ECE496 Documentation

Project Title: Cloud backend system for remote monitoring system

Supervisor Name: Jorg Liebeherr

Team Members: Mohammed Ozair, Rahul Chopra, Fabin Flasius, Simrah Najeeb

1. Documentation for Gateway Code:

- Purpose: The sketch acts as the firmware for a gateway device responsible for hosting a web server and transmitting sensor data to an MQTT manager. The documentation provides an overview of the sketch's functionality and how it interacts with the `GatewayWebServerModule` and `GatewayMqttManager` classes.
- Setup:
    - The `setup()` function initializes serial communication and starts the web server using the `begin()` method of the `GatewayWebServerModule` instance (`gwServer`).
- Loop:
    - In the `loop()` function, web server operations are executed using the `loop()` method of the `GatewayWebServerModule` instance.
    - Every 10 seconds, the sketch generates random sensor readings and constructs a timestamp to represent data from 60 days ago.
    - If there are sensors available, it generates random readings for each sensor and sends them to the MQTT manager using the `generateAndSendSensorReadings()` method of the `GatewayWebServerModule` instance.
- Sensor Reading Transmission:
    - The sketch simulates sensor data transmission by generating random sensor readings and sending them to the MQTT manager.
    - It uses node IDs to identify the source of the sensor data and includes a timestamp to represent the time of data collection.
- Note:
    - This sketch assumes the existence of the `GatewayWebServerModule` and `GatewayMqttManager` classes, which handle web server operations and MQTT communication, respectively.

Now, Let's Delve Deeper into the `GatewayWebServerModule` and `GatewayMqttManager` classes:

GatewayMqttManager

- Purpose: `GatewayMqttManager` class manages MQTT communication for the gateway device.

- Constants:
    - `MAX_SENSOR_TYPES`: Maximum number of sensor types supported by the gateway.
- Private Member Variables:
    - `WiFiClientSecure net`: Secure WiFi client instance for MQTT communication over TLS.
    - `PubSubClient client`: MQTT client instance for handling MQTT communication.
    - `String systemName`: Name of the system associated with the gateway.
    - `String gatewayId`: Unique ID assigned to the gateway.
    - `String privateKey`: Private key used for authentication with the MQTT broker.
    - `String accessKey`: Access key used for authentication with the MQTT broker.
    - `String publishTopicBeforeInitialization`: Topic for publishing initialization message before AWS connection.
    - `String publishTopicAfterInitialization`: Topic for publishing messages after AWS connection.
    - `String latestControlMessage`: Latest control message received from the MQTT broker.
    - `String latestSensorMessage`: Latest sensor message received from the MQTT broker.
    - `int sensorTypes[MAX_SENSOR_TYPES]`: Array to store sensor types.
    - `int sensorCount`: Number of sensor types configured.
- Static Method:
    - `static void messageReceived(char* topic, byte* payload, unsigned int length)`: Static method to handle incoming MQTT messages.
- Public Member Functions:
    - Constructor:
        - `GatewayMqttManager()`: Initializes member variables and client instances.
    - Accessors and Mutators:
        - Methods to set and get system name, private key, access key, gateway ID, publish topics, sensor types, and sensor count.
    - Initialization:
        - `generateGatewayId()`: Generates a unique gateway ID based on ESP32 chip ID.

