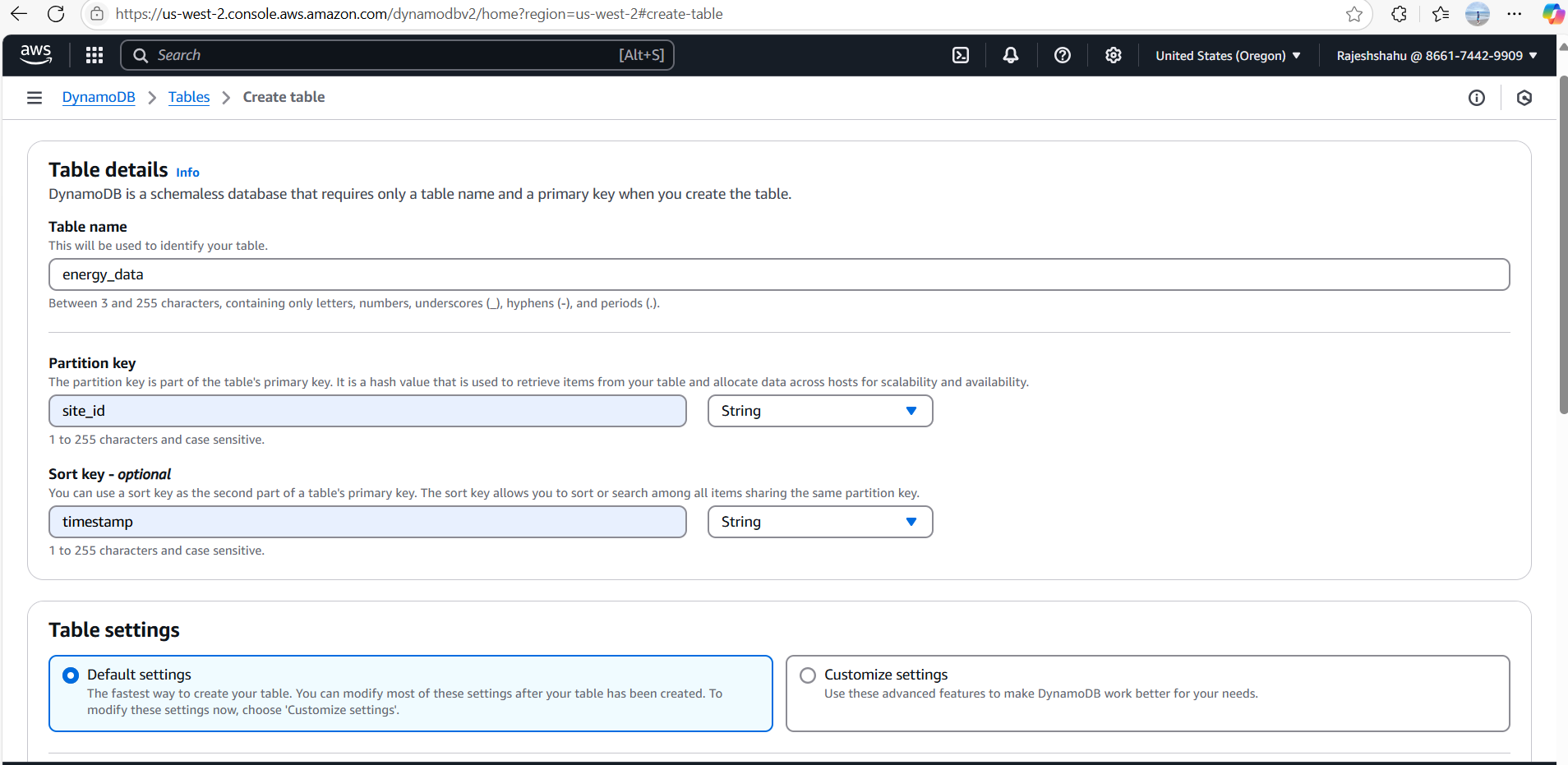
Sort key: timestamp (Type: String)

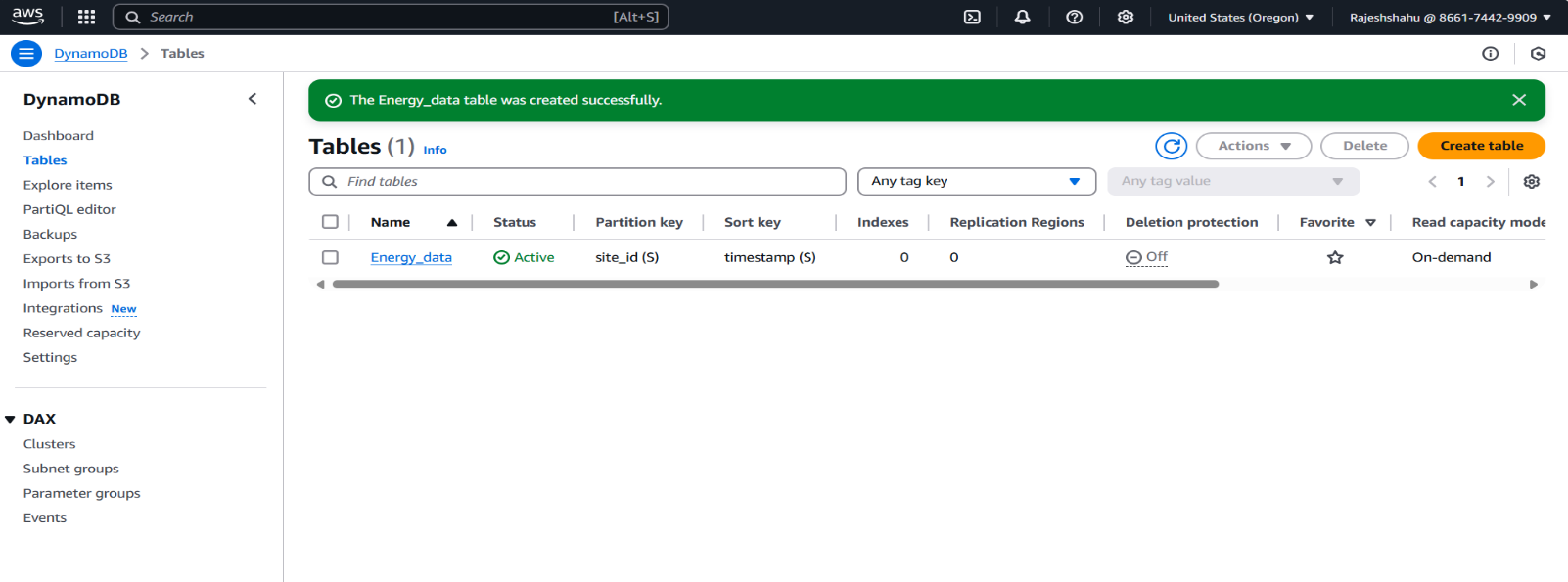
These keys ensure each site’s records are uniquely identified by the timestamp.

* **Kept other settings as default**

I didn’t enable DynamoDB Streams, auto-scaling, or encryption, as this project focuses on simplicity.

* **Clicked ‘Create’**

****

****

**Create IAM role for AWS Lambda Access**

To allow my AWS Lambda function to interact securely with other AWS services like S3, DynamoDB, and SNS, I created a dedicated IAM role named **lambda-energy-data-role**. This role grants Lambda the necessary permissions to perform its tasks in the pipeline.

**Why this step is needed:**

Lambda runs in a secure, isolated environment and needs explicit permissions to access external services like:

* Reading files from S3
* Writing processed data to DynamoDB
* Sending alerts through SNS

**Steps to follow:**

1. Go to IAM > Roles > Create Role
2. Select Trusted Entity:

* Chose **AWS service**
* Selected **Lambda** as the use case

1. Attached Required Policies

 AmazonS3FullAccess

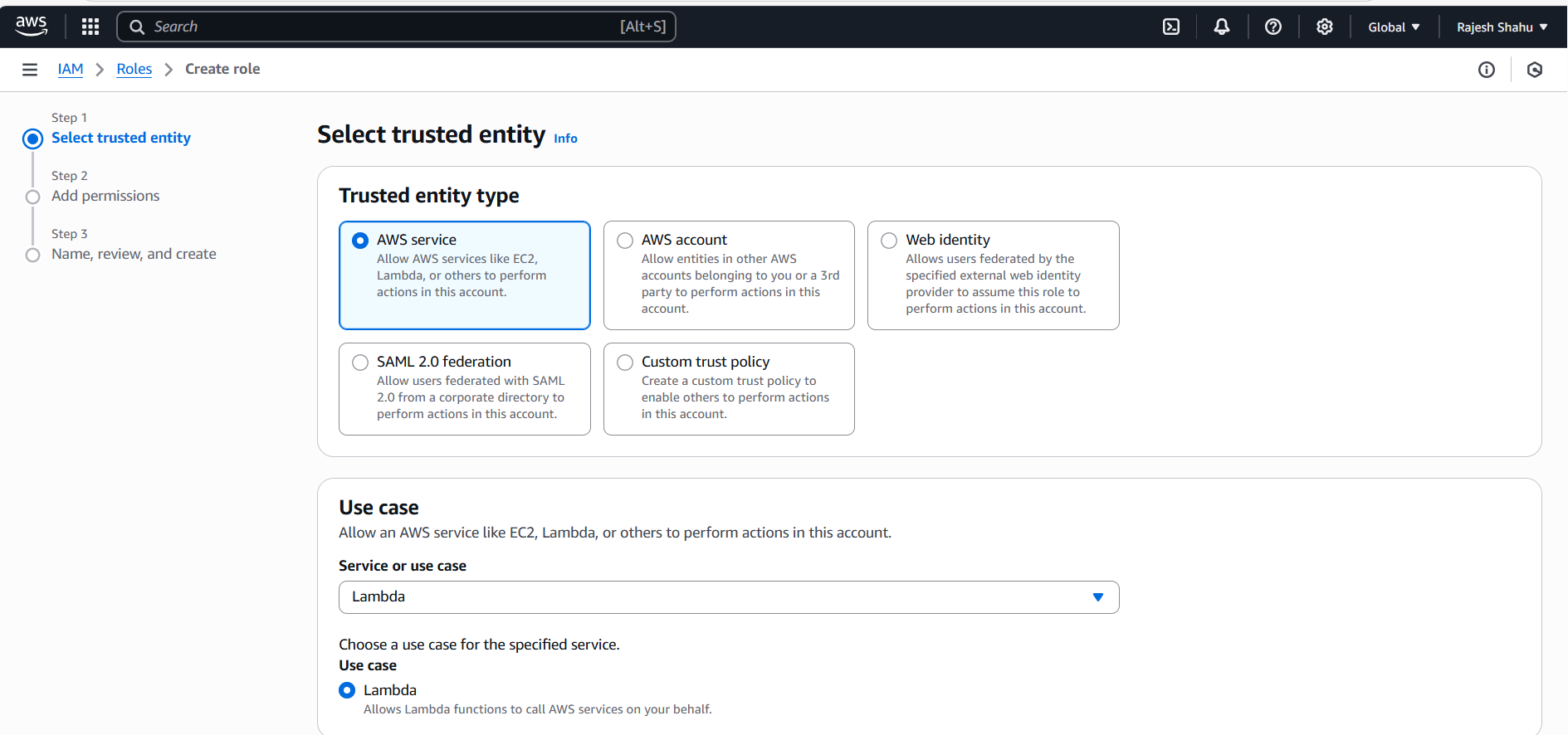
 AmazonDynamoDBFullAccess

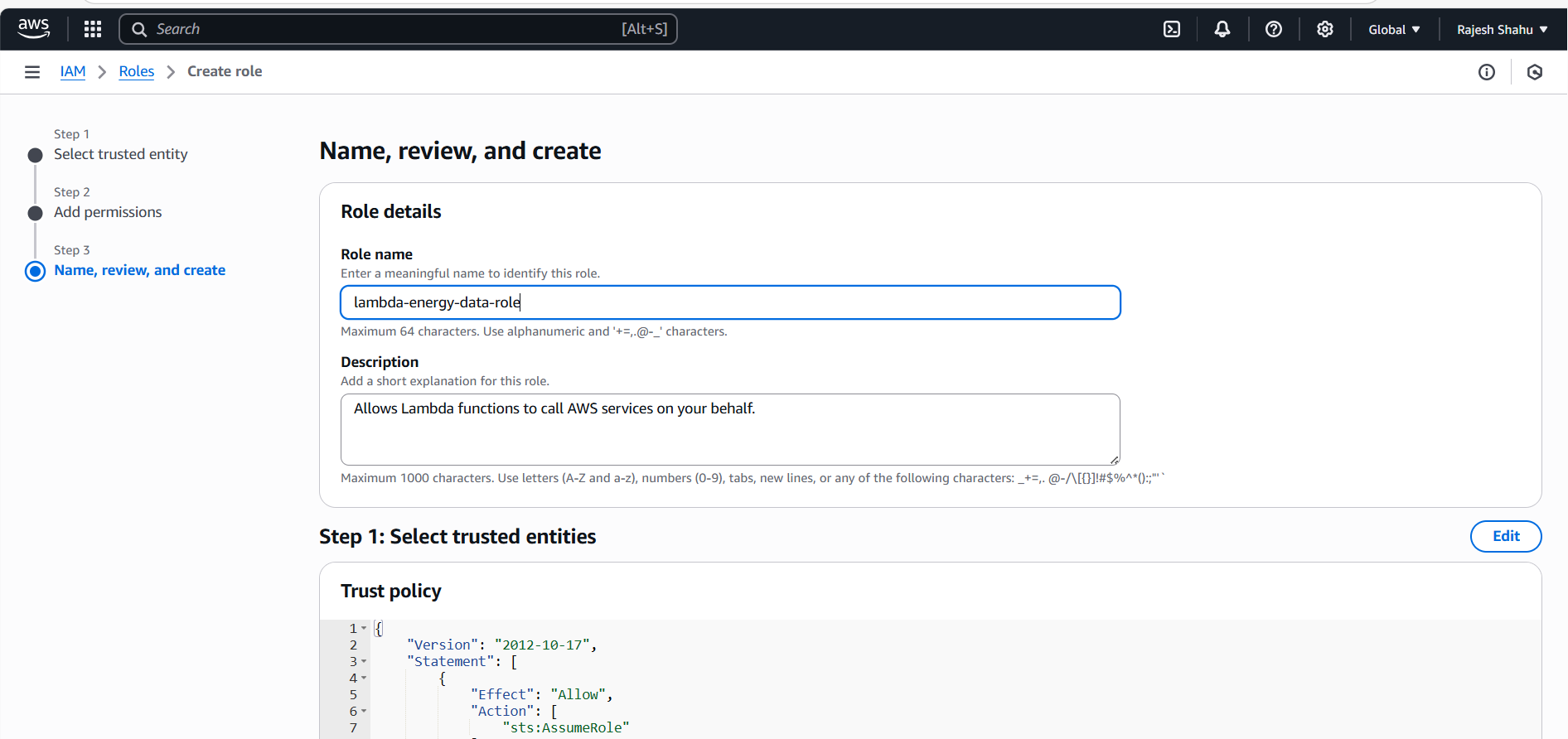
 AmazonSNSFullAccess

1. Name and Create Role

* Role name: lambda-energy-data-role
* Description: Allow Lambda Functions to call AWS services on your behalf.

Once this role was created, I assigned it to the Lambda function during its setup.





