HO CHI MINH CITY, UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEER

Internet of Things Application Development

Real-Time System Project Assignment

Student	Name	ID
1	Bùi Trần Hải Đăng	2352248
2	Lê Minh Hào	2310846
3	Nguyễn Đặng Minh Trường	2353262

HO CHI MINH CITY, DECEMBER 2025

Content

Table 1: Member list & workload

No.	Fullname	Student ID	Problems	% done
1	Bùi Trần Hải Đăng	2352248	- Task 1 - Task 5 - Report	100%
2	Lê Minh Hào	2310846	- Task 2 - Task 6 - Report	100%
3	Nguyễn Đặng Minh Trường	2353262	- Task 3 - Task 4 - Report	100%

Abstract

This project represents a comprehensive exploration of real-time embedded systems through the development and implementation of an RTOS-based Internet of Things (IoT) application on the ESP32-S3 (YOLO-UNO Board) microcontroller platform. The fundamental objective centers on gaining hands-on experience with real-time operating system concepts while extending an existing RTOS project framework to create a functionally distinct embedded system.

The project leverages the ESP32 S3 platform, a powerful microcontroller featuring dual-core processing capabilities, integrated Wi-Fi and Bluetooth connectivity, and extensive peripheral support, making it ideally suited for complex IoT applications. Development is conducted using the PlatformIO development environment, which provides a comprehensive framework for embedded systems programming with extensive library support and cross-platform compatibility.

The implementation encompasses five major functional domains: temperature-responsive LED control with semaphore synchronization, humidity-based NeoPixel RGB LED pattern generation, real-time environmental monitoring with LCD display integration, web-based device control through an access point interface, and TinyML model deployment for edge-based machine learning inference. Each domain requires careful orchestration of multiple concurrent tasks while maintaining system responsiveness and meeting real-time constraints

Acknowledgement

We would like to express our sincere gratitude to our course lecturer, **Dr. Lê Trọng Nhân**, for his dedicated guidance and invaluable support throughout the completion of this assignment. His clarity in teaching, thoughtful feedback, and his passion for embedded IoT programming and his many contributions to various smart-things projects have significantly advanced the school's educational mission.

1 Introduction and Objectives

Real-Time Operating Systems (RTOS) have become indispensable in modern embedded applications where deterministic timing behavior and predictable task execution are critical requirements. Unlike general-purpose operating systems (GPOS), which prioritize system throughput and fairness, RTOS platforms are specifically designed to satisfy strict timing constraints and provide guaranteed response times for time-critical tasks. This distinction is especially important in embedded IoT systems where multiple concurrent operations—such as sensor data acquisition, display updates, network communication, and TinyML inference—must execute reliably within well-defined time bounds.

This project aims to provide hands-on experience with these real-time constraints by modifying and extending the existing `YoloUNO_PlatformIO - RTOS_Project` on the ESP32-S3 platform. The objective is to redesign the original codebase and integrate new functionalities such that at least 30% of the implementation differs from the starter project. Throughout the development process, we apply RTOS concepts including multitasking, inter-task synchronization, and semaphore-based concurrency control to build a responsive and reliable embedded IoT system.

The assignment consists of six key tasks, each demonstrating essential IoT capabilities: adaptive LED behavior based on temperature, NeoPixel visualization controlled by humidity levels, real-time monitoring with LCD updates, a redesigned web server operating in Access Point (AP) mode, TinyML deployment with on-device accuracy evaluation, and cloud data publishing to the CoreIOT platform. These tasks showcase how embedded systems integrate sensing, actuation, scheduling, machine learning, and cloud connectivity under an RTOS environment.

By completing this project, our group gains practical experience in embedded software design and in constructing dependable real-time systems. The final implementation highlights the importance of deterministic task execution, synchronized resource management, and efficient IoT system integration when building modern, intelligent embedded solutions.

The primary objectives of this project are structured to provide comprehensive learning outcomes in real-time embedded systems development:

1.1 Task 1: Single LED Blink with Temperature Conditions

This task implements temperature-dependent LED blinking logic, where a standard GPIO-controlled LED exhibits frequency variations corresponding to environmental temperature conditions. The task receives dequeued sensor data and translates temperature values into dynamic LED flash rates, demonstrating practical application of semaphore-protected critical sections for serial communication.

Console output displays formatted temperature readings and calculated blink frequencies (in Hz), demonstrating that the semaphore-protected serial communication prevents output corruption even when multiple tasks attempt concurrent logging. The dynamic frequency response provides immediate visual feedback on thermal conditions.

1.2 Task 2: NeoPixel LED Control Based on Humidity

This task consumes temperature and humidity data from the sensor queue and drives an addressable NeoPixel RGB LED strip (typically 1-4 LEDs) with colors that represent humidity levels. The task transforms abstract sensor measurements into intuitive visual representations, enabling users to immediately perceive environmental conditions through color coding.

Console output provides detailed status information including current humidity percentage, status category, and color representation, demonstrating synchronized multi-task logging through semaphore-protected critical sections. The visual feedback mechanism enables quick system status assessment at a glance.

1.3 Task 3: Environmental Sensor Acquisition and Monitoring

This foundational task implements the primary sensor interface layer, responsible for acquiring temperature and humidity measurements from a DHT20 environmental sensor via the I2C protocol. Operating on a 5-second cycle, the task continuously reads sensor data and presents real-time environmental conditions on a 16x2 character LCD display through I2C communication.

The sensor task continuously enqueues SensorData structures containing temperature and humidity values to three separate message queues, providing the foundational data feed that powers the LED control, NeoPixel visualization, and machine learning inference tasks. Temperature and humidity thresholds serve as visual indicators of system state and trigger responsive behaviors in downstream tasks.

1.4 Task 4: Web Server in Access Point Mode

This task implements an embedded web server operating in ESP32 Access Point (AP) mode, providing a web-based user interface for real-time monitoring and remote control of IoT device parameters. The server presents an interactive dashboard displaying current temperature, humidity, LED states, and system status, enabling users to monitor and control the embedded system from any wireless-enabled device without requiring external infrastructure.

The web interface provides a professional user experience suitable for IoT deployment scenarios where non-technical users require intuitive system status monitoring and control capabilities. HTML rendering includes international character support (Vietnamese) and intuitive emoji-based visual indicators.

1.5 Task 5: TinyML Deployment and Accuracy Evaluation

This advanced task deploys a trained TensorFlow Lite Micro neural network model on the microcontroller to perform on-device anomaly detection based on temperature and humidity patterns. The task normalizes incoming sensor data using pre-computed mean and standard deviation values, executes the quantized ML model, and produces inference results with adaptive threshold adjustment based on observed anomaly frequency.