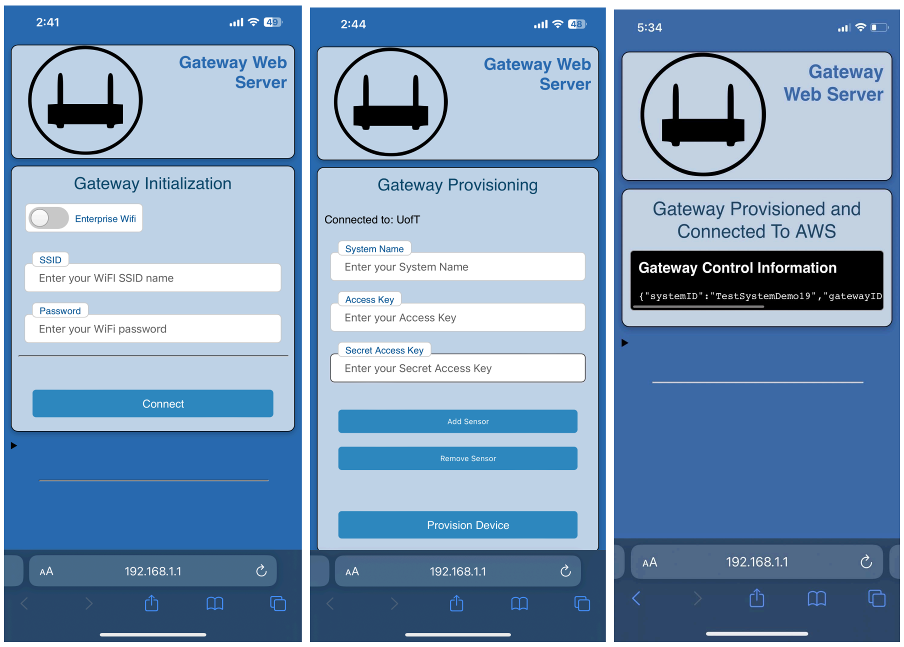
        - `initializeClient()`: Initializes the MQTT client with necessary configurations.
        - `connectAWS(char thingName[])`: Connects to AWS IoT Core with the specified Thing Name.
        - `isConnected()`: Checks if the MQTT client is connected to the broker.
        - `disconnect()`: Disconnects the MQTT client from the broker.
    - Message Handling:
        - `subscribeToTopics()`: Subscribes to relevant MQTT topics.
        - `messageReceived()`: Callback method invoked when an MQTT message is received.
        - `updateControlMessage(const String& message)`: Updates the latest control message.
        - `updateSensorMessage(const String& message)`: Updates the latest sensor message.
    - Message Publishing:
        - `publishInitializationMessage()`: Publishes an initialization message to the MQTT broker.
        - `constructSensorReadingMessage(int currentNode, int parentNode, const int sensorReadings[], unsigned long timestamp)`: Constructs and publishes a sensor reading message to the MQTT broker.
    - Loop:
        - `loop()`: Handles MQTT client loop operations.

GatewayWebServerModule

- Purpose: `GatewayWebServerModule` class manages the web server operations for the gateway device.
- Private Member Variables:
    - `AsyncWebServer server`: Instance of the AsyncWebServer class for handling HTTP requests.
- Public Member Functions:
    - Constructor:
        - `GatewayWebServerModule(int port)`: Initializes the web server with the specified port.
    - Initialization and Loop:

  - `begin()`: Starts the web server.
  - `loop()`: Handles DNS requests and MQTT client loop operations.
- Route Handling:
  - `addRoute(const String& uri, WebRequestMethodComposite method, ArRequestHandlerFunction onRequest)`: Adds a custom route to the server.
- Sensor Data Handling:
  - `generateAndSendSensorReadings(int currentNode, int parentNode, const int sensorReadings[], unsigned long timestamp)`: Generates and sends sensor readings.
- Accessors:
  - `getSensorCount()`: Gets the number of sensors configured.

Additionally, the index_html.h file has the Frontend of the Gateway Web Server which looks something like this.



Connected to: UofT

Gateway Provisioned and Connected To AWS

```
{"systemID":"TestSystemDemo19","gatewayID
```

Finally let's have a look at main sketch file:

Arduino Sketch

- Purpose: This sketch defines the setup and loop functions for a gateway device that hosts a web server and sends sensor readings to an MQTT manager.
- Setup Function:
  - `setup()`: Initializes serial communication and starts the web server.
- Loop Function:
  - `loop()`: Executes web server operations in the loop and sends sensor readings to the MQTT manager every 10 seconds.
- Sensor Reading Transmission:
  - Every 10 seconds, the sketch generates random sensor readings for each available sensor.
  - It then generates a timestamp and subtracts 60 days from the current timestamp.
  - If there are sensors available (determined by `gwServer.getSensorCount()`), it constructs an array to store the sensor readings.
  - For each sensor, it generates a random reading and stores it in the array.
  - Finally, it calls `gwServer.generateAndSendSensorReadings()` to send the sensor readings, along with node IDs and timestamp, to the MQTT manager.