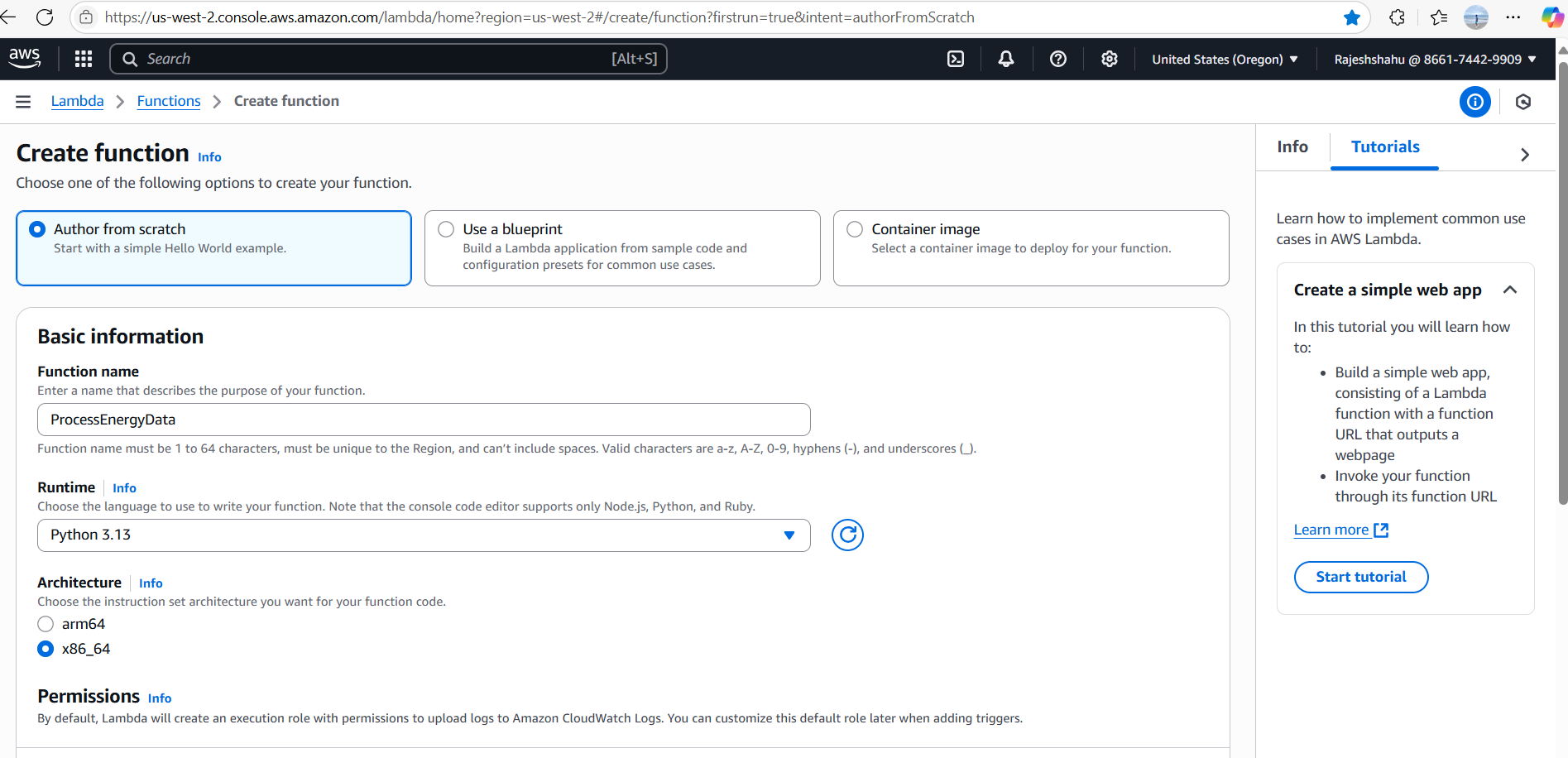
**AWS Lambda Function Setup**

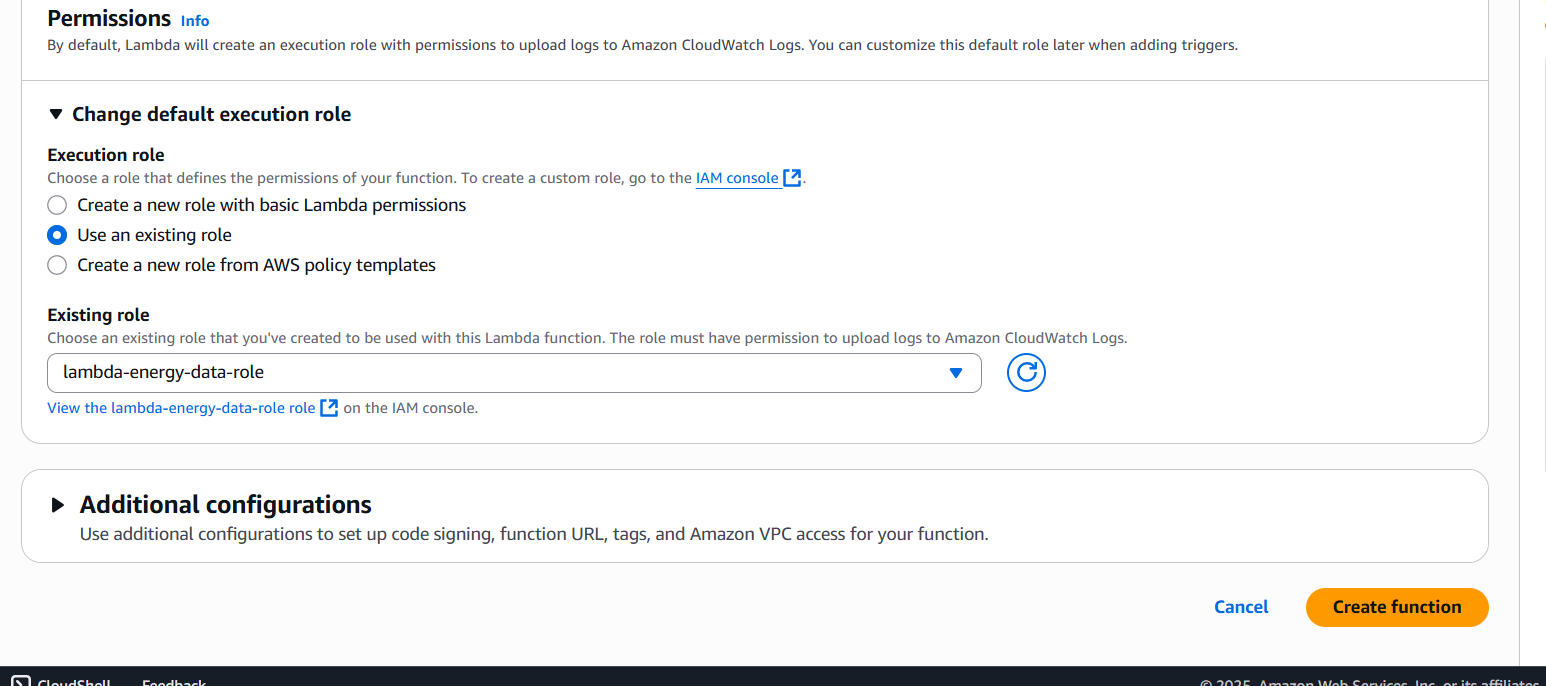
After creating the IAM role (lambda-energy-data-role), I proceeded to set up the Lambda function that would be triggered every time a new file is uploaded to S3.

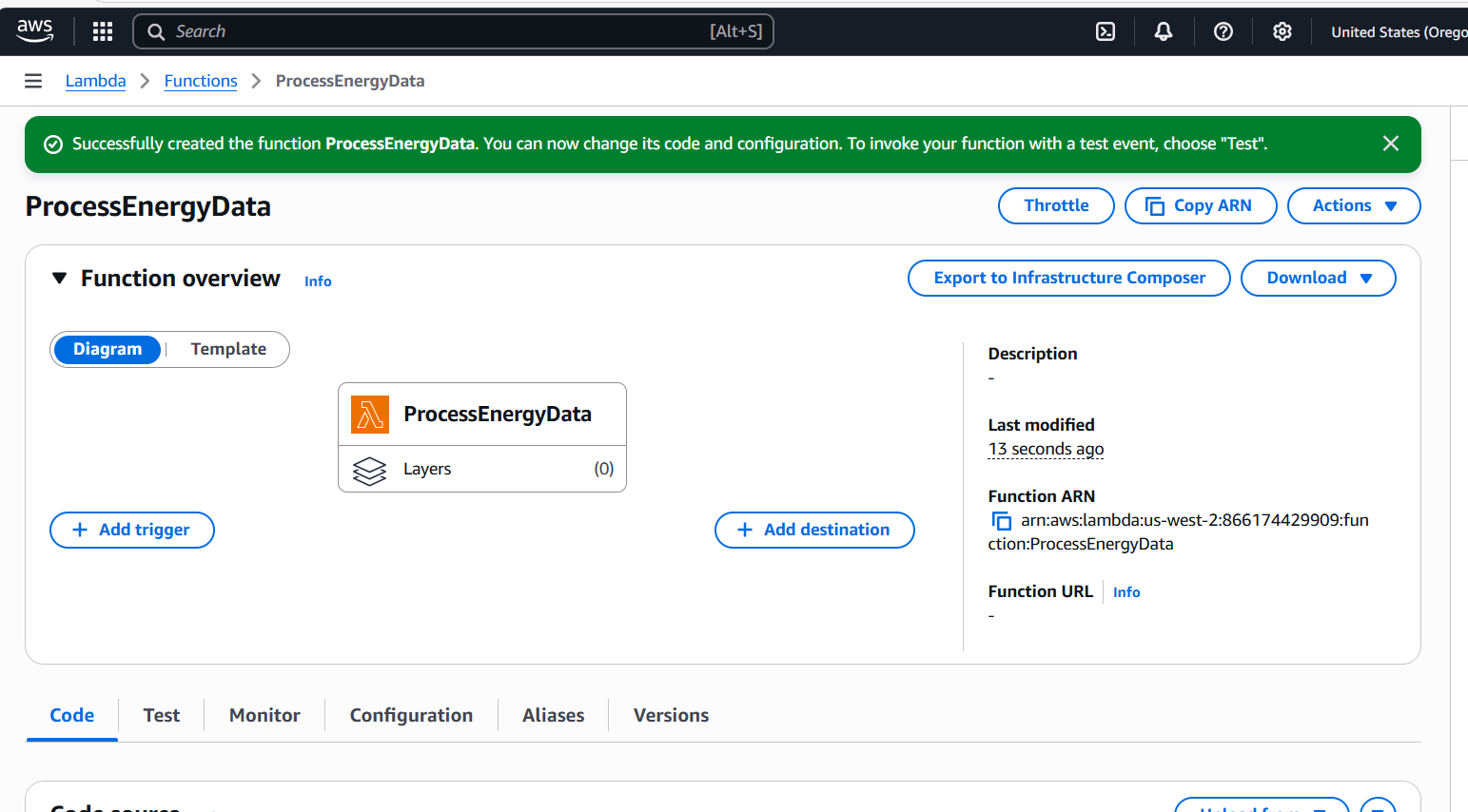
* Navigate to AWS Lambda Console
* Create function → Author from scratch.
* Fill basic configuration:
* Function name: ProcessEnergyData
* Runtime: Python 3.13
* Architecture: x86\_64
* Assigned IAM Role:

Choose an existing role and selected the one I created earlier (lambda-energy-data-role).

* Click on Create function







**Write Lambda Function Code**

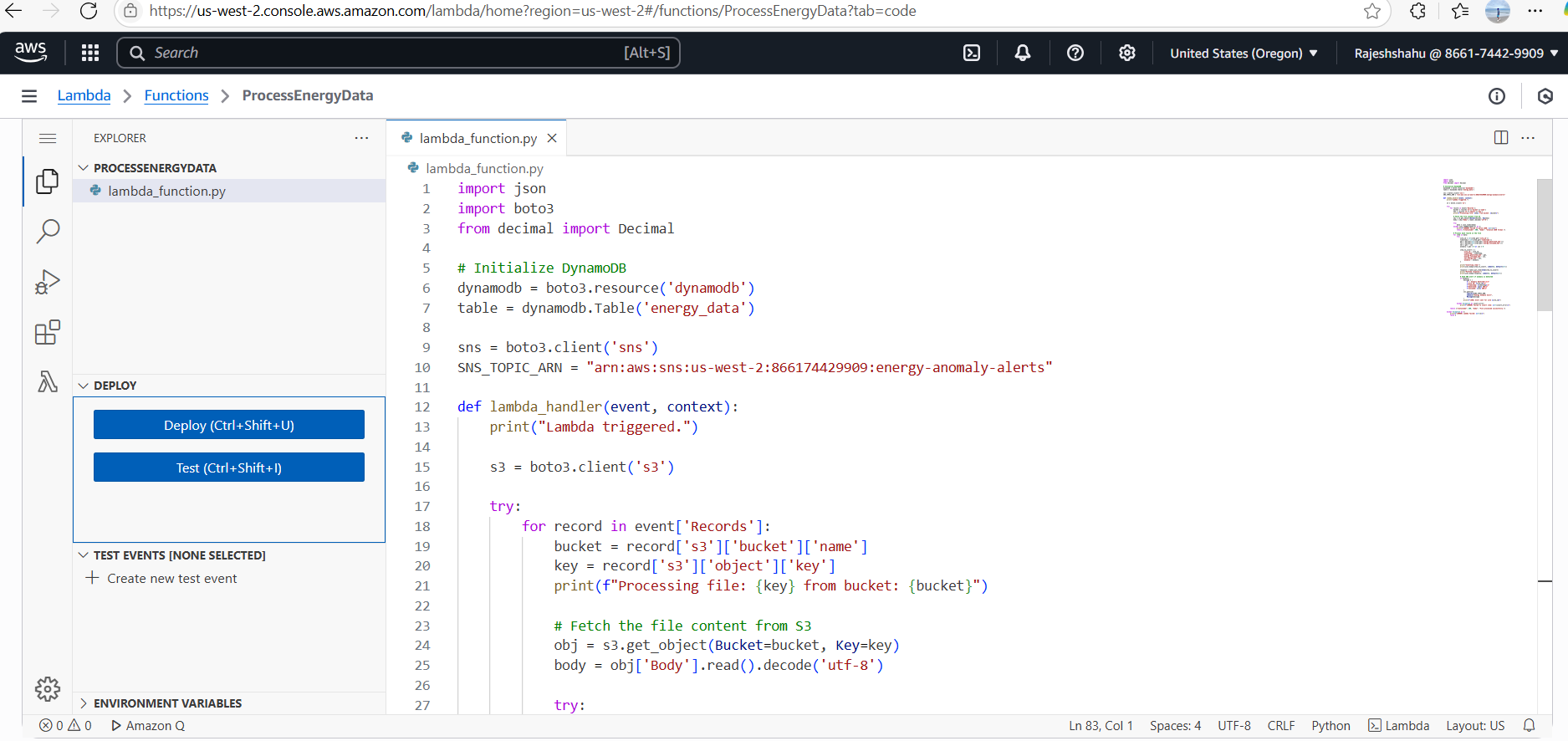
After setting up the Lambda function shell in the AWS Console, I developed the actual logic locally in a Python file named lambda\_function.py. This allowed easier testing, editing, and version control before deploying it to the cloud.