Console output displays raw inference results, current threshold values, and anomaly classifications, providing transparency into the ML decision-making process. The adaptive threshold mechanism demonstrates intelligent system behavior that self-calibrates to match observed data characteristics.

1.6 Task 6: Data Publishing to CoreIOT Cloud Server

This optional task demonstrates cloud connectivity through MQTT protocol integration with external IoT platforms such as CoreIOT. When enabled, the task manages connection establishment, MQTT subscription to remote procedure calls, and bidirectional telemetry data transmission to cloud-based monitoring systems.

Console output confirms connection status, subscription success, and telemetry transmission, providing visibility into cloud synchronization operations. This task demonstrates enterprise-grade IoT patterns suitable for production deployments requiring centralized monitoring and remote management capabilities.

2 Implementation and Experimental results

2.1 Overview Project

The system architecture follows a producer-consumer pattern where a central sensor acquisition task (Task `temp_humi_monitor`) acts as the primary data producer, broadcasting environmental readings to three specialized consumer tasks (Tasks `led_blinky`, `neo_pixel`, `tiny_ml`, mainserver, coreiot) through dedicated FIFO message queues.

2.2 Global Resource Initialization

The file `global.cpp` declares and initializes all FreeRTOS synchronization primitives before any tasks are created:

```
SemaphoreHandle_t xBinarySemaphoreInternet = xSemaphoreCreateBinary();
SemaphoreHandle_t xSemaphoreMutex = xSemaphoreCreateMutex();
QueueHandle_t xQueueForLedBlink = xQueueCreate(5, sizeof(SensorData));
QueueHandle_t xQueueForNeoPixel = xQueueCreate(5, sizeof(SensorData));
QueueHandle_t xQueueForTinyML = xQueueCreate(5, sizeof(SensorData));
QueueHandle_t xQueueForCoreIOT   = xQueueCreate(1, sizeof(SensorData));
QueueHandle_t xQueueForMainServer   = xQueueCreate(5, sizeof(MLResult));
```

Initialization Sequence:

- Binary Semaphore (`xBinarySemaphoreInternet`)
- **Mutex (`xSemaphoreMutex`)**: Created to protect Serial console output. This mutex ensures that when multiple tasks attempt to log information simultaneously, output statements remain atomic and readable.
- **Message Queues**: Three queues are created with identical capacity but serve different consumer tasks:
  - `xQueueForLedBlink`: Delivers sensor data to LED control task
  - `xQueueForNeoPixel`: Delivers sensor data to NeoPixel visualization task
  - `xQueueForTinyML`: Delivers sensor data to machine learning inference task
  - `xQueueForCoreIOT`: Delivers sensor data to the CoreIOT publish task
  - `xQueueForMainServer`: Delivers the results of the machine learning task to the web server task

2.3 Detailed Task Implementations

2.3.1 Task LED_BLINKY (`led_blinky`)

The LED blinky task functions as a consumer task within the producer-consumer architecture, contrasting fundamentally with the producer-oriented temperature and humidity monitoring

task. Provide immediate visual feedback of temperature conditions through dynamic LED blink frequency modulation. The task implements a three-tier threshold system where LED blink rate increases proportionally with temperature elevation.

Key Implementation Highlights:

- **Blocking Queue Reception:** The core control flow implements `xQueueReceive(xQueueForLedBlink, &data_receive, portMAX_DELAY)` with a timeout parameter set to `portMAX_DELAY`. This configuration creates a blocking reception mechanism where the task surrenders CPU time to the RTOS scheduler when no data is available.

  The semantics are fundamentally different from the producer task's zero-timeout non-blocking send: instead of immediately returning failure, the receiving task transitions to the BLOCKED state and remains there until either new data arrives or the system encounters a critical timeout.

  The blocking queue mechanism provides implicit task synchronization without requiring explicit binary semaphores or condition variables. The queue state inherently synchronizes producer and consumer execution phases: the producer remains blocked (if a queue is full) until consumers process data, and consumers remain blocked until producers supply data. This natural back-pressure regulation prevents queue overflow and ensures neither producer nor consumers race ahead uncontrollably.

- **Frequency Calculation and Mapping:**
  - Temperature < 30°C: 500ms ON + 500ms OFF = 1000ms period $\rightarrow$ 1 Hz
  - Temperature 30–35°C: 250ms ON + 250ms OFF = 500ms period $\rightarrow$ 2 Hz
  - Temperature > 35°C: 125ms ON + 125ms OFF = 250ms period $\rightarrow$ 4 Hz

- **Mutex-Protected Critical Section:** The critical section encompasses all four Serial.print() statements:

```
Serial.print("[LED] Temperature: ");
Serial.print(temperature, 2);
Serial.print("°C | Frequency: ");
Serial.print(1.0 / (((float)xfrequency * 2) / 1000.0), 1);
Serial.println(" HZ");
```

  This entire sequence executes atomically—no other task can interleave serial output during this section because no other task can acquire the mutex until the LED task releases it via xSemaphoreGive(xSemaphoreMutex). The Give operation returns the mutex to the available pool, potentially unblocking other tasks waiting for it.

  Example output:

```
[LED] Temperature: 28.50°C | Frequency: 1.0 HZ
[LED] Temperature: 32.20°C | Frequency: 2.0 HZ
[LED] Temperature: 36.80°C | Frequency: 4.0 HZ
```

2.3.2 Task NeoPixel LED control (neo_blinky)

The NeoPixel blinky task serves as the second consumer task within the multi-consumer architecture, operating in parallel with the LED blinky task while processing identical sensor data for a distinct visualization purpose. While the LED blinky task maps temperature to blink frequency, the NeoPixel task maps humidity to color states, creating a complementary visual feedback system. This separation of concerns—temperature visualization via one indicator and humidity visualization via another—exemplifies the single responsibility principle applied to real-time embedded systems, where each task performs one well-defined function.

Key Implementation Highlights:

- **Queue Reception:** Like the LED blinky task, the NeoPixel task implements an event-driven consumer pattern using blocking queue reception with `xQueueReceive(xQueueForNeoPixel, &data_receive, portMAX_DELAY)`. The timeout parameter `portMAX_DELAY` creates indefinite blocking semantics: the task transitions to the BLOCKED state when no data is available in its queue and remains blocked until the producer task enqueues new sensor data.
- **Humidity-Based Four-Tier Classification System:**
  - **Error State Detection (White Color)**: The first conditional branch handles anomalous humidity readings through two checks: humidity < 0 and isnan(humidity). Negative humidity values are physically impossible (relative humidity ranges from 0% to 100%), so any negative reading indicates sensor malfunction, calibration error, or data corruption.
  - **Low Humidity State (Red Color, 0–60%:)** Humidity readings in the range [0, 60) are classified as LOW, displayed with red illumination (RGB: 255, 0, 0). From an environmental perspective, relative humidity below 60% represents increasingly dry conditions.

