**Solution Overview and Design Decisions**

**Objective**

In this project, my goal was to build a real-time, serverless data pipeline that could simulate and process energy generation and consumption data coming from multiple renewable energy sites. I wanted to detect anomalies as the data came in, store it efficiently, and make it accessible through REST APIs and visualizations.

**My Architecture Approach**

I decided to go with a fully serverless, event-driven architecture to keep things scalable and low-maintenance. The data flow I implemented looks like this:

**S3 → Lambda(SNS, Coudwatch) → DynamoDB → FastAPI → Visualization**

**Why I Chose These Components**

**Python for Simulation**

I used Python to write a data generator that continuously simulates energy data for different sites. It outputs JSON records with timestamps, which mimics real-time feeds.

**Amazon S3**

I used S3 to act as the landing zone for incoming data. Each time a file gets uploaded, it automatically triggers the Lambda function. I picked S3 because it’s durable, cheap, and integrates well with other AWS services.

**AWS Lambda**

Lambda was perfect for this since I didn’t want to manage servers. Every time new data arrives in S3, Lambda runs my script to validate, transform, and check for anomalies in the records.

**Amazon DynamoDB**

I stored the processed data in DynamoDB. It gave me fast lookups based on site\_id and timestamp, and it scales easily with the time-series nature of energy data.

**Amazon SNS**

When the Lambda function finds an anomaly, it sends an alert using SNS. I configured it to email me whenever abnormal data (like negative or too-high energy values) is detected.

**FastAPI**

To expose the data, I created a FastAPI backend. It lets users query records and anomalies through clean REST endpoints. I like FastAPI because it's fast, easy to build with, and comes with Swagger UI by default.

**Amazon CloudWatch**

For logs and metrics, I used CloudWatch. It helped me debug Lambda executions and track how the pipeline was performing.

**Seaborn / Matplotlib / Plotly**

I used Seaborn and Matplotlib to create static charts, and Plotly for interactive plots. This helped visualize energy trends and detect patterns over time.

**GitHub Actions for CI/CD**

To automate deployment, I added a GitHub Actions workflow. Now, every time I push changes to the main branch, my Lambda function and infrastructure get deployed automatically.

**AWS CLI + Bash Scripts**

I wrote shell scripts to simplify the deployment of AWS resources.

**Design Choices I Made**

* **Serverless-first** so I didn’t have to manage any infrastructure.
* **Real-time alerts** using SNS and logs in CloudWatch.
* **Modular design** – each component (data generation, processing, API, visualization) is independent.
* **Ease of use** – I made sure setup is clear and APIs are documented via Swagger.