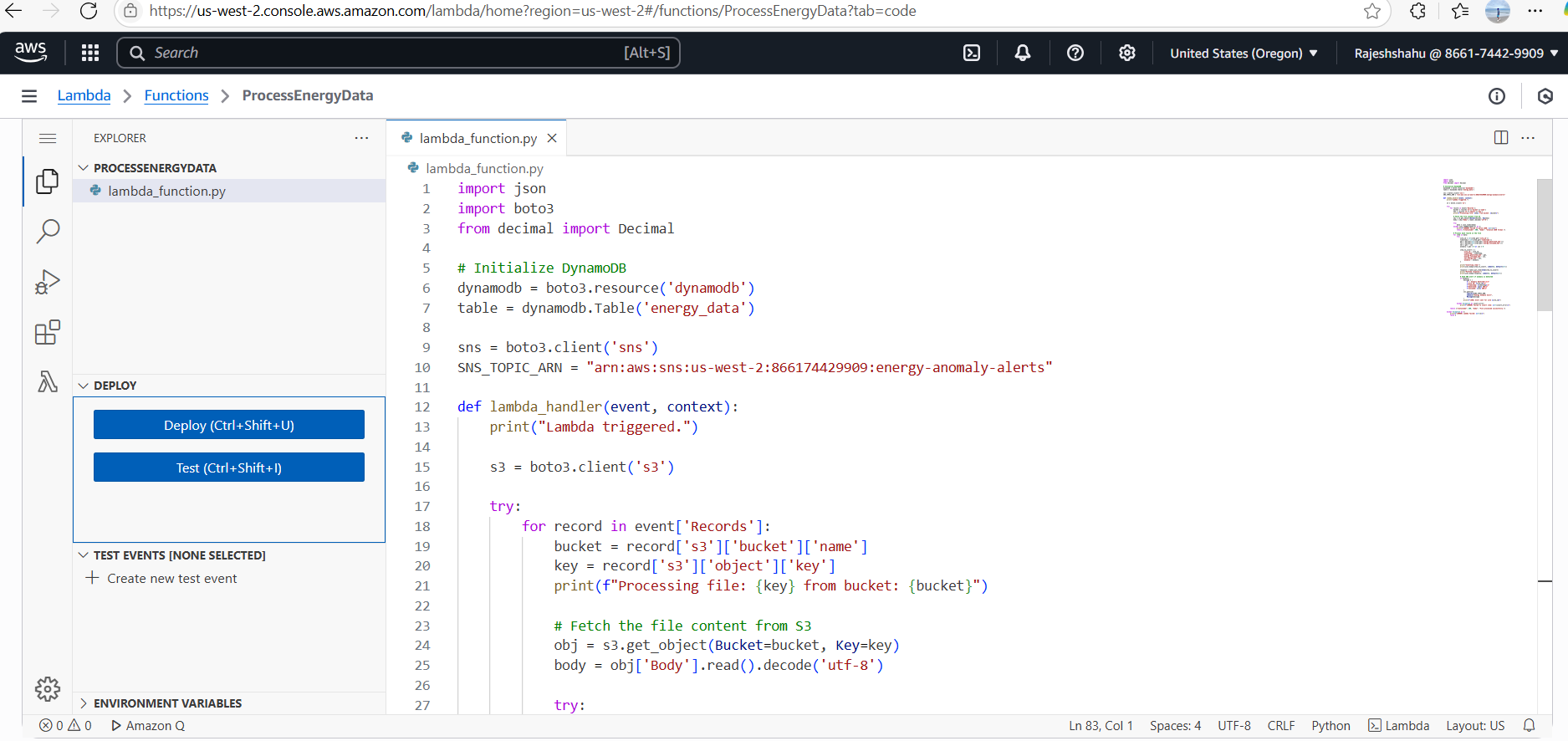
**What this function does:**

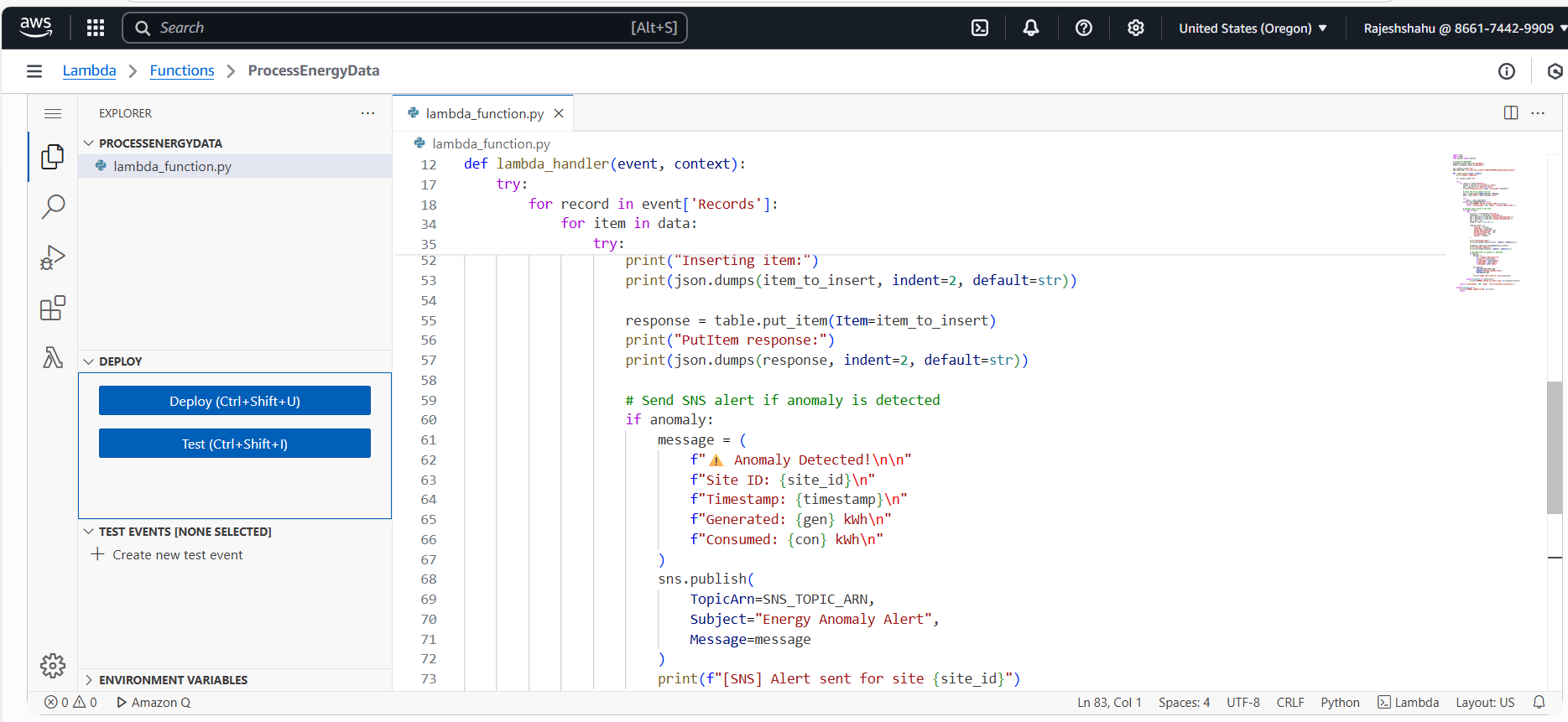
* Triggered automatically by an S3 PUT event (when a new JSON file is uploaded).
* Reads the uploaded file from S3.
* Parses energy site records.
* Detects anomalies: negative energy generation or consumption.
* Pushes valid records into DynamoDB.
* If anomalies are found, sends real-time alerts via Amazon SNS.
* Handles malformed records and logs errors cleanly.

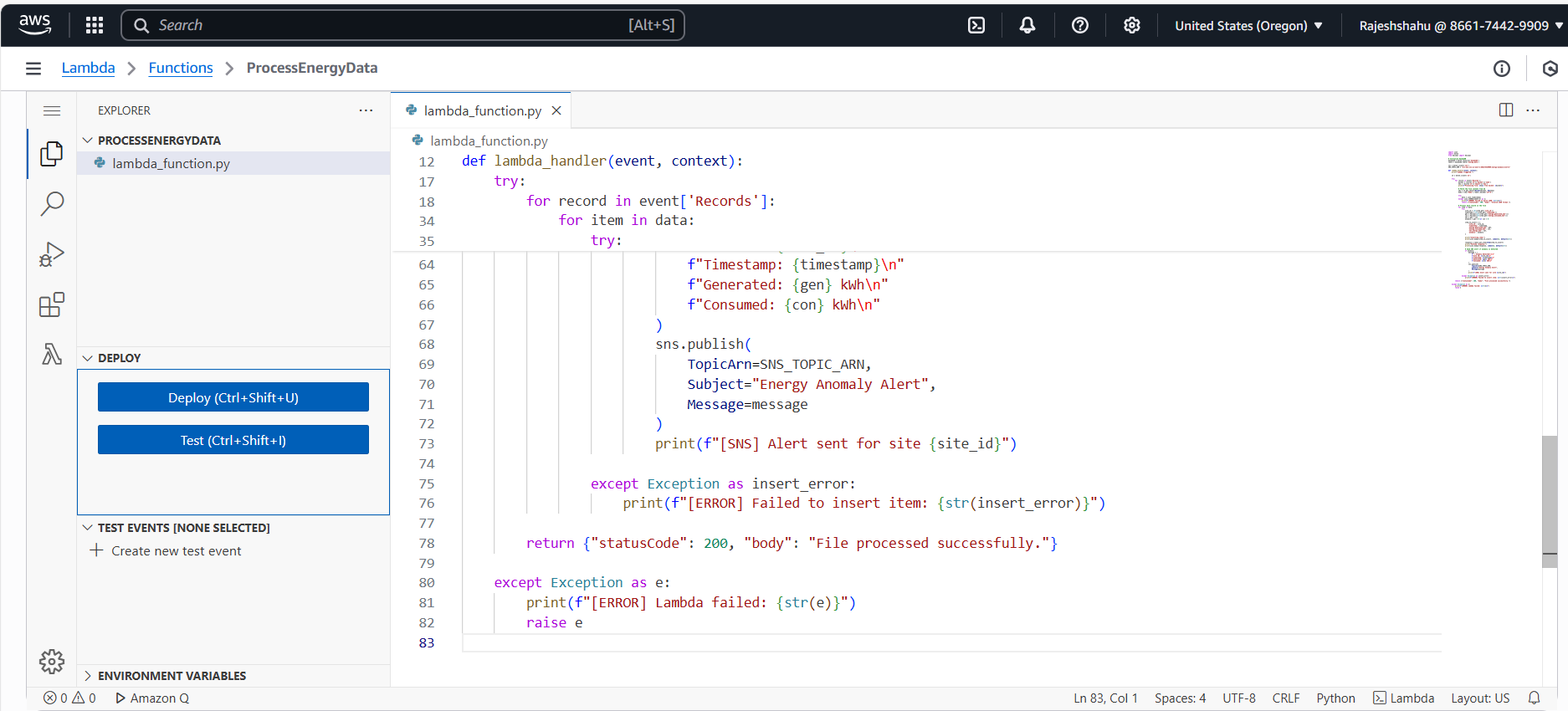
**Deploying Lambda Function Code**

1. Open your Lambda function (ProcessEnergyData) in the AWS Console.
2. Go to the **Code** tab.
3. Paste the script into the inline editor or upload a .zip if using dependencies.
4. Click **Deploy** to save and activate the function.









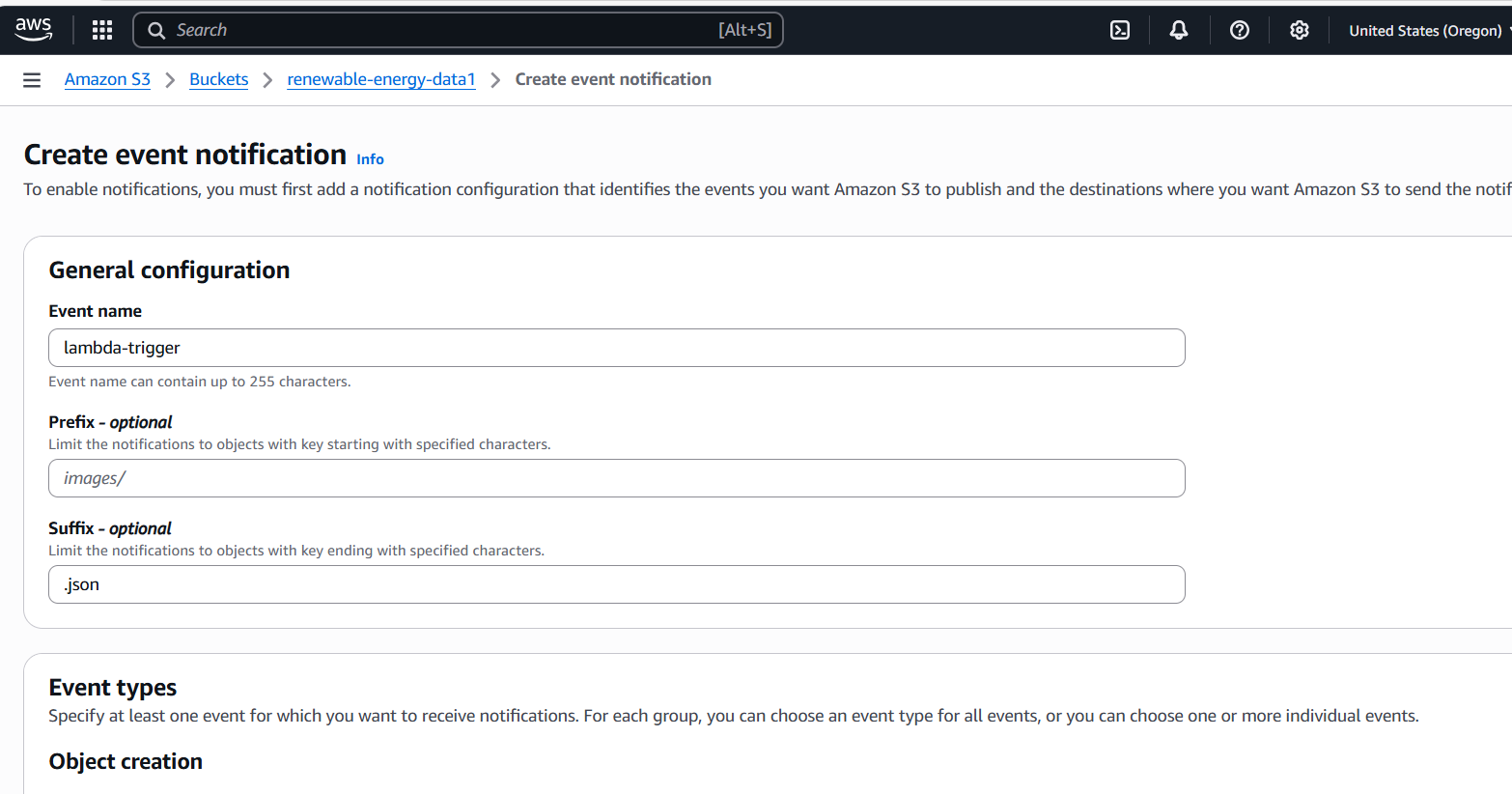
**Create S3 Event Notification to Trigger Lambda**

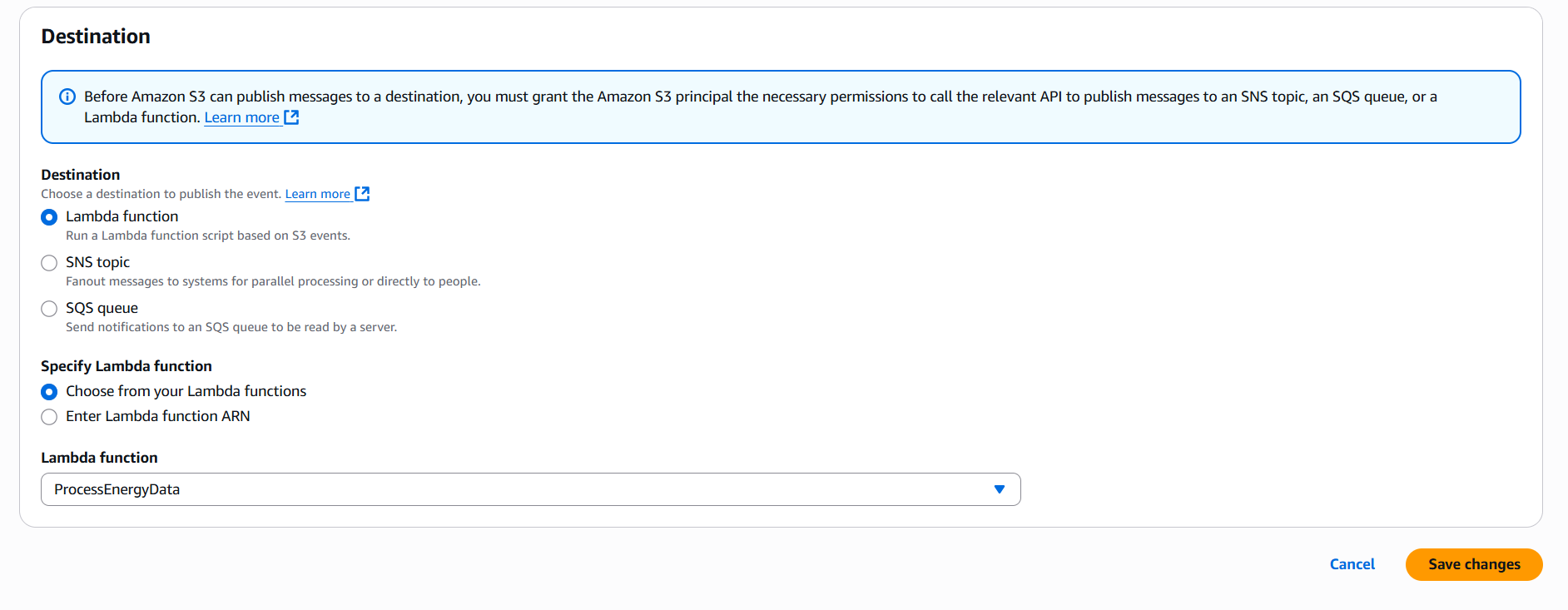
To automate processing whenever new data is uploaded to the S3 bucket, set up an event notification to trigger the Lambda function.