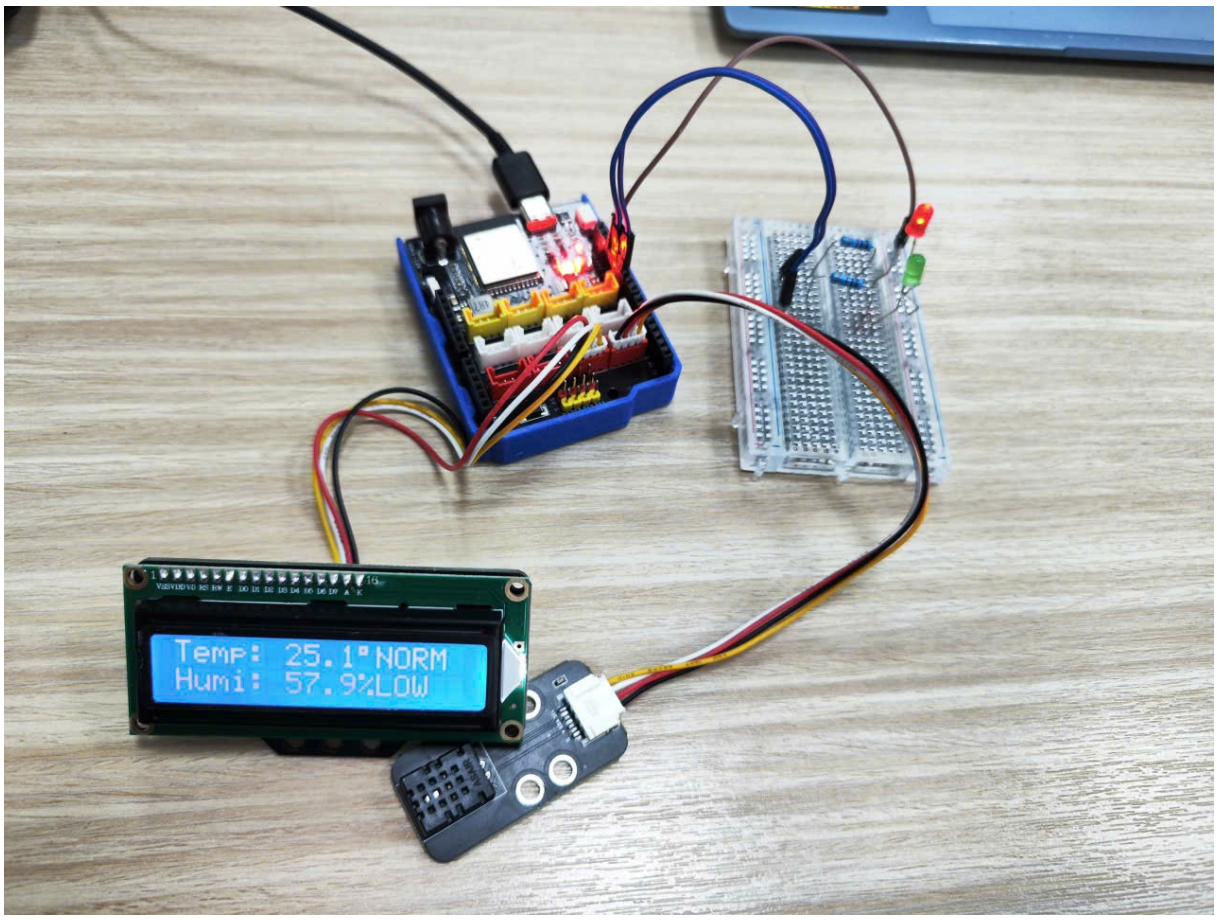

Figure 1: The LED turns red when the humidity is 57.9%.

– **Normal Humidity State (Green Color, 60–80%):** Humidity readings in the range [60, 80) are classified as NORMAL, displayed with green illumination (RGB: 0, 255, 0). This range represents optimal environmental conditions for most applications: comfortable for human occupancy, appropriate for electronics (preventing static discharge without risking condensation), and suitable for material storage.