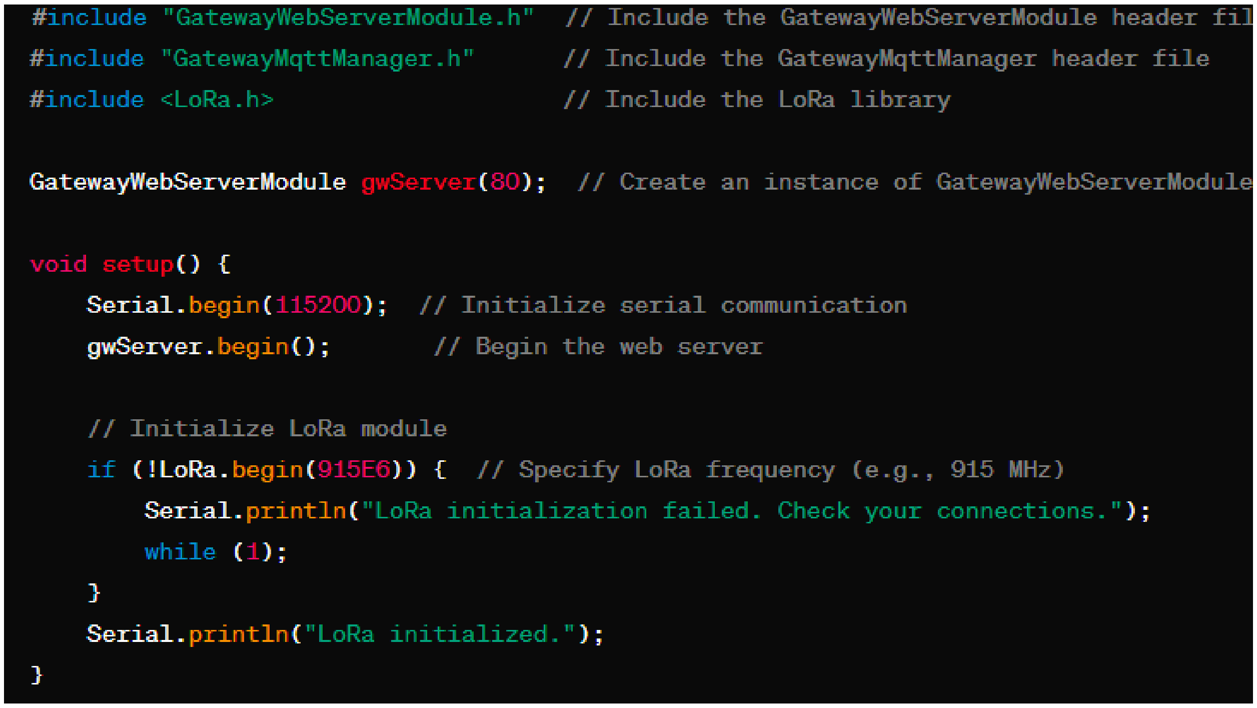
Now, to modify the functionality provided to read from LoRa sensors and pass the data to `generateAndSendSensorReadings`, you'll need to integrate LoRa communication functionality into the setup and loop functions. Here's a high-level overview of how you can do this:

Modifications:

1. Include LoRa Library: Include the necessary LoRa library to enable communication with LoRa sensors.
2. Initialize LoRa Communication: Set up LoRa communication in the `setup()` function to configure the LoRa module and join the LoRa network.
3. Read from LoRa Sensors: In the `loop()` function, implement code to receive sensor data from the LoRa sensors.

4. Pass Sensor Data to `generateAndSendSensorReadings()`: Once sensor data is received from the LoRa sensors, format it appropriately and pass it to `generateAndSendSensorReadings()` to transmit the data to the MQTT manager.

Example Pseudocode:

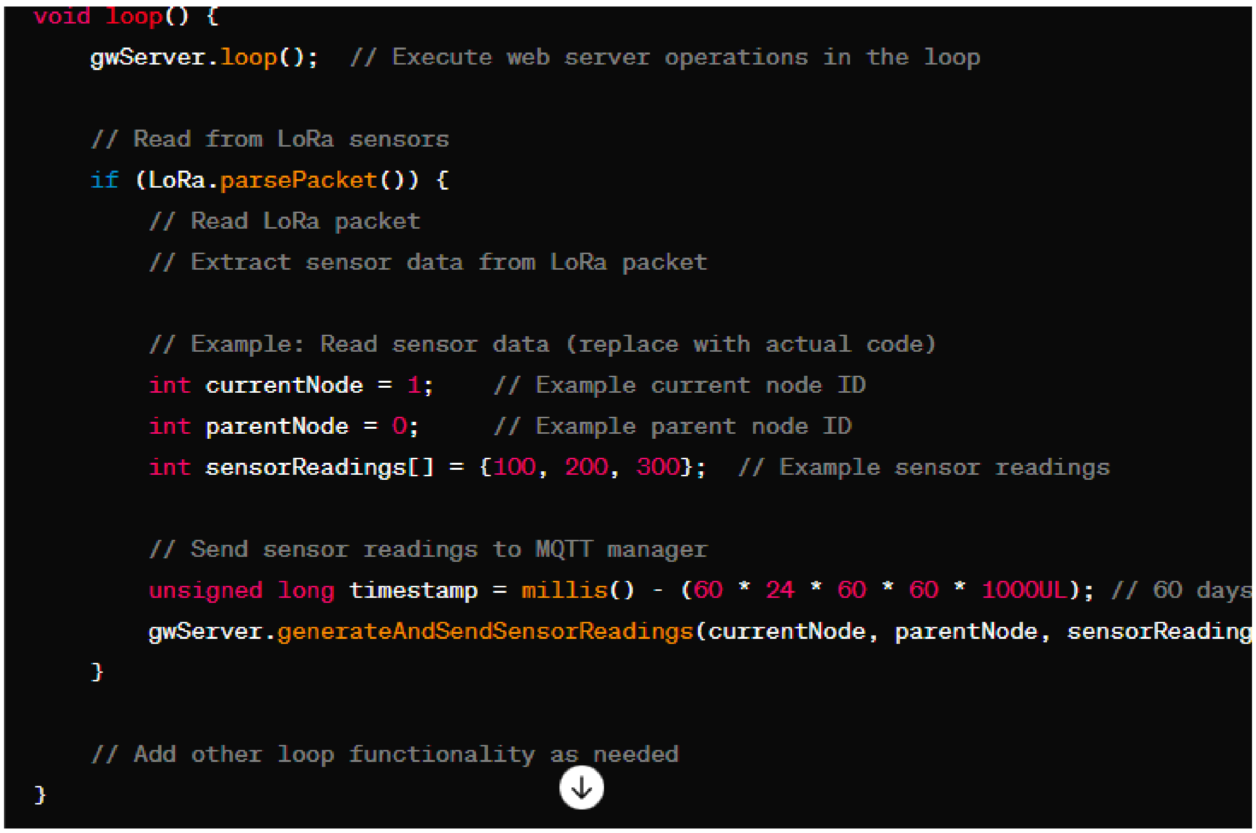


```
#include "GatewayWebServerModule.h"  // Include the GatewayWebServerModule header fil
#include "GatewayMqttManager.h"      // Include the GatewayMqttManager header file
#include <LoRa.h>                    // Include the LoRa library

GatewayWebServerModule gwServer(80);  // Create an instance of GatewayWebServerModule

void setup() {
    Serial.begin(115200);  // Initialize serial communication
    gwServer.begin();       // Begin the web server

    // Initialize LoRa module
    if (!LoRa.begin(915E6)) {  // Specify LoRa frequency (e.g., 915 MHz)
        Serial.println("LoRa initialization failed. Check your connections.");
        while (1);
    }
    Serial.println("LoRa initialized.");
}
```



```
void loop() {
    gwServer.loop();  // Execute web server operations in the loop

    // Read from LoRa sensors
    if (LoRa.parsePacket()) {
        // Read LoRa packet
        // Extract sensor data from LoRa packet

        // Example: Read sensor data (replace with actual code)
        int currentNode = 1;    // Example current node ID
        int parentNode = 0;     // Example parent node ID
        int sensorReadings[] = {100, 200, 300};  // Example sensor readings

        // Send sensor readings to MQTT manager
        unsigned long timestamp = millis() - (60 * 24 * 60 * 60 * 1000UL); // 60 days
        gwServer.generateAndSendSensorReadings(currentNode, parentNode, sensorReading
    }

    // Add other loop functionality as needed
}
```

Explanation:

- LoRa Initialization: In the `setup()` function, initialize the LoRa module with the appropriate frequency (e.g., 915 MHz).
- LoRa Sensor Reading: In the `loop()` function, use `LoRa.parsePacket()` to check for incoming LoRa packets. If a packet is available, read the packet and extract sensor data.
- Sensor Data Transmission: Format the sensor data (e.g., node IDs, sensor readings) and pass it to `generateAndSendSensorReadings()` to transmit the data to the MQTT manager.

Notes:

- Replace the example LoRa setup and sensor reading code with the actual implementation for your LoRa sensors.
- Ensure that the LoRa frequency matches the frequency used by your LoRa sensors and network.
- Test the modified sketch with your LoRa sensors to verify proper functionality.

2. Documentation for Lambda Functions:

AWS Lambda Function - cloud/lambda_functions/gateway_data_handler.py

This Python code is designed to run as an AWS Lambda function responsible for processing sensor data and storing it in DynamoDB tables. It is triggered by events, typically generated by IoT devices or other AWS services.