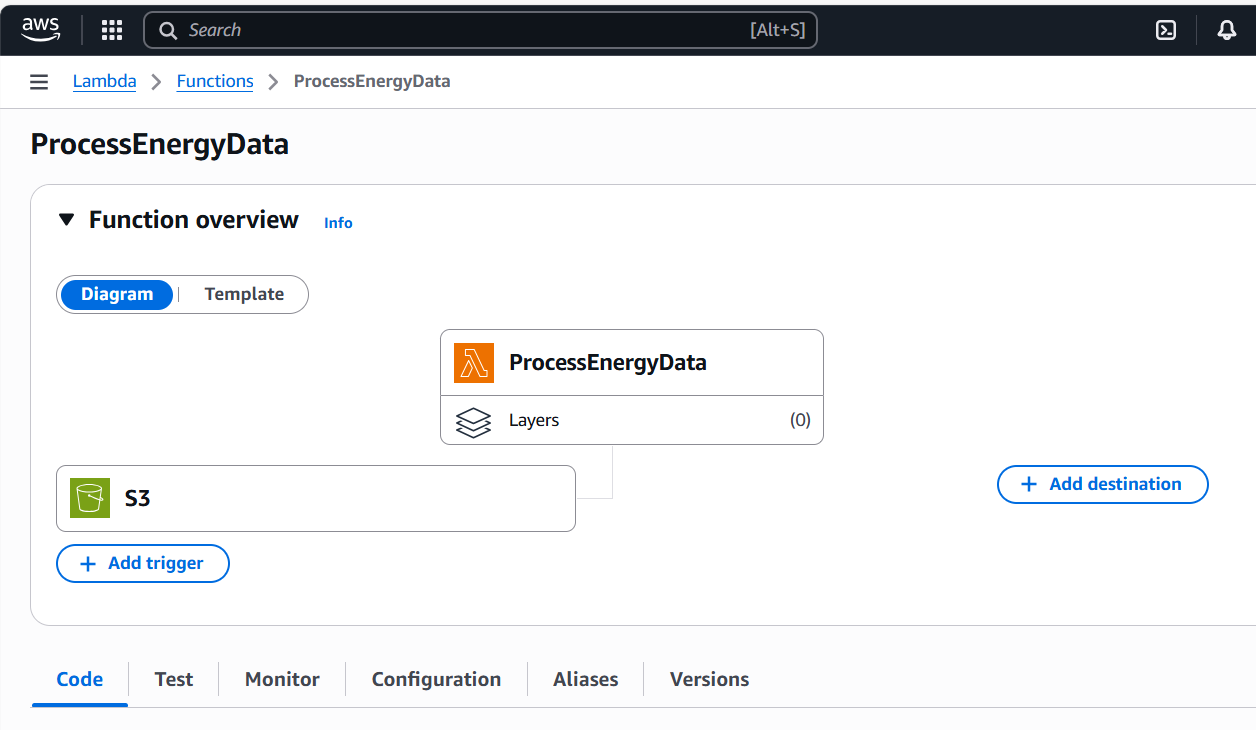
**Steps to follow:**

* Go to the Amazon S3 console and open the bucket (renewable-energy-data1).
* Navigate to the Properties tab.
* Scroll down to Event notifications and click Create event notification.
* Fill in the details:
* Name: lambda-trigger
* Event type: PUT (for new object uploads)
* Prefix: Leave blank or specify if needed
* Suffix: .json
* Destination: Select Lambda function
* Choose your Lambda function (ProcessEnergyData)
* Save the configuration.









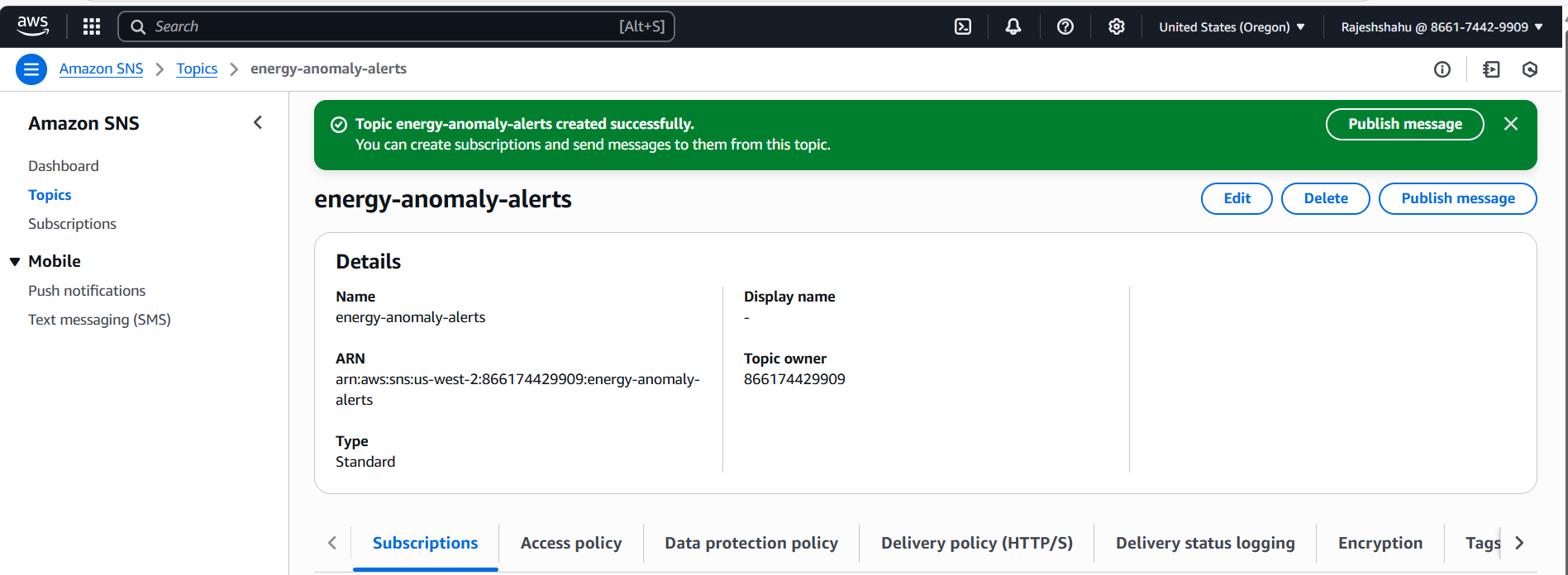
**Amazon SNS – Setup for Real-time Alerts**

After setting up the Lambda function, I integrated **Amazon SNS (Simple Notification Service)** to trigger real-time email alerts whenever an anomaly is detected in the incoming data.

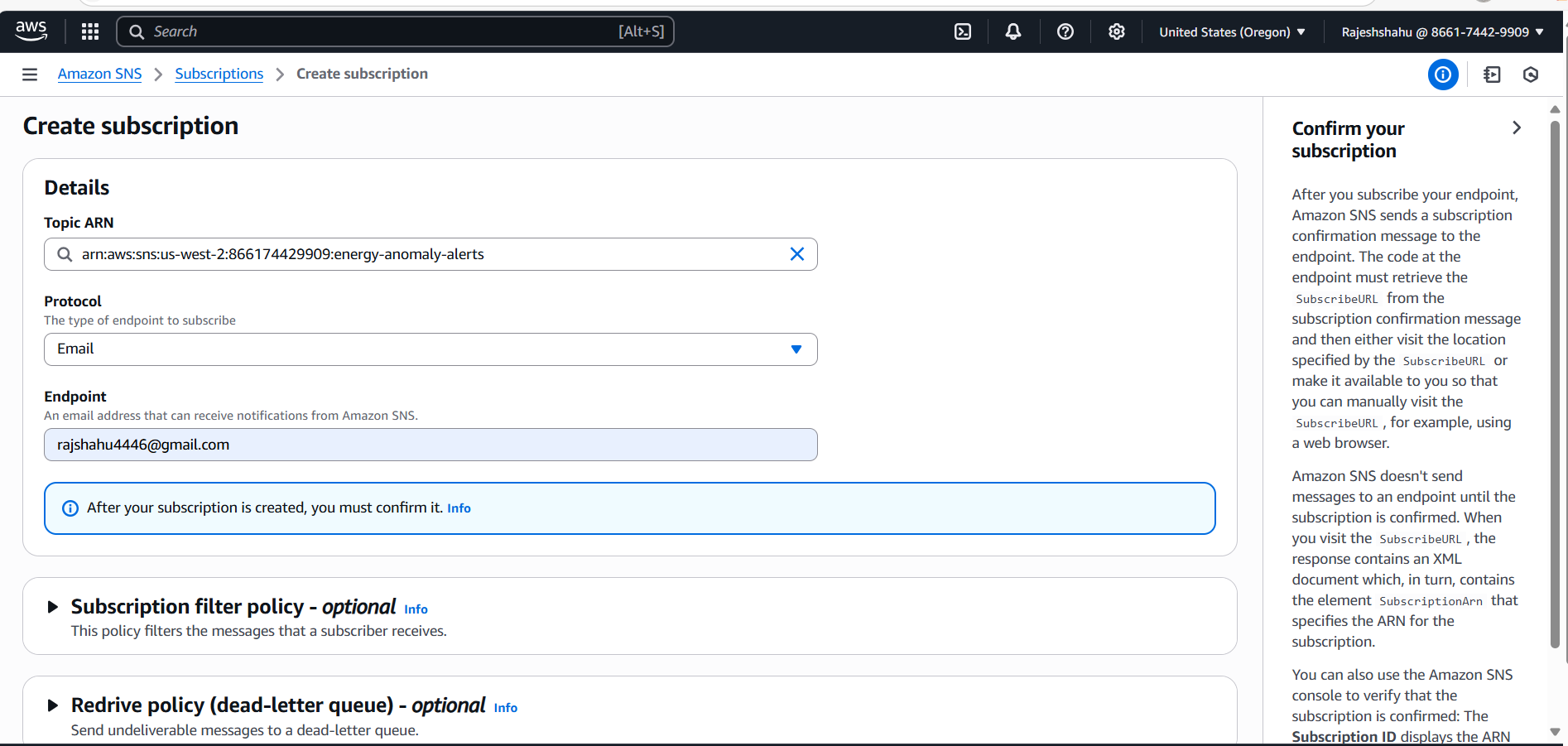
**Steps I Followed:**

1. **Created an SNS Topic**
   * Topic name: energy-anomaly-alerts
   * This acts as a communication channel for alerts.

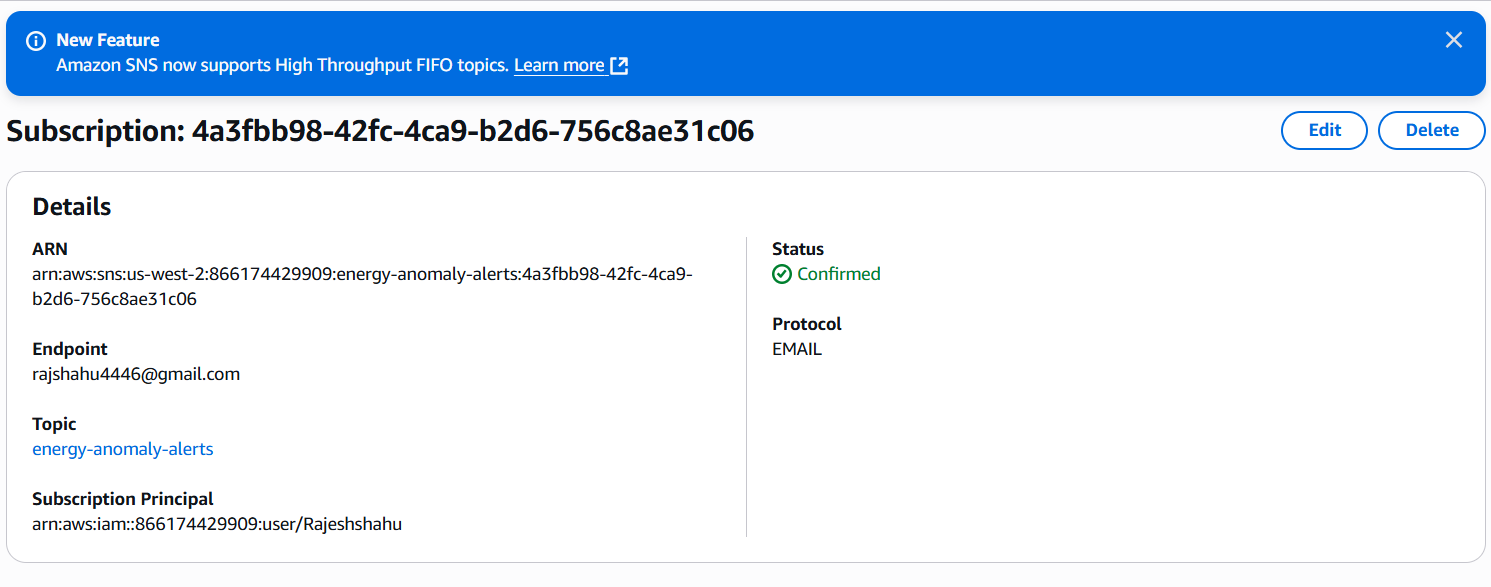




1. **Copied the Topic ARN**
   * Used this ARN in the Lambda function to publish anomaly alerts.
2. **Created a Subscription**
   * Selected **Protocol**: Email
   * Entered my email address to receive alerts.