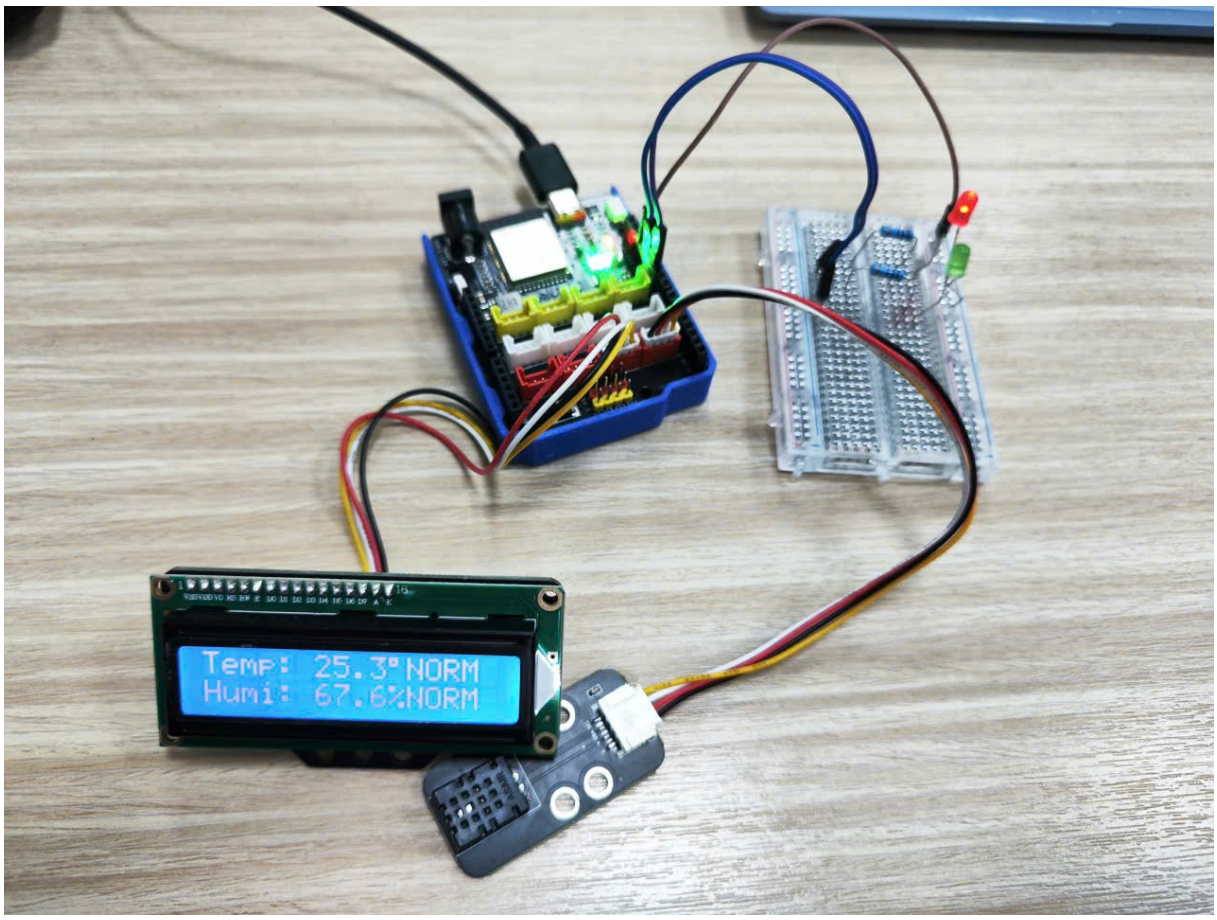

Figure 2: The LED turns green when the humidity is 67.7%.

– **Wet/High Humidity State (Blue Color, $>$ 80%):** Humidity readings at or above 80% are classified as WET, displayed with blue illumination (RGB: 0, 0, 255). High humidity poses risks including condensation on electronic components, mold growth, material degradation, and human discomfort.

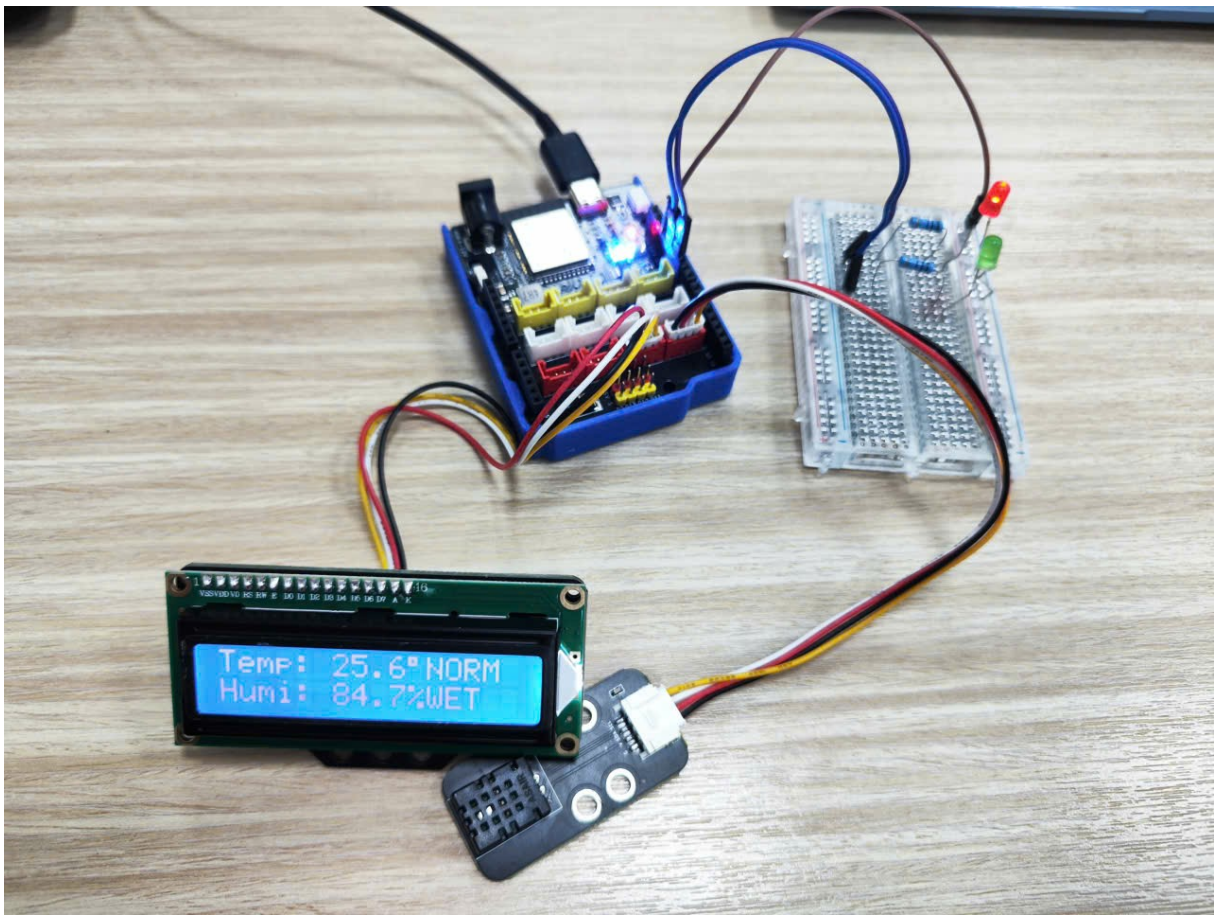

Figure 3: The LED turns blue when the humidity is 84.7%.

- **Mutex-Protected Serial Logging with Atomic String Construction:** The task implements mutex-protected serial output following the same pattern as the LED blinky task, but with an enhanced atomicity strategy through pre-constructed string concatenation.

```
String logMessage = "[NeoPixel] Update - Humidity: " + String(humidity, 1) +
                    "% | Status: " + status +
                    " | Color: " + colorName;
Serial.println(logMessage);
```

2.3.3 Task: Temperature-Humidity Monitor (`temp_humi_monitor`)

This task serves as the system's primary data producer, responsible for periodic acquisition of environmental sensor readings and distribution to all consumer tasks. Additionally, it provides real-time visual feedback through an LCD display module.