Functionality Overview:

1. Initialization:
    - The code initializes the AWS SDK (boto3) and connects to the DynamoDB service in the specified AWS region (ca-central-1).
2. Lambda Handler:
    - The `lambda_handler` function serves as the entry point for the Lambda function. It receives an event and context object from the Lambda runtime.
    - It extracts the system name from the event, which is assumed to be provided as part of the event data.
3. Retrieving Tables:
    - The function retrieves necessary DynamoDB tables:
        - `GatewayTopicMappingsTable`: Stores mappings between MQTT topics and gateway IDs.
        - `GatewayNodeTable`: Stores information about gateway nodes.
        - `SensorTable`: Stores sensor metadata, including sensor types and IDs.
4. Fetching Sensor Names:
    - Sensor names are dynamically retrieved from the `SensorTable` associated with the specified system name.
    - The items are sorted based on the 'ID' attribute to ensure consistency in sensor order.
5. Creating Sensor Tables:
    - Dynamic creation of DynamoDB tables is performed for each sensor type found in the `SensorTable`. These tables will store sensor readings.
6. Retrieving Gateway Information:
    - The function queries the `GatewayTopicMappingsTable` to obtain the gateway ID based on the provided topic name.
7. Timestamp Generation:

- The current UTC date and time are obtained and adjusted for the local time zone (UTC-5) to create a timestamp string.

8. Processing Sensor Readings:
    - Sensor readings are extracted from the event data using dynamic key extraction based on the sensor index (e.g., 'sensor1Reading', 'sensor2Reading').
    - Readings are converted to Decimal format and scaled accordingly (assuming a factor of 100).
9. Storing Node Information:
    - Information about the gateway node, including the gateway ID, node ID, and parent node ID, is added to the `GatewayNodeTable`.
10. Storing Sensor Readings:
    - Sensor readings are stored in their respective DynamoDB tables, with each sensor reading associated with the node ID, timestamp, and sensor value.
11. Error Handling:
    - The function includes error handling to catch and log any exceptions that may occur during the DynamoDB operations.
12. Return Value:
    - Upon successful execution, the function returns 0 to indicate completion.
13. Helper Function:
    - The `add_to_dynamodb` function simplifies the process of adding an item to a DynamoDB table. It handles the DynamoDB put operation and includes error handling logic.

This Lambda function can be integrated into an AWS IoT solution to efficiently process and store sensor data in DynamoDB, providing a scalable and reliable solution for IoT data management.

AWS Lambda Function - cloud/lambda_functions/gateway_initialization.py

This Python code is designed to run as an AWS Lambda function responsible for initializing gateway devices and managing IoT communication. It is triggered by events, typically generated when a gateway device is added to the system or when a new topic is assigned to a gateway.

Functionality Overview:

1. Initialization:
    - The code initializes necessary resources such as the AWS SDK (boto3), logging, and connections to DynamoDB, Lambda, and IoT services.
2. Lambda Handler:
    - The `lambda_handler` function serves as the entry point for the Lambda function. It receives an event and context object from the Lambda runtime.
    - It extracts necessary information from the event, including the gateway name, system name, topic, and authentication credentials.
3. Checking Credentials:
    - The function checks if required credentials (access key, secret key, topic) are provided in the event data. If any credential is missing, it publishes an unauthorized message.
4. IAM Access Check:
    - The function checks IAM access by using the provided access key and secret key to list IAM users. This validates the provided credentials.
5. Adding Gateway Mapping:
    - The function adds mappings between system names and gateway IDs to the `SystemToGatewayTable` DynamoDB table.
    - It also adds mappings between topics and gateway IDs to the `GatewayTopicMappingsTable`.
6. Creating Gateway Node Table:
    - If not already existing, the function creates a DynamoDB table named `<system_name>GatewayNodeTable` to store information about gateway nodes.
7. Fetching Sensor Types:
    - The function fetches sensor types from the event data and queries the `SensorTypes` DynamoDB table to obtain sensor mappings.
8. Creating Sensor Tables:
    - If sensor tables don't exist, the function dynamically creates them based on the sensor types.
    - Each sensor table is named `<system_name>_<sensor_name>_Table` and stores sensor readings associated with specific sensors.
9. Creating IoT Rule:
    - If the mapping is added successfully, the function creates an IoT rule to route messages to a designated Lambda function for further processing.
10. Publishing Messages:

- The function publishes confirmation or unauthorized messages to AWS IoT topics, indicating the success or failure of the gateway initialization process.