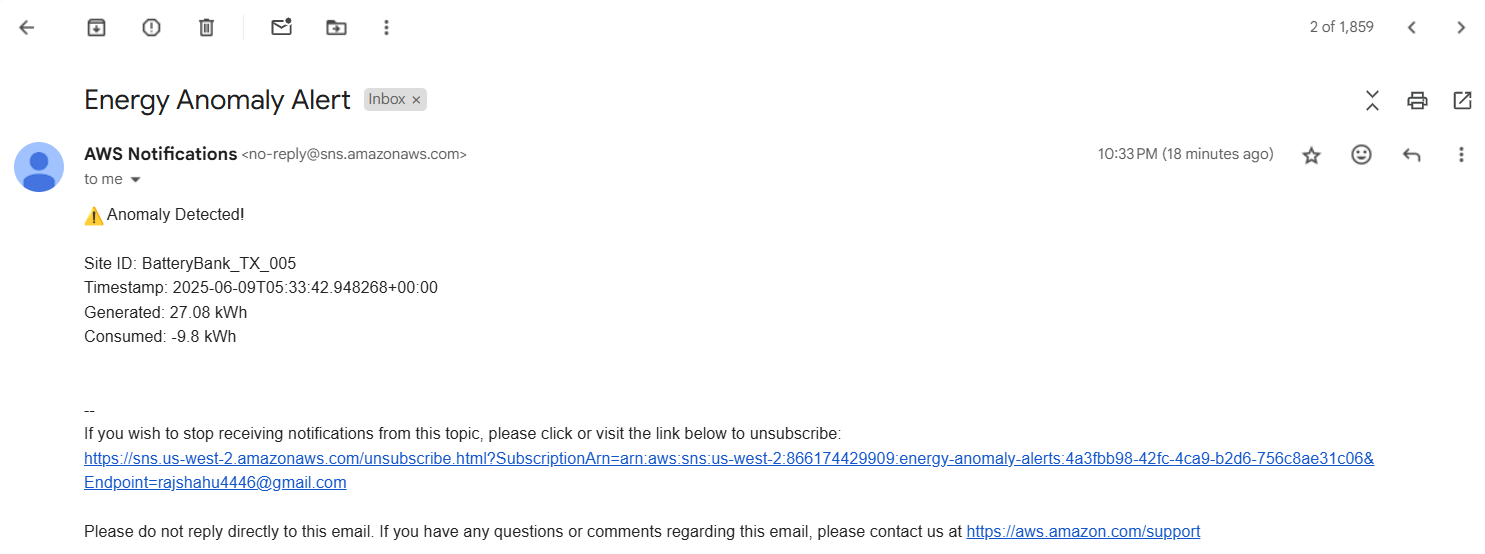


1. **Confirmed the Subscription**
   * Clicked the confirmation link sent to my email inbox to activate the subscription.



1. **Modified Lambda Function to Publish Alerts**
   * Added SNS code inside the Lambda to check for anomalies (energy\_generated\_kwh < 0 or energy\_consumed\_kwh < 0)
   * If found, it publishes a JSON-formatted alert to the SNS topic.

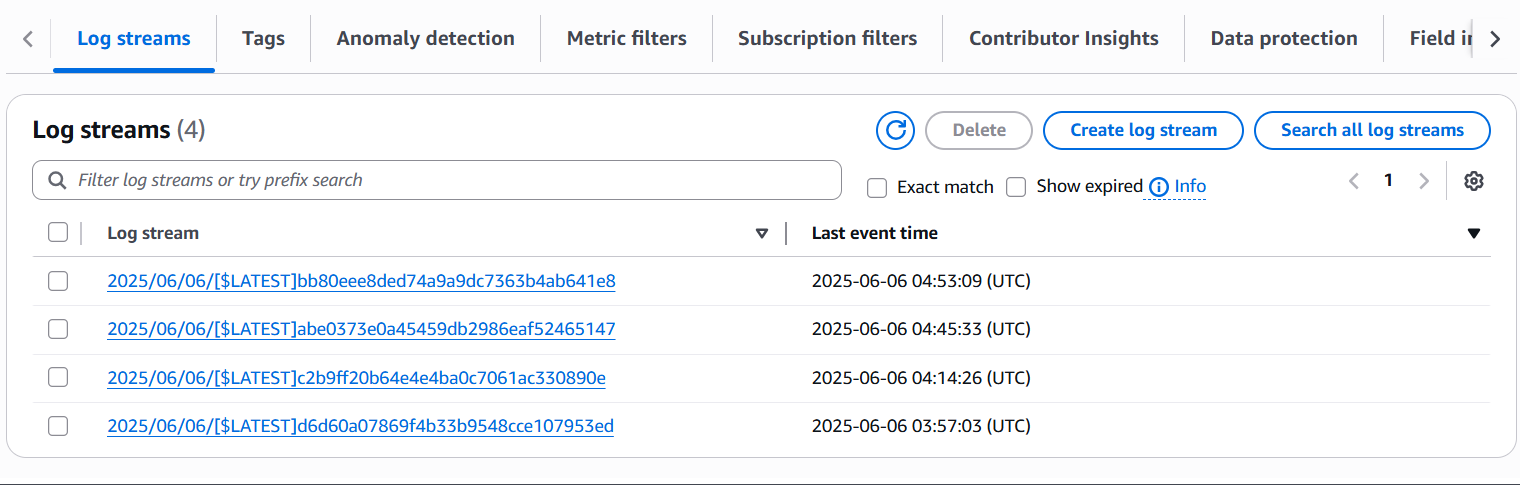
**Generated real-time anomaly alert:**

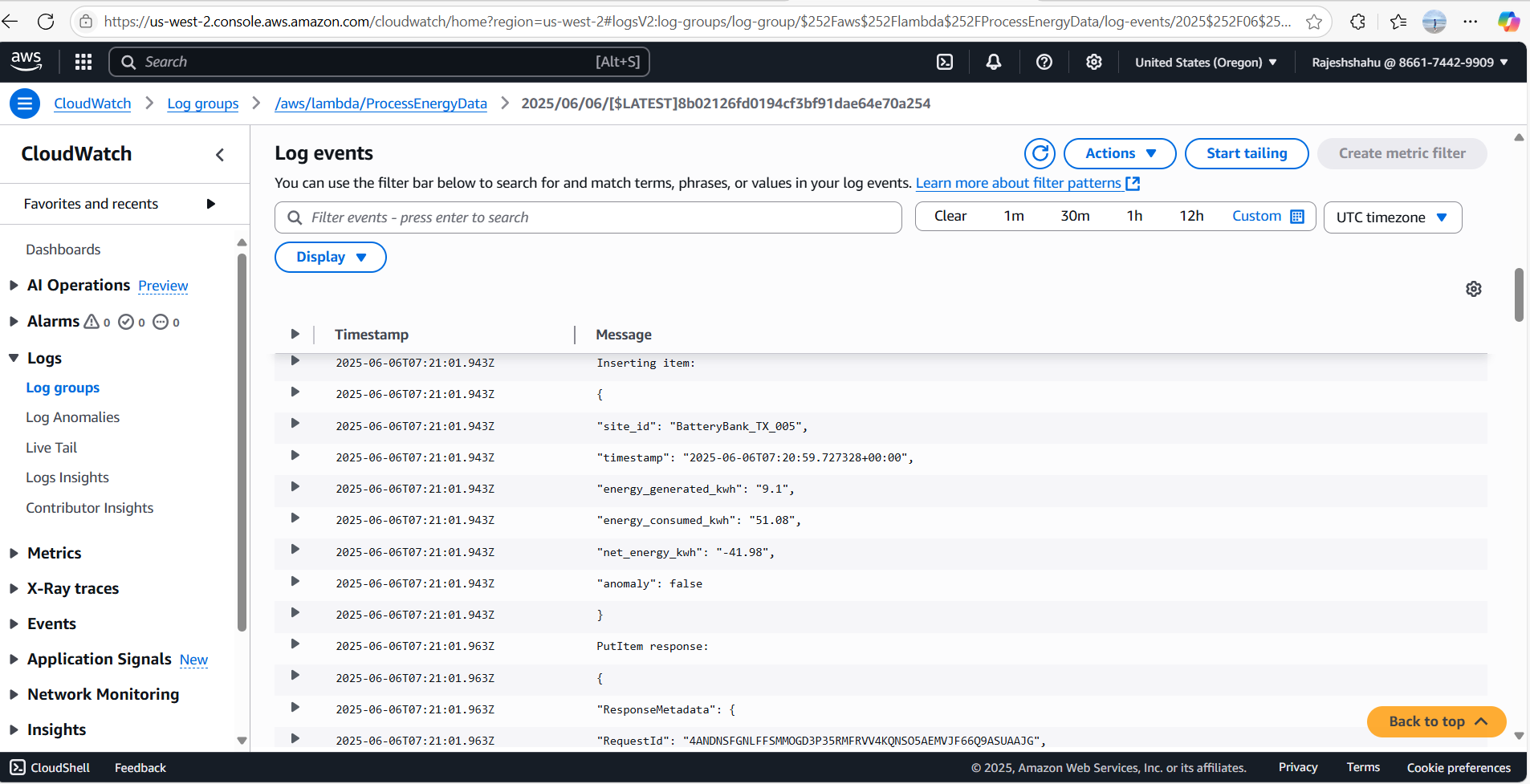
****

**Check Lambda Execution Logs in Cloudwatch**

After deploying the Lambda function and triggering it via S3, I verified the execution by checking the logs in Amazon CloudWatch.

* I navigated to the **CloudWatch** console and clicked on **Logs**.
* Opened the log group named /aws/lambda/ProcessEnergyData.
* Viewed the latest log stream generated after S3 file upload.
* The logs clearly showed:
* Lambda was triggered successfully.
* The file was fetched from S3.
* Each record was parsed and inserted into DynamoDB.
* PutItem responses were logged for confirmation.





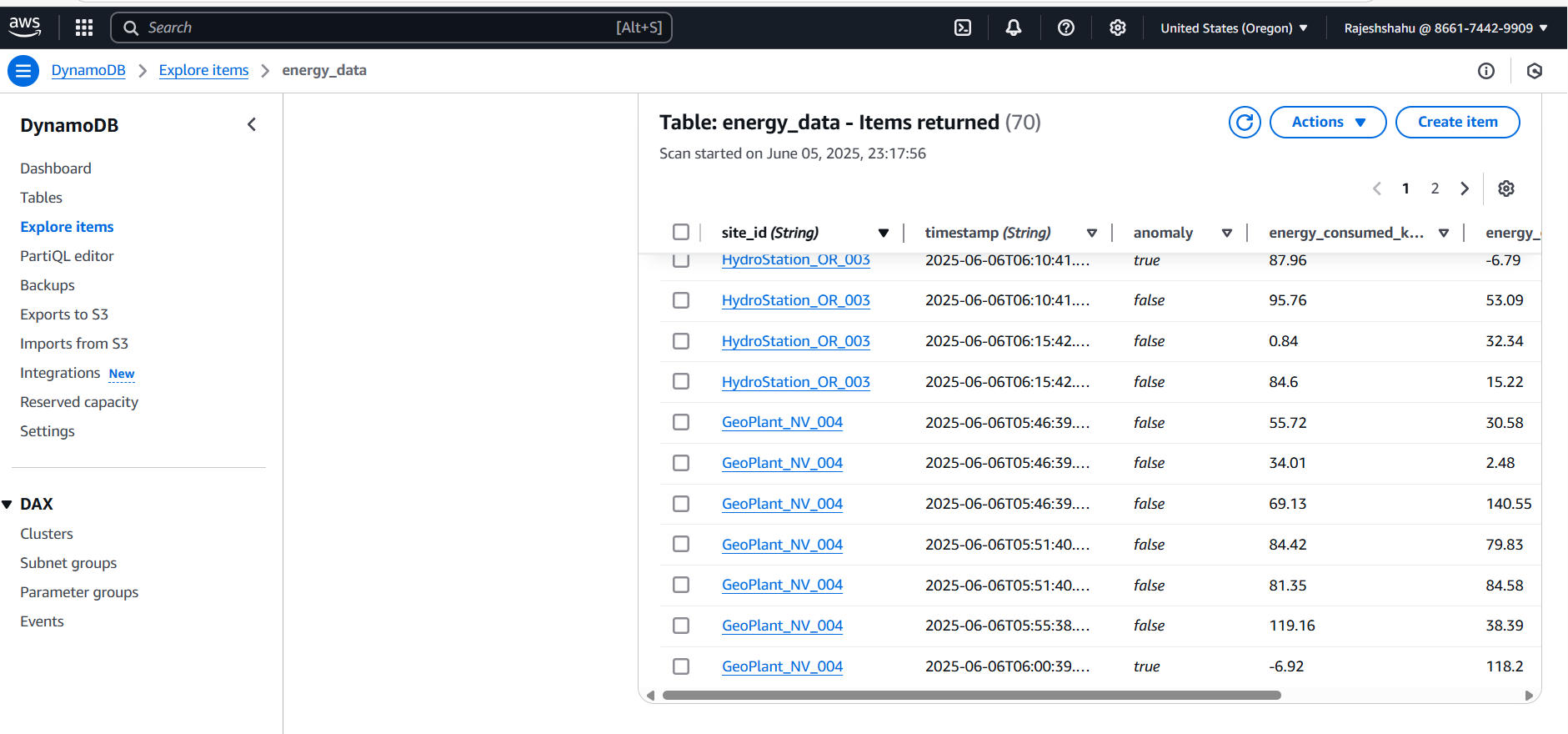
**Verify Data in DynamoDB Table**

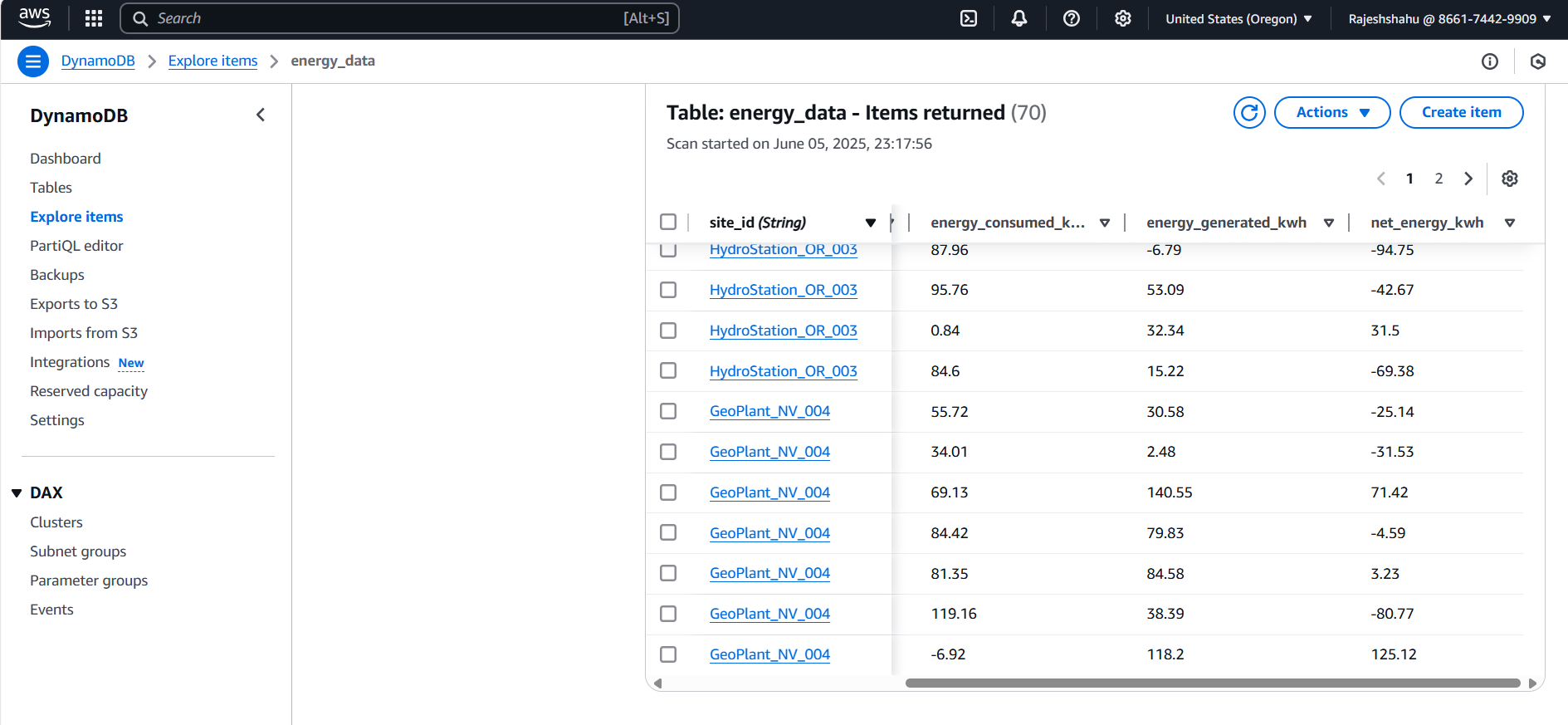
To confirm successful data ingestion, check the DynamoDB table:

* Opened the **Amazon DynamoDB** console.
* Selected table (energy\_data).
* Clicked on **Explore Table Items** feature to check that the data has been added with fields like:
  + site\_id
  + timestamp
  + energy\_generated\_kwh
  + energy\_consumed\_kwh
  + net\_energy\_kwh
  + anomaly

Note: If any data has energy\_generated\_kwh or energy\_consumed\_kwh as negative, those entries should have "anomaly": true.