This multi-queue distribution pattern exemplifies a publish-subscribe architecture where a single data source disseminates information to multiple subscribers with varying processing requirements

Key Implementation Highlights:

- **RTOS Queue Communication Architecture**

  In this project, multiple concurrent tasks exchange sensor information through FreeRTOS message queues. The queues operate under a **First-In First-Out (FIFO)** discipline, ensuring deterministic delivery order across all consumer tasks. Figure 4 illustrates the overall communication flow among the Sensor Task (producer) and four independent consumer tasks: LED Control, NeoPixel Display, TinyML Inference, and CoreIoT Communication.

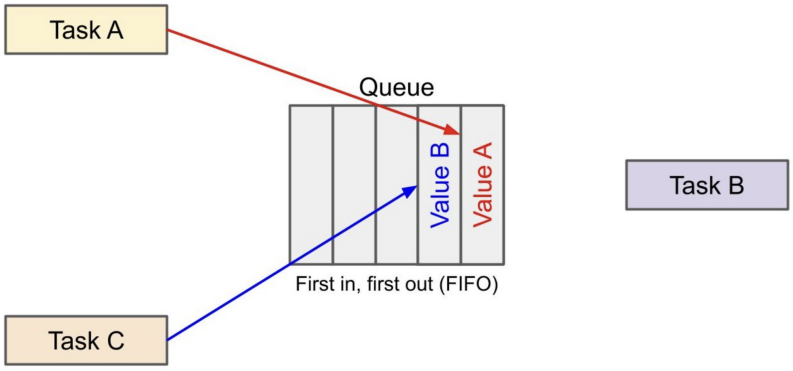

Figure 4: RTOS queue architecture with FIFO ordering between producer and multiple consumers.

Producer Task: Sensor Sampling and Distribution

The sensor task periodically reads temperature and humidity from the DHT20 module. If the reading is valid, a `SensorData` struct is broadcast to all consumer tasks via four distinct queues:

```
1   else {
2       SensorData sensordata;
3       sensordata.temperature = temperature;
4       sensordata.humidity = humidity;
5
6       xQueueSend(xQueueForLedBlink, &sensordata, 0);
7       xQueueSend(xQueueForNeoPixel, &sensordata, 0);
8       xQueueSend(xQueueForTinyML, &sensordata, 0);
9       xQueueSend(xQueueForCoreIOT, &sensordata, 0);
10  }
```

Each send operation uses a **non-blocking timeout parameter** (0 ticks), implementing a *fail-fast* mechanism that preserves real-time deadlines.

Non-Blocking Queue Send Strategy

The critical design decision concerns the third argument of `xQueueSend()`, which specifies the maximum wait time if the queue is full.

- **timeout = 0 ticks (Non-blocking)** Operation returns immediately if the queue is full.
  - Avoids stalling the sensor task
  - Guarantees periodic sampling (5s cycle)
  - Ensures system reliability even if a consumer task fails
- **timeout = portMAX_DELAY (Blocking indefinitely)** Would cause fatal issues in our architecture:
  - Sensor task could block forever if a consumer stops receiving
  - Violates soft real-time constraint
  - System becomes unresponsive in worst-case scenarios

This choice is essential for maintaining deterministic timing behavior in a multi-task IoT system.

FIFO Queue Algorithm.

FreeRTOS queues maintain strict FIFO order. Internally, the queue is modeled as a circular buffer with two moving indices: `head` (read) and `tail` (write). Algorithm 2.3.3 summarizes the enqueue operation.

[H] FIFO Enqueue Operation in FreeRTOS [1] Queue Q, Data item d Q is full return **fail** $Q[tail] \leftarrow d$ $tail \leftarrow (tail + 1)$ mod $Q.capacity$ return **success**

The dequeue process follows a symmetrical structure:

[H] FIFO Dequeue Operation in FreeRTOS [1] Queue Q Q is empty block or return fail (depending on timeout) $d \leftarrow Q[head]$ $head \leftarrow (head + 1)$ mod $Q.capacity$ return d

These algorithms ensure that the first data item inserted is always the first item extracted, preserving temporal ordering across all tasks.

Multi-Queue Broadcast Algorithm

Since one sensor reading must be delivered to four tasks, the system uses **parallel non-blocking broadcasts**. The overall logic is described in Algorithm 2.3.3.

[H] Non-Blocking Broadcast to Multiple Queues [1] Broadcast$data$, $queueList$ $q \in queueList$ result $\leftarrow$ `xQueueSend(q, data, 0)` result == fail log "Queue full: data dropped"

This algorithm guarantees:

- Sensor task never blocks
- Each consumer receives the newest available data
- System remains responsive even under high load

Real-Time Constraints and System Reliability

Using non-blocking queue sends ensures that the Sensor Task always completes its cycle within the required deadline. Even if:

- An LED task freezes,
- NeoPixel animation runs too slowly,
- TinyML inference takes longer due to model size,
- Network communication becomes unstable,

the system continues to operate and sample new data without interruption.

This design aligns with RTOS best practices for IoT workloads:

- isolate timing-critical tasks,
- prevent dependency-induced blocking,
- guarantee periodic sensor updates.

- **Sensor Data Acquisition:** The sensor reading sequence follows a multi-stage pipeline. The DHT20 sensor communication is initiated with `dht20.read()`, which initiates the I^2C protocol sequence to retrieve environmental measurements from the sensor's internal registers. The subsequent extraction of temperature and humidity values through `dht20.getTemperature()` and `dht20.getHumidity()` retrieves the processed float values representing measurements in Celsius and percentage relative humidity respectively.

- **Error Detection and Recovery Protocol:** The implementation incorporates a critical validation layer using the `isnan()` (is not a number) function to detect sensor communication failures or data corruption. This function examines the IEEE 754 floating-point representation to determine whether the retrieved value is a valid number or a NaN sentinel value. In embedded systems contexts, sensors often return NaN when I^2C communication fails, checksum validation errors occur, or the sensor enters an error state. The conditional check if `(isnan(temperature) || isnan(humidity))` detects either condition and implements graceful degradation by logging the failure and displaying an error message on the LCD without attempting to distribute corrupted data to consumer queues.

- **Sensor Data Structure and Queue Distribution:** The SensorData structure encapsulates both temperature and humidity measurements into a unified packet:

```
1       SensorData sensordata;
2       sensordata.temperature = temperature;
3       sensordata.humidity = humidity;
```

  The identical packet is queued to all three consumer queues simultaneously, enabling parallel processing of the same data by independent tasks.