11. Error Handling:
    - The function includes error handling to catch and log exceptions, such as value/key errors or client errors, during the initialization process.
12. Helper Functions:
    - Several helper functions are included to simplify DynamoDB operations, message publishing, IoT rule creation, and permission management.

This Lambda function streamlines the process of initializing gateway devices within an IoT ecosystem, ensuring proper authentication, data routing, and storage. It integrates with other AWS services to provide a scalable and secure solution for managing IoT devices and data.

AWS Lambda Function- cloud/lambda_functions/sensor_data_api_handler

This Python code is designed to run as an AWS Lambda function responsible for handling HTTP GET requests to retrieve sensor data from DynamoDB tables. It is triggered by HTTP GET requests and returns sensor data in JSON format.

Functionality Overview:

1. Initialization:
    - The code initializes the AWS SDK (boto3) and connects to DynamoDB.
2. Lambda Handler:
    - The `lambda_handler` function serves as the entry point for the Lambda function. It receives an event and context object from the Lambda runtime.
    - It checks if the HTTP request method is GET and extracts the value of the 'systemName' query string parameter from the event.
3. Fetching Sensor Data:
    - If 'systemName' is provided, the function dynamically fetches sensor names from the `{SystemName}SensorTable` DynamoDB table.

- It then iterates over each sensor table, scans for items, and retrieves the first 100 items sorted by timestamp in descending order.
- The retrieved sensor data is stored in a dictionary named `combined_items`, where keys are sensor names and values are lists of sensor readings.

4. Handling Missing Parameters:
   - If 'systemName' is not provided in the query string, the function returns a 400 status code with an error message in the response body.
5. Returning Response:
   - Upon successful retrieval of sensor data, the function returns a 200 status code with the `combined_items` dictionary as the response body in JSON format.
   - If an error occurs during execution, the function returns a 500 status code with the error message in the response body.

This Lambda function provides a convenient way to retrieve sensor data from DynamoDB tables via HTTP GET requests, making it suitable for integration with web applications or other HTTP-based clients.

AWS Lambda Function - cloud/lambda_functions/contol.py

This Python code is designed to run as an AWS Lambda function responsible for receiving control data via HTTP POST requests, parsing the data, and publishing it to an MQTT topic. The function is intended to facilitate remote control and management of IoT devices within a system.

Functionality Overview:

1. Initialization:
   - The code imports the required libraries (`json` and `boto3`) and creates an IoT client using the `boto3.client` method.
2. Lambda Handler:
   - The `control_handler` function serves as the entry point for the Lambda function. It receives an event and context object from the Lambda runtime.
   - It prints the received event for debugging purposes and attempts to parse the JSON body from the event.
3. Parsing Data:

    - The function extracts relevant control data from the parsed JSON body, including the system ID, gateway ID, and sleep hours.
4. Publishing to MQTT Topic:
    - It constructs an MQTT topic based on the system ID and gateway ID.
    - The control message, containing system ID, gateway ID, and sleep hours, is serialized into JSON format and published to the MQTT topic using the IoT client's `publish` method.
    - If successful, it prints a success message indicating the published message.
5. Handling Errors:
    - Any exceptions that occur during data processing are caught, and an error message is printed.
    - The function prepares an error response with a status code of 500 (Internal Server Error) if an exception occurs.
6. Returning Response:
    - The function returns the appropriate response, either a success response with a status code of 200 and a success message or an error response with a status code of 500 and an error message.

This Lambda function facilitates the reception and processing of control data, enabling remote management and configuration of IoT devices within a system. It seamlessly integrates with MQTT messaging, making it suitable for controlling devices in real-time through an MQTT-based communication protocol.

3. Documentation for Dashboards:

3.1. Analytics Dashboard

Python Script:

- This is a simple polling script. The goal of this script is to connect to our AWS instance (where our raw data is stored) and the elasticsearch instance (where we display the data in illustrations).
- For every fixed time interval (the interval can be costumized), the script requests the latest data from AWS and sends it to Elasticsearch.

3.2. Control Dashboard

Server.js:

- This code creates a web server using Express, connects to DynamoDB to fetch data, processes that data and serves it as JSON in response to requests made to the '/combineddata' endpoint.

Index.html:

- This code creates a webpage where users can select a system and gateway, input sleep hours, and submit the data to the API gateway. This page dynamically updates elements based on user interactions and data fetched from the server.