Observed that new records were continuously being added to DynamoDB every 5 minutes as JSON files arrived in S3.





**Table Schema:**

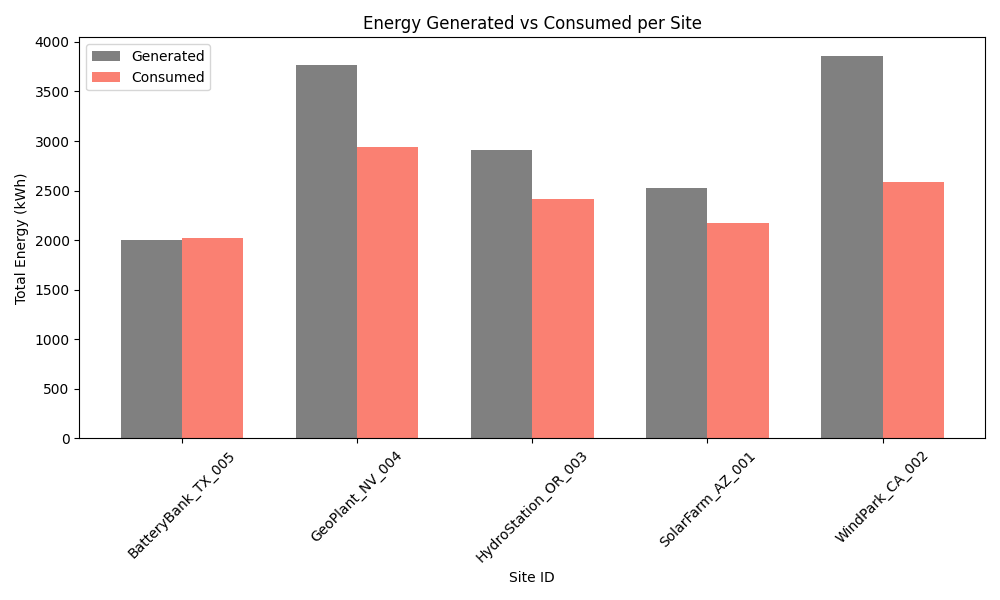
|  |  |  |
| --- | --- | --- |
| Field Name | Type | Description |
| site\_id | String | Partition key – identifies the energy site |
| timestamp | String | Sort key – records the timestamp of event |
| energy\_generated\_kwh | Number(Float) | Amount of energy generated |
| energy\_consumed\_kwh | Number(Float) | Amount of energy consumed |
| net\_energy\_kwh | Number(Float) | Difference between generated and consumed |
| anomaly | Boolean | Flag if anomaly detected |

**Data Visualization**

After loading the processed energy data into DynamoDB, I extracted the records using Python and performed data visualization to uncover key trends and anomalies. These visualizations help interpret energy generation, consumption patterns, and anomaly distributions across different renewable sites.

**1. Energy Generated vs. Consumed per Site**

To analyze the energy performance across different renewable energy sites, I created a grouped bar chart comparing total energy generated and total energy consumed per site.

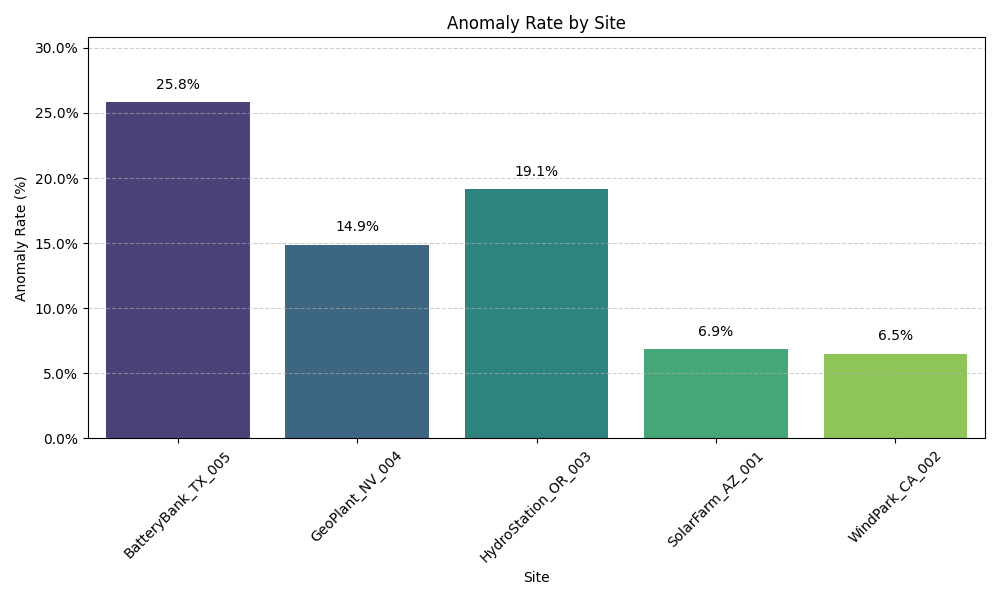


**Key Observations:**

* WindPark\_CA\_002 and GeoPlant\_NV\_004 have the highest total energy generation.
* BatteryBank\_TX\_005 shows very close values for generation and consumption, indicating a more balanced or storage-based setup.
* In all cases, energy generation exceeds consumption, suggesting energy surplus or efficient production across sites.

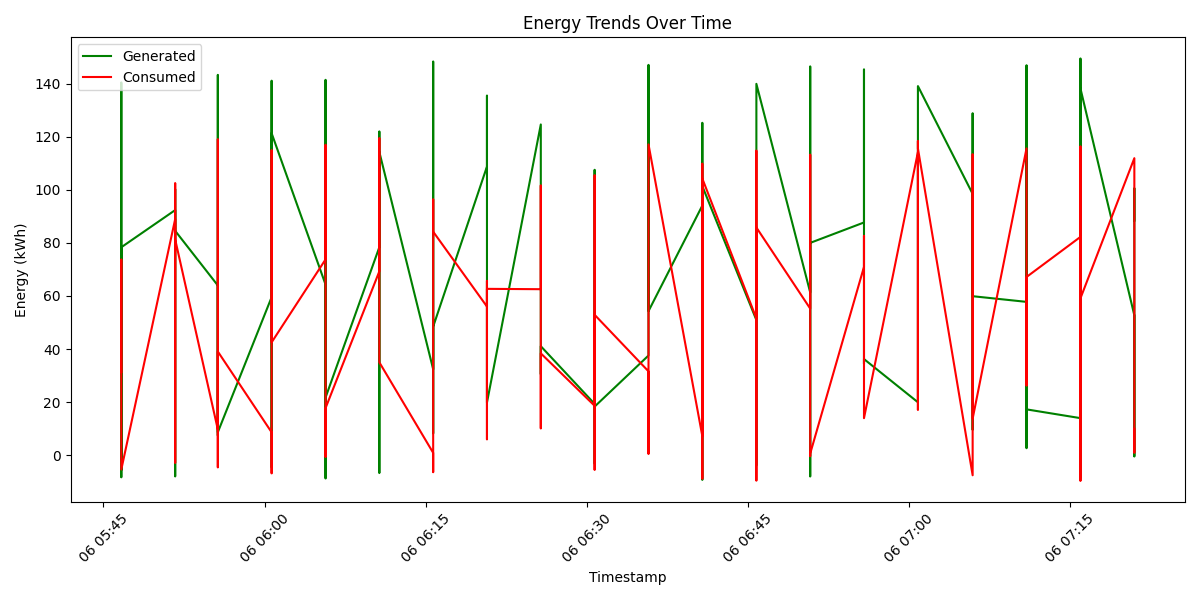
**2. Anomaly Rate by Site**

This visualization highlights the percentage of anomalous readings for each energy site. It clearly indicates that BatteryBank\_TX\_005 exhibits the highest anomaly rate at 25.8%, suggesting potential sensor errors or irregular energy behavior. In contrast, WindPark\_CA\_002 and SolarFarm\_AZ\_001 show minimal anomalies, indicating relatively stable performance.



**3. Energy Trends Over Time**

This chart shows the variation of energy generation and consumption over time. The data reveals consistent fluctuations between generated (green) and consumed (red) energy. Spikes or gaps indicate operational anomalies, load changes, or potential sensor noise. This view helps analyze system stability and detect irregular energy behaviors.



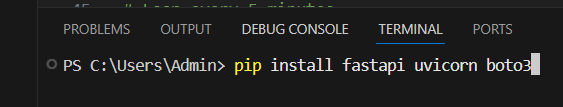
**FastAPI Setup & API Development**

After verifying that data was successfully being inserted into DynamoDB and visualization, I developed a **FastAPI application** to expose the data through RESTful API endpoints.

**Setup:**

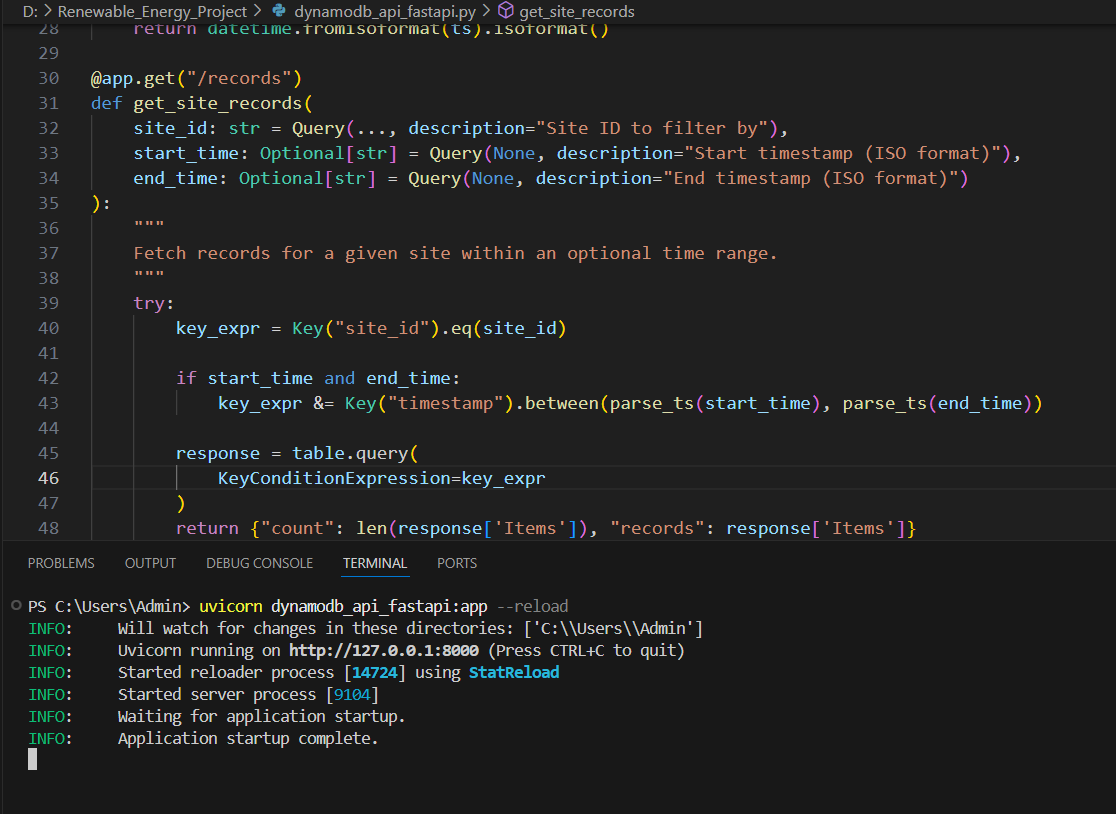
1. Installed required Python packages:

*pip install fastapi uvicorn boto3*



1. Created a Python script dynamodb\_api\_fastapi.py containing:
   * /records: To fetch records by site\_id and time range.
   * /anomalies: To fetch all records marked as anomalies.
2. Ran the FastAPI app using:

*uvicorn dynamodb\_api\_fastapi:app –reload*

**

1. Accessed the interactive Swagger documentation:
   * URL: <http://localhost:8000/docs>

**API Outputs**

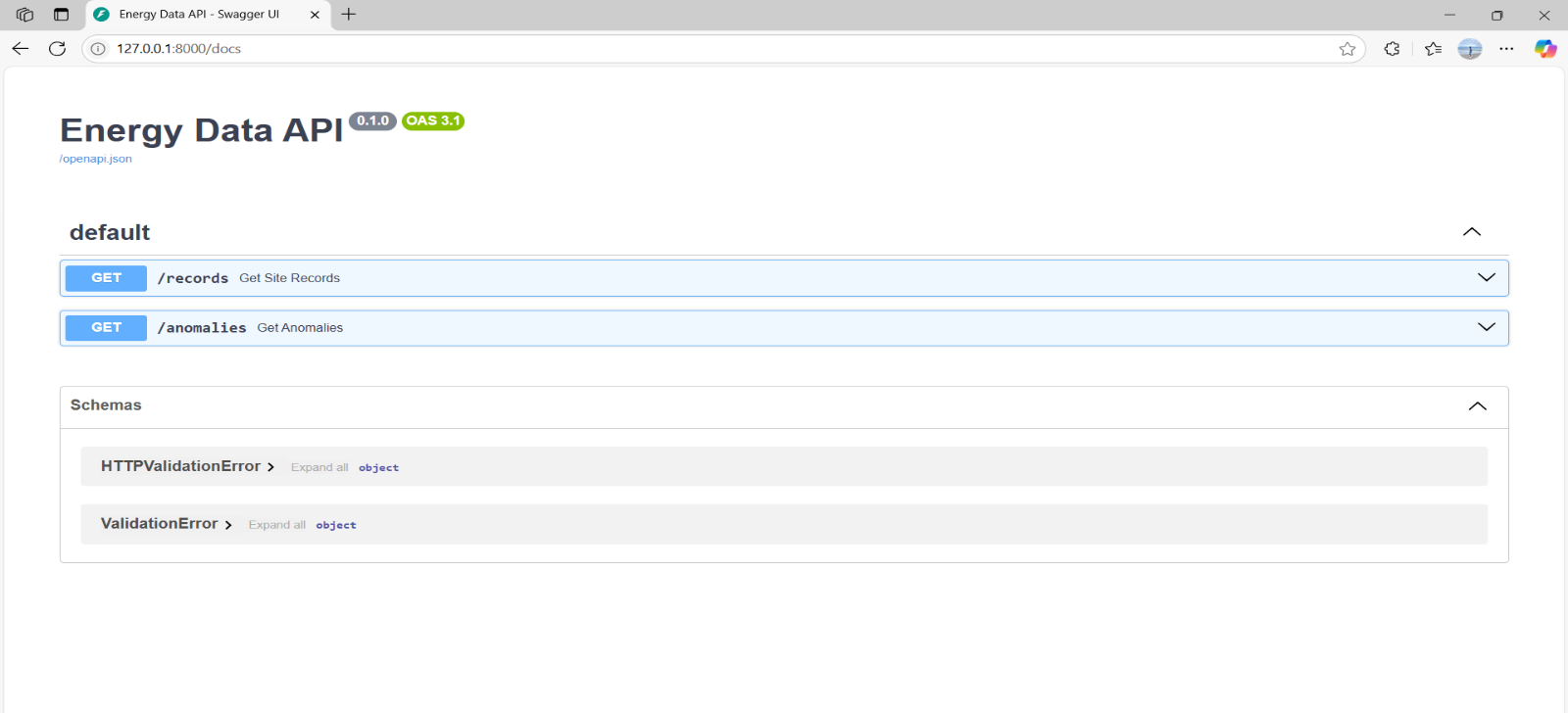
* Queried /records for a specific site and timeframe.
* Queried /anomalies to see which entries had anomaly=true.