- **Temperature Threshold Classification Logic:**

- NORM: $< 30\%$ (typical room temperature)
- WARN: 30–35°C (elevated temperature)
- CRIT: > 35°C (concerning temperature)

- **Humidity Threshold Classification Logic:**
  - ERROR: $< 0\%$
  - LOW: 0–60%
  - NORM: 60–80%
  - WET: $> 80\%$

- **LCD Display Module Integration and Real-Time Feedback:** The LCD display operates as a real-time human-machine interface providing immediate visual feedback of system state. The 16×2 character LCD is initialized at I^2C address 0x33 with 16 columns and 2 rows. The display output follows a structured layout strategy:
  - The first row (LCD row 0) presents compact sensor readings: "Temp: XX.X°C", where the degree symbol is rendered using ASCII code 223 (char(223)). This is displayed alongside the temperature classification state (NORM, WARN, or CRIT) positioned at column 11, creating a right-aligned status indicator.
  - The second row (LCD row 1) displays "Humi: XX.X%" alongside the humidity classification state (ERROR, LOW, NORM, or WET), positioned at column 11 on the same row.

2.3.4 Task: Web Server in Access Point Mode (`mainserver`)

Introduction

In this task, we redesigned the ESP32 web server interface to create a more intuitive, user-friendly, and visually appealing dashboard while operating in Access Point (AP) mode. The redesigned interface enables direct control over two devices (LED1 and LED2), each equipped with clear ON/OFF action buttons. Additionally, the web server provides real-time temperature and humidity readings as well as the machine-learning inference results described in Section 2.3.4.

This task also demonstrates the mechanism for switching the ESP32 between Access Point mode and Station mode. This flexibility allows the system to operate either independently (AP mode) or as part of an existing network infrastructure (Station mode), depending on the application's requirements.

Theory

Access Point (AP) Mode: In this mode, the ESP32 creates its own WiFi network that other devices can connect to. This is useful for direct communication without relying on an external

router. When you set your ESP32 board as an access point, you can be connected using any device with Wi-Fi capabilities without connecting to your router. When you set the ESP32 as an access point, you create its own Wi-Fi network, and nearby Wi-Fi devices (stations) can connect to it, like your smartphone or computer.

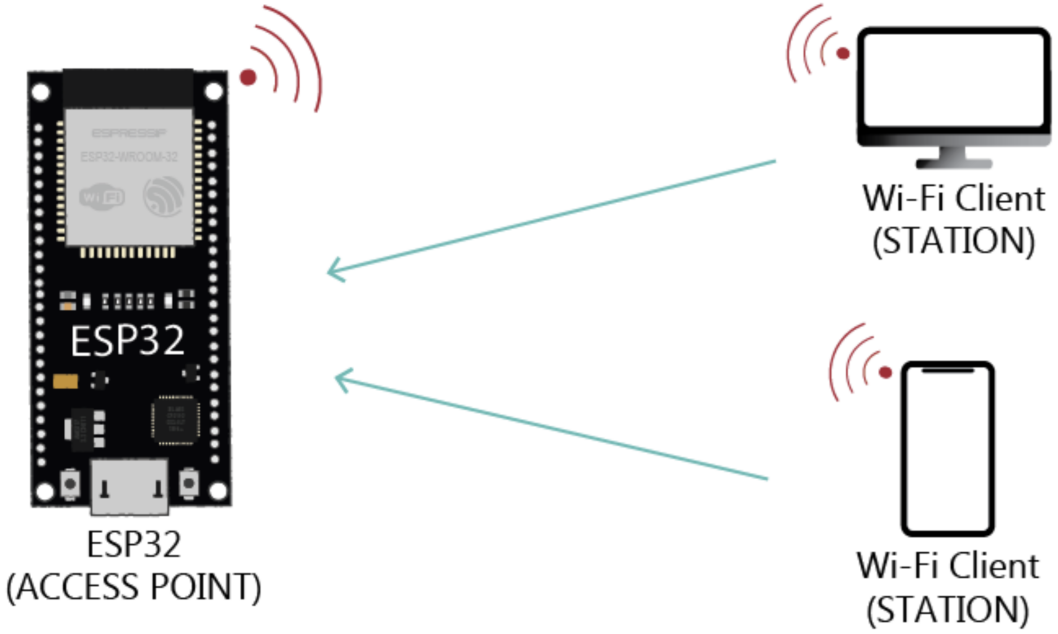

Figure 5: Access Point Mode

Station (STA) Mode: In this mode, the ESP32 connects to an existing WiFi network as a client device. This allows the ESP32 to access the internet or communicate with other devices on the same network.

When the ESP32 is set as a Wi-Fi station, it can connect to other networks (like your router). In this scenario, the router assigns a unique IP address to your ESP board. You can communicate with the ESP using other devices (stations) that are also connected to the same network by referring to the ESP unique IP address.

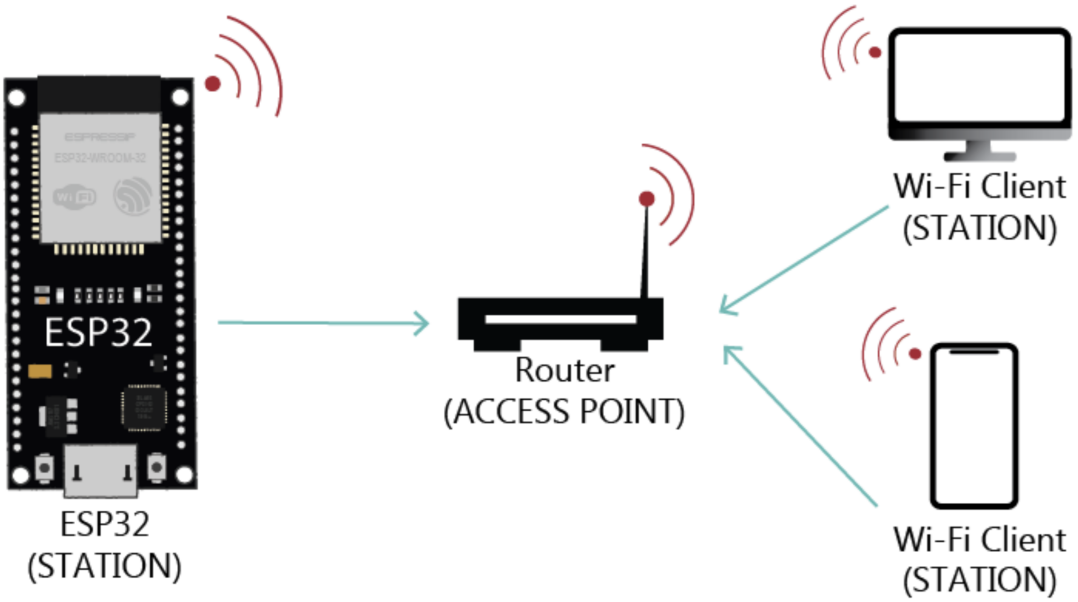

Figure 6: Station Mode

Implementation

Connecting to the ESP32 Access Point

After uploading the program to the ESP32, the board automatically switches to **Access Point (AP) mode** and creates its own Wi-Fi network. In this mode, the ESP32 functions as a wireless hotspot, allowing laptops or smartphones to connect directly without requiring any external router.