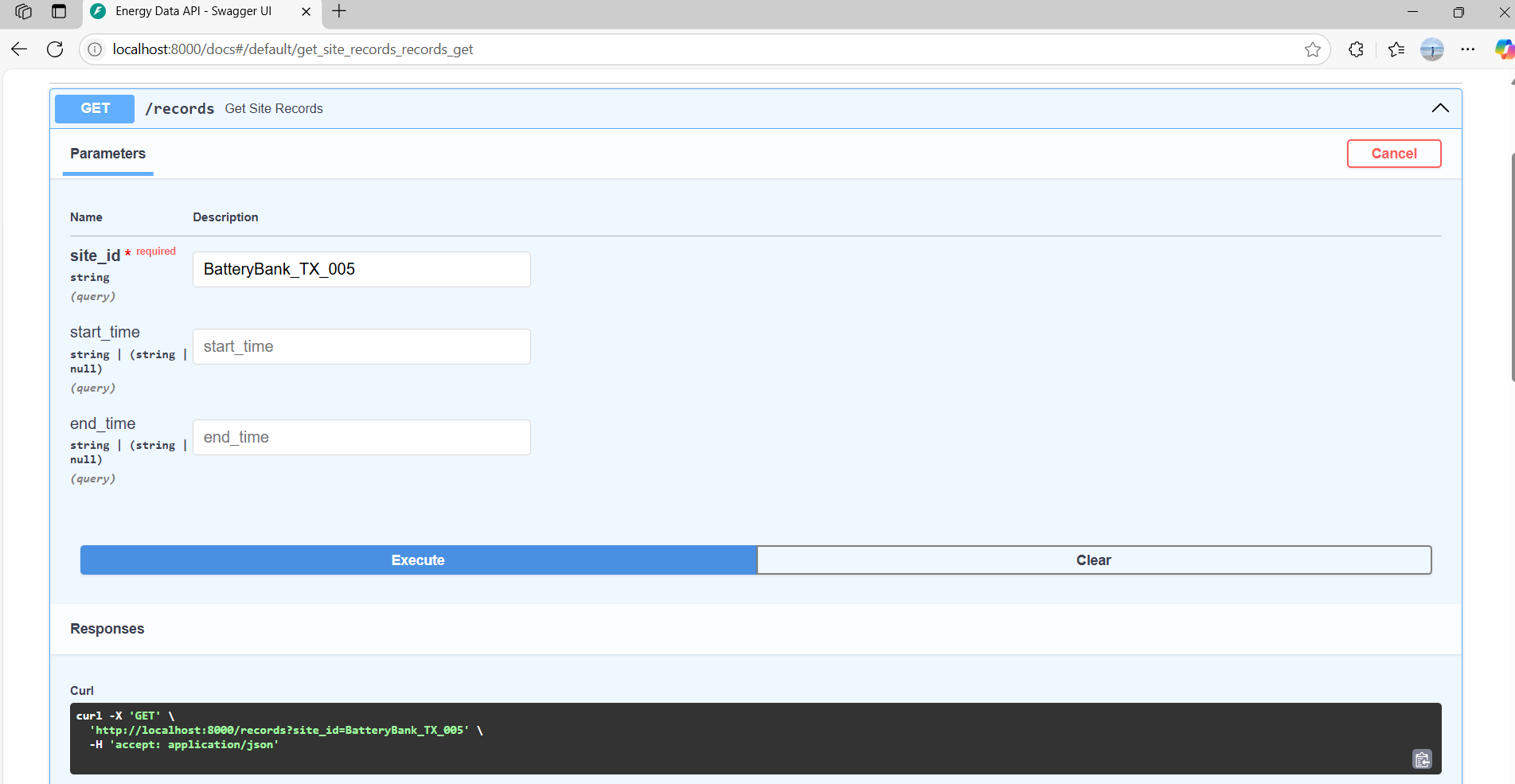
I created two main endpoints:

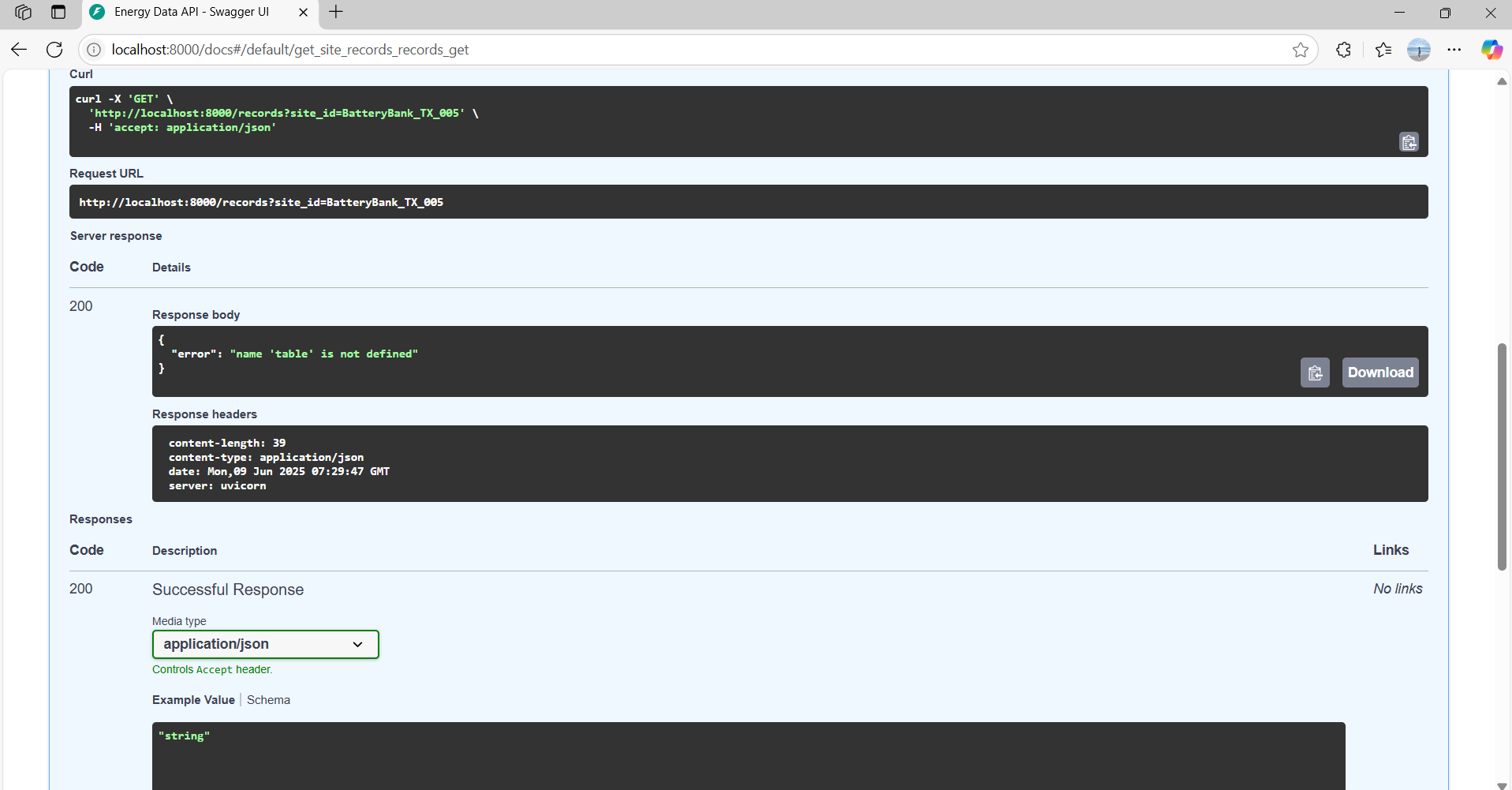
* GET /records: This lets me fetch all energy records for a specific site\_id, with optional start\_time and end\_time filters to narrow the results.
* GET /anomalies: This endpoint returns only the records where anomaly is True, helping me easily identify faulty or unusual energy readings for a given site.

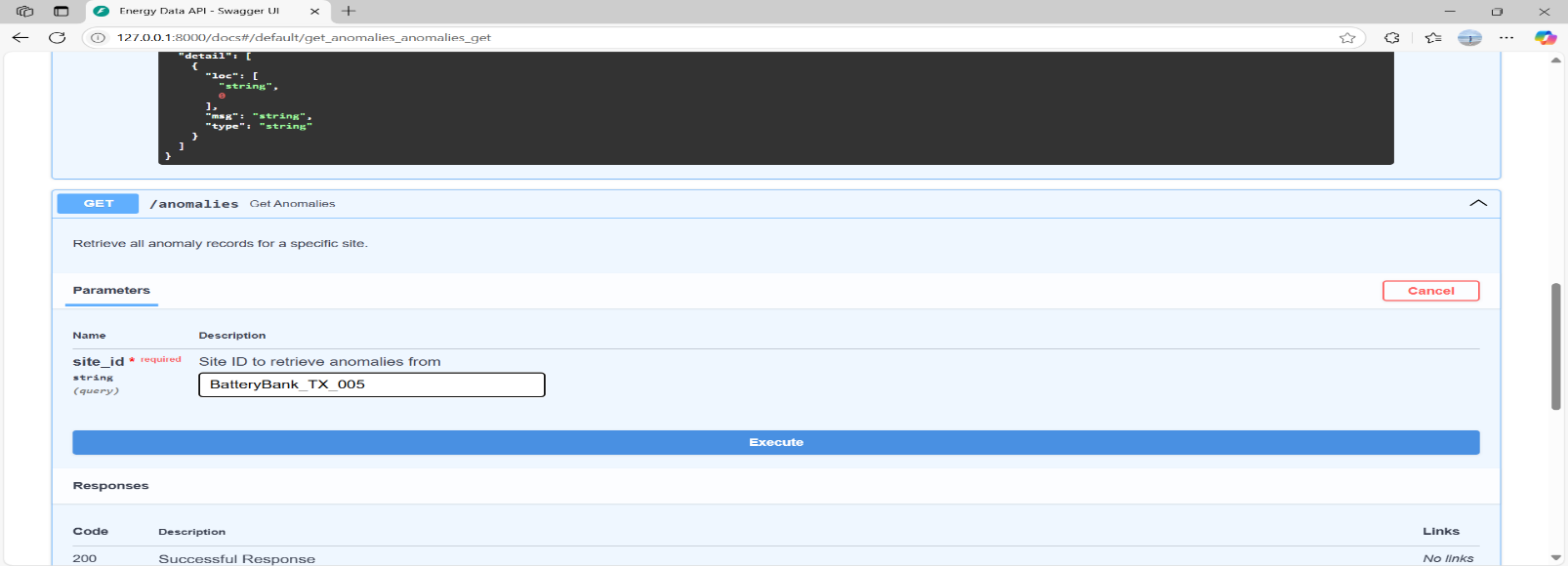
I tested both endpoints using Swagger UI (http://127.0.0.1:8000/docs), which FastAPI generates automatically. The results confirmed that:

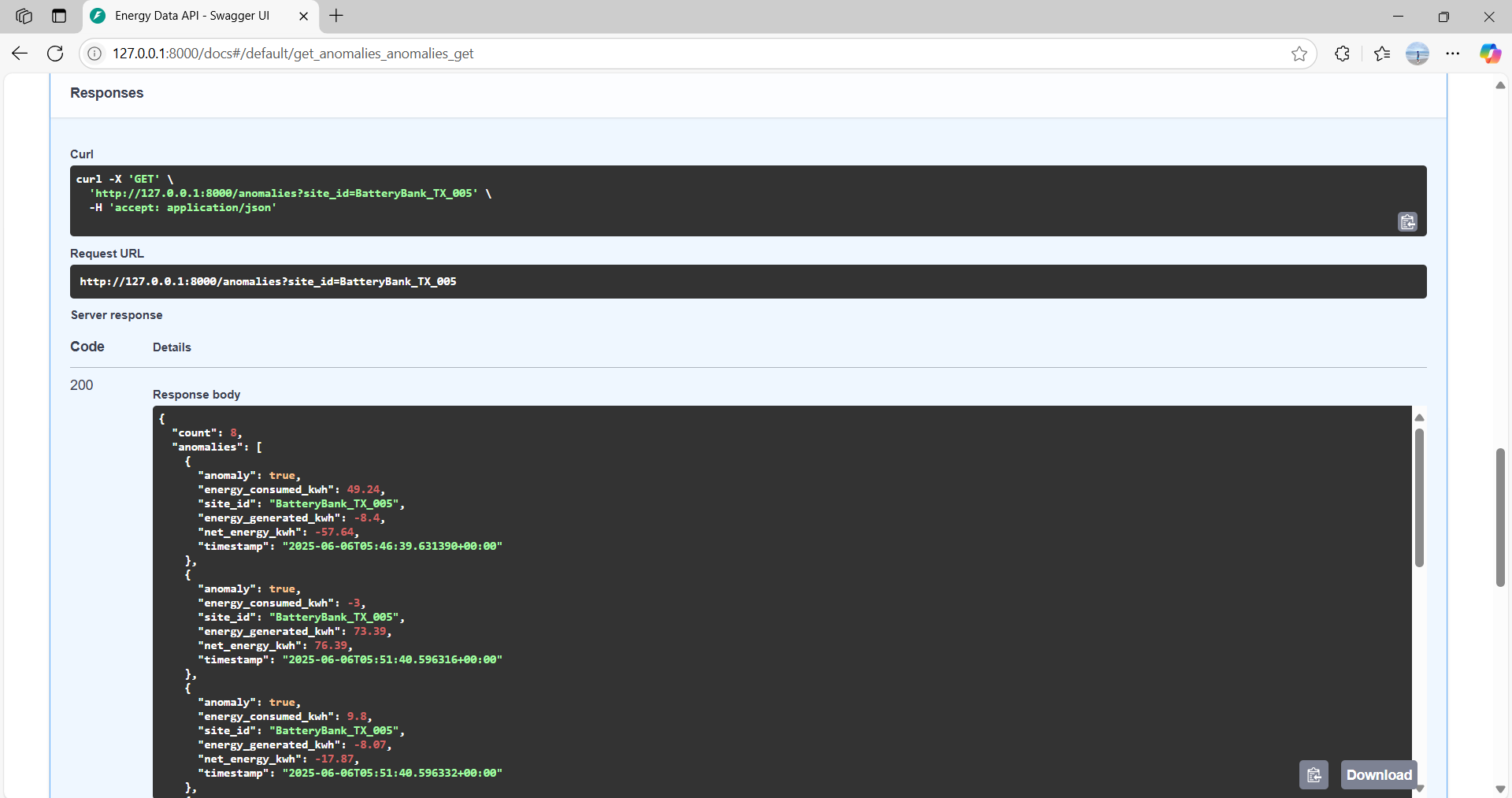
* Records were being correctly retrieved based on site and time filters.
* Anomalies were detected and displayed accurately for each energy site.