In our implementation, the AP network is configured with the following parameters:

- **SSID:** `ESPREE`
- **Password:** `123456789`

Once the ESP32 is powered on, this Wi-Fi network becomes visible in the list of available wireless connections on nearby devices. Figure **??** shows an example captured from a laptop, where the network `ESPREE!!!` appears in the list, confirming that the ESP32 AP has been successfully activated and is ready for connection.

Figure 7: Web after redesign

After selecting the ESP32 Access Point and entering the correct password, the device establishes a secure connection with the ESP32. At this stage, although the network is labeled as "No internet, secured", the connection is fully functional for local communication. This is expected behavior because the ESP32 operates as a standalone Access Point that provides a local web server rather than internet access.

Once the connection is successful, the user can access the ESP32's web interface through the assigned local IP address to control devices (LED1, LED2) and view real-time sensor data. Figure 8 shows the Wi-Fi status display on the laptop after the connection has been established.

Figure 8: Laptop successfully connected to the ESP32 Access Point.

Sensor Data Output and Web Dashboard Access

After the laptop successfully connects to the ESP32 Access Point, the system begins transmitting real-time sensor readings from the DHT20 module to the Serial Monitor. These values include temperature, humidity, anomaly detection results, and the LED status updates generated by the machine-learning model. Figure 9 illustrates an example of the continuous data stream displayed on the Serial Monitor during operation.

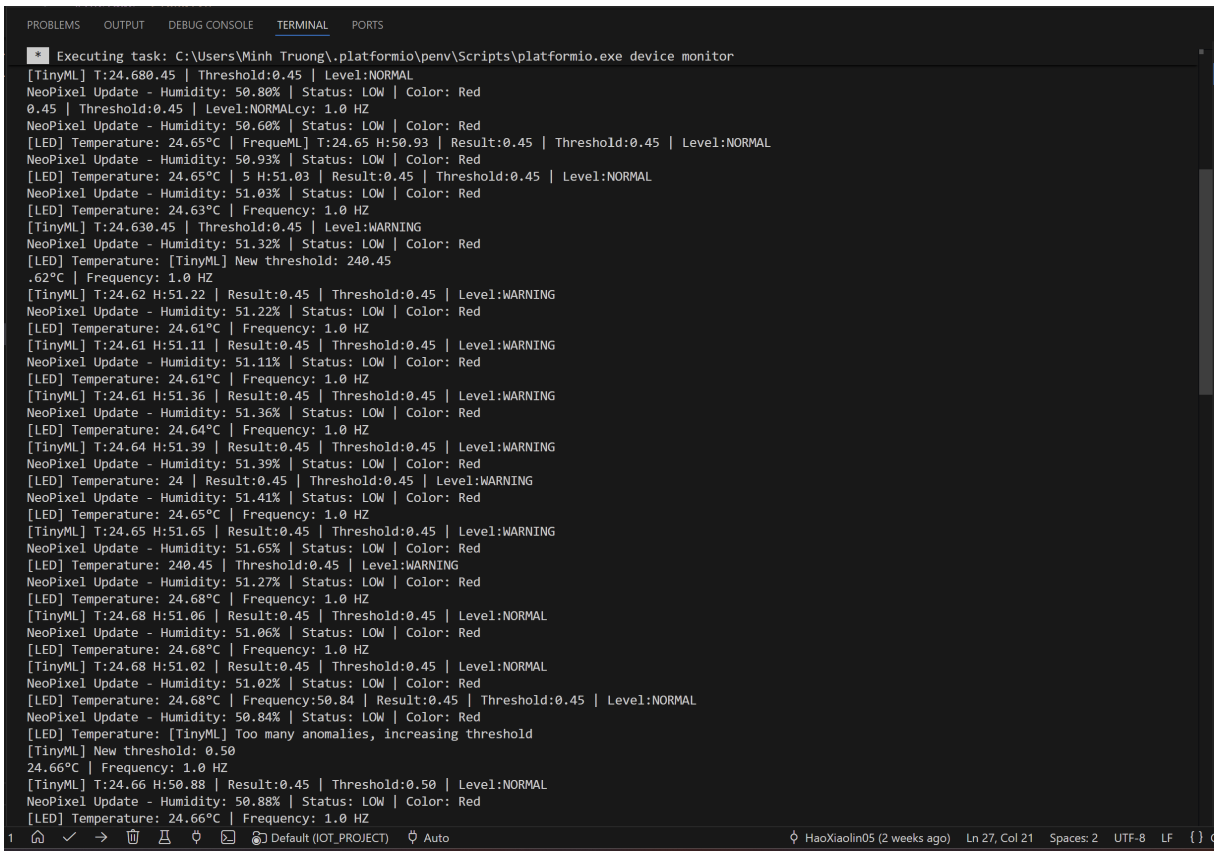

Figure 9: Real-time sensor readings and system status printed on the Serial Monitor.

Once the device is connected, the user can access the ESP32's built-in web server by navigating to the default AP IP address:

http://192.168.4.1

This web interface provides a graphical dashboard for monitoring temperature, humidity, AI predictions, and anomaly rate, as well as controlling LED1 and LED2 directly through interactive buttons. The redesigned dashboard delivers a more intuitive and visually enhanced user experience. Figure 10 shows the final layout of the updated interface.