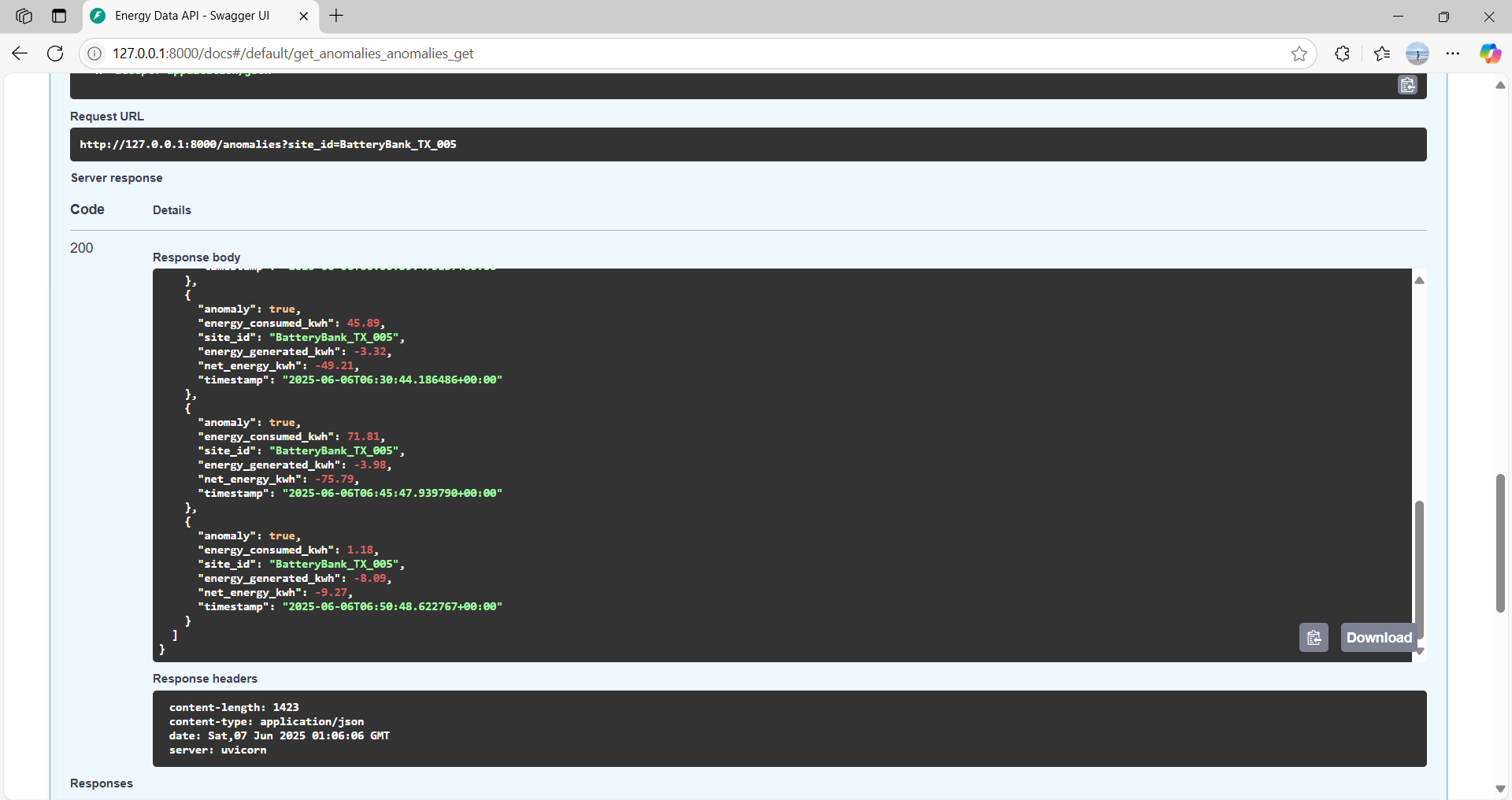












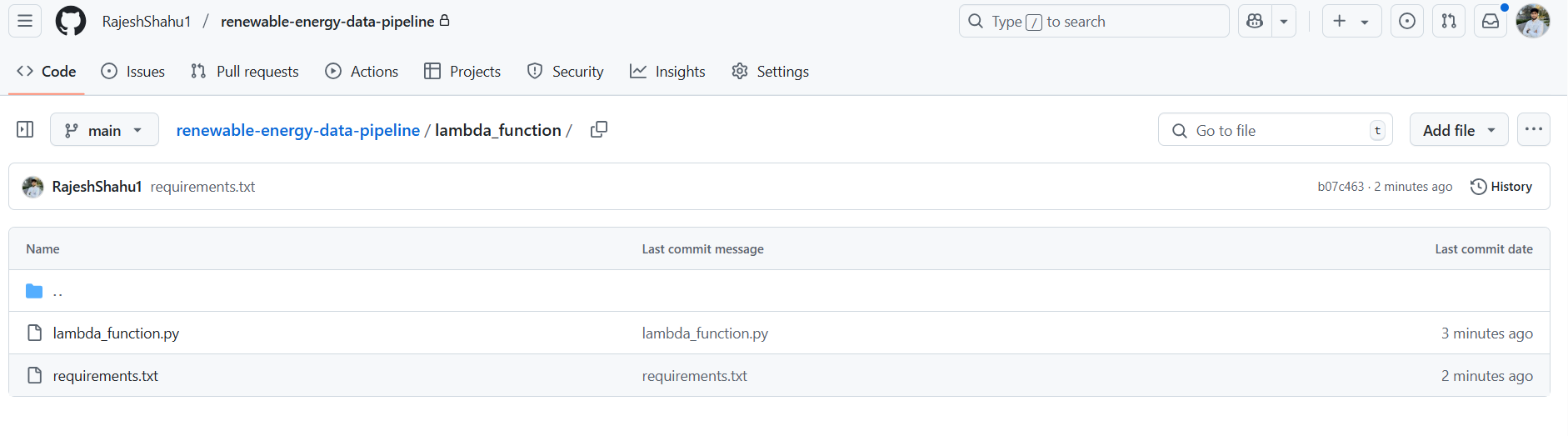
**GitHub CI/CD Pipeline Integration**

To streamline and automate the deployment of our “ProcessEnergyData” Lambda function, I implemented a CI/CD pipeline using GitHub Actions. This ensures that any updates pushed to the repository are automatically deployed to AWS without manual intervention.

**Folder Structure**

A folder lambda\_function/ was created in the root of the GitHub repository. This folder contains:

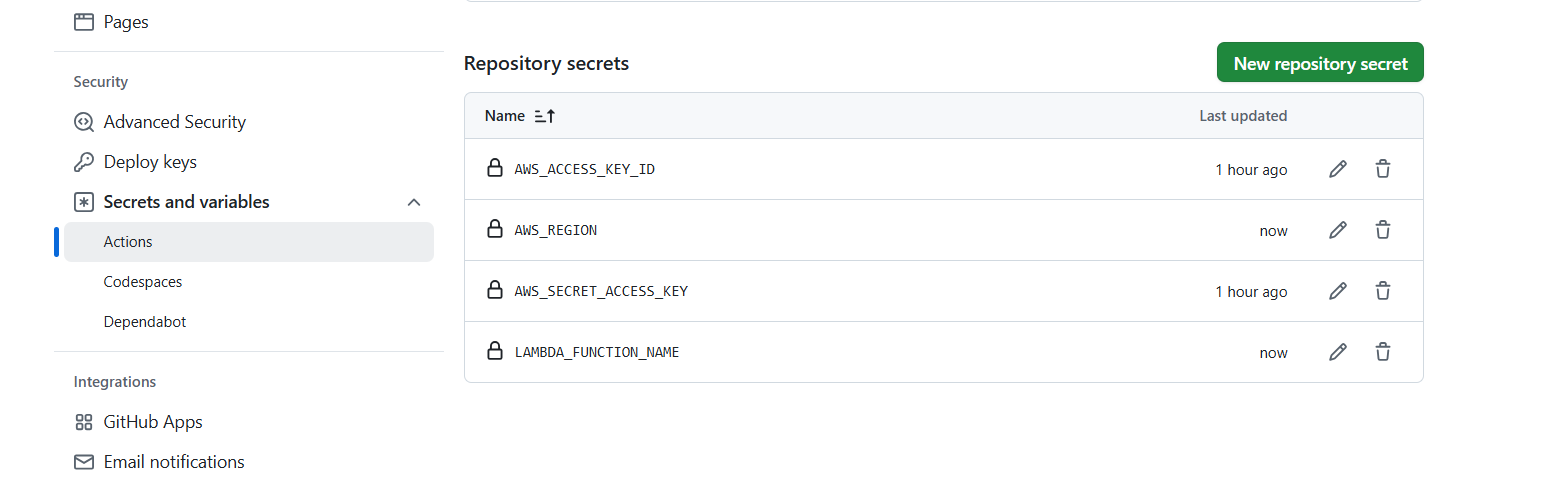
* lambda\_function.py – The actual Lambda handler script
* requirements.txt – Lists Python dependencies (boto3)

**GitHub Secrets Configuration**

To allow GitHub Actions to authenticate with AWS securely, I configured the following secrets under:

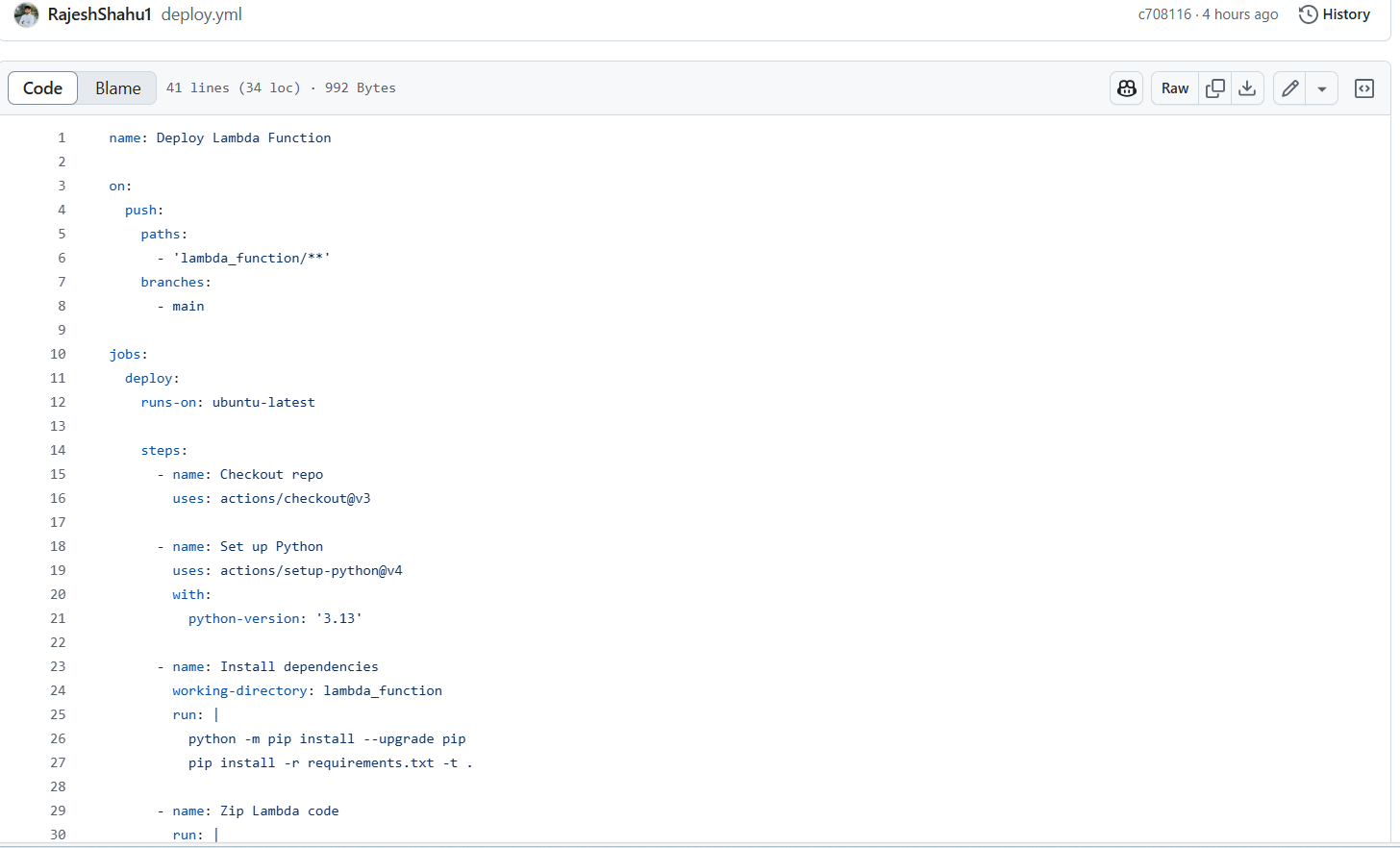
GitHub → Settings → Secrets → Actions → New repository secret

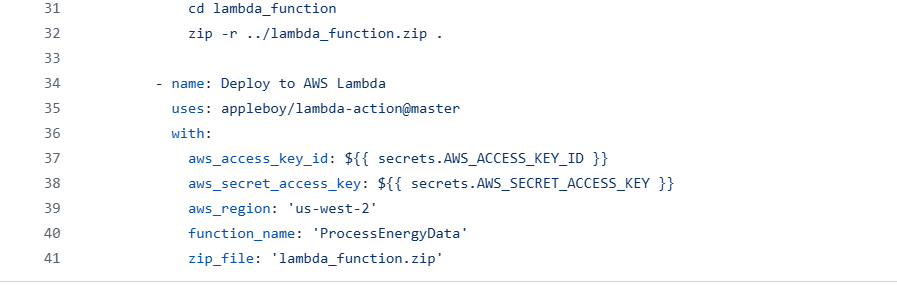
|  |  |
| --- | --- |
| **Secret Name** | **Purpose** |
| AWS\_ACCESS\_KEY\_ID | IAM user's access key ID |
| AWS\_SECRET\_ACCESS\_KEY | IAM user's secret key |
| AWS\_REGION | Deployment region (us-west-2) |
| LAMBDA\_FUNCTION\_NAME | Name of the Lambda function (ProcessEnergyData) |



**Workflow File Setup**

Created .github/workflows/deploy.yml directory with the following logic:





**How It Works**

1. On every **push to main**, the workflow is triggered.
2. It installs dependencies and zips the Lambda function code.
3. It deploys the zip package to AWS Lambda using the AWS CLI.
4. The deployment is now **fully automated**, consistent, and version-controlled.

This CI/CD setup improves the reliability, scalability, and maintainability of our cloud-based data pipeline.