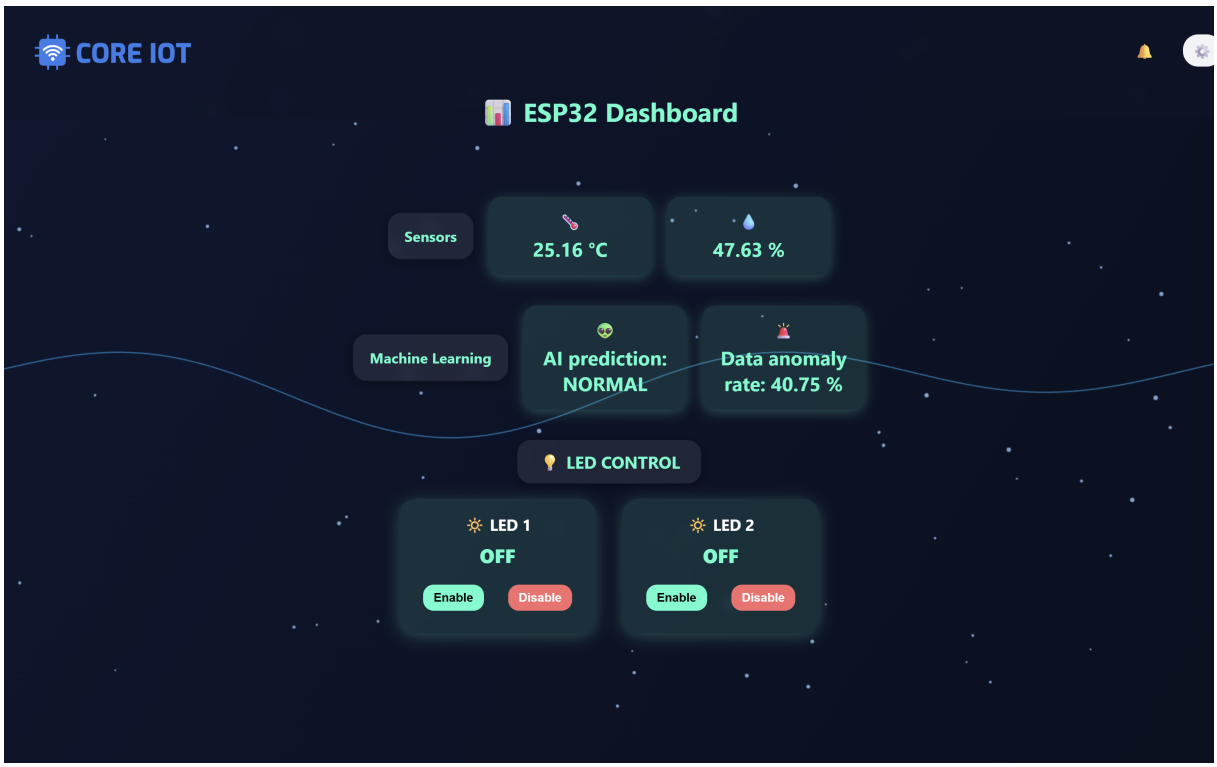

Figure 10: Redesigned ESP32 Dashboard displaying sensor data, AI prediction, anomaly rate, and LED controls.

Switching from Access Point Mode to Station Mode

After completing the interaction in Access Point (AP) mode, the ESP32 web server allows the user to switch to **Station (STA) mode** to connect the device to an external Wi-Fi network. To begin this process, the user clicks the *Settings* button located at the top-right corner of the dashboard interface. Once selected, the system automatically redirects to the Wi-Fi configuration page, where the user can enter the SSID and password of the target network.

Figure 11 shows the redesigned Wi-Fi settings interface. In this example, the user enters the credentials for the network named `gelato gelato`.

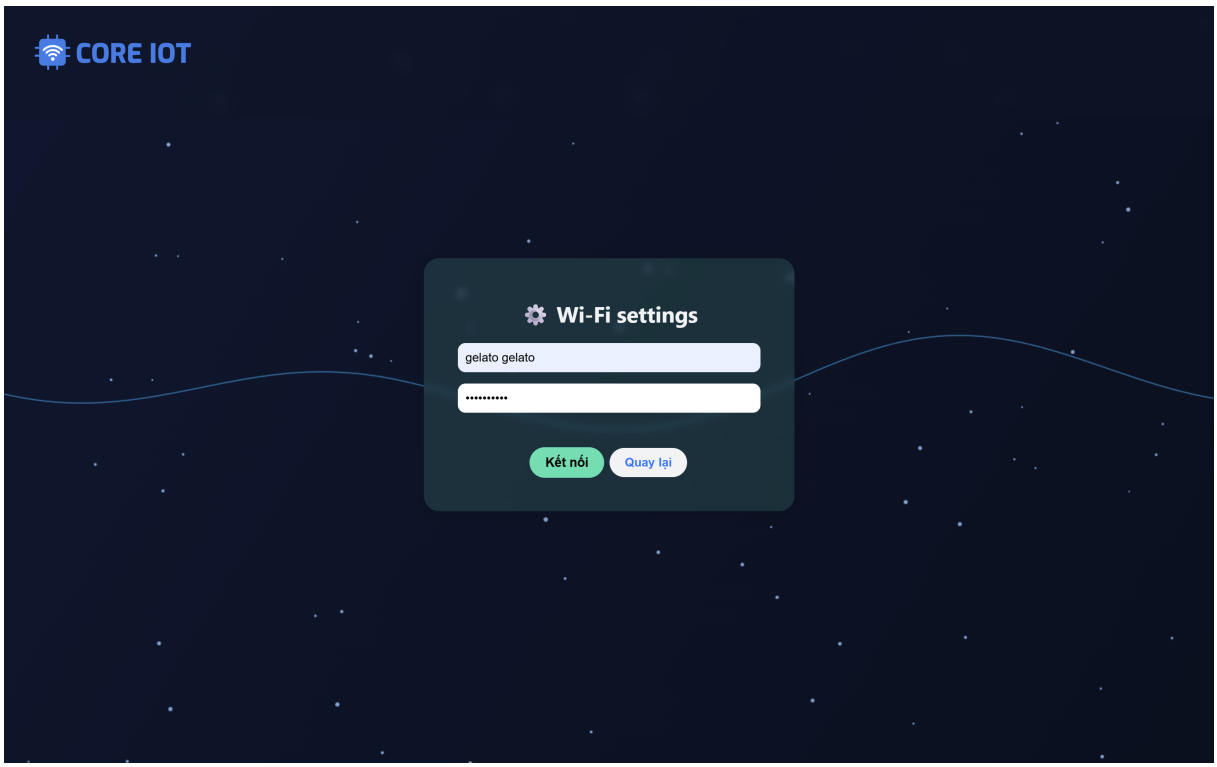

Figure 11: Redesigned Wi-Fi settings page for configuring Station mode.

After entering the correct credentials, the ESP32 attempts to connect to the specified Wi-Fi network. Once the connection is successfully established, the system triggers an automatic redirection feature.

This auto-redirect mechanism is commonly found in modern commercial web systems and enhances user experience by eliminating the need to manually re-enter the device's IP address after switching networks. As soon as the ESP32 connects to the external Wi-Fi network, the browser automatically opens the home page of the cloud platform—**CoreIOT** in our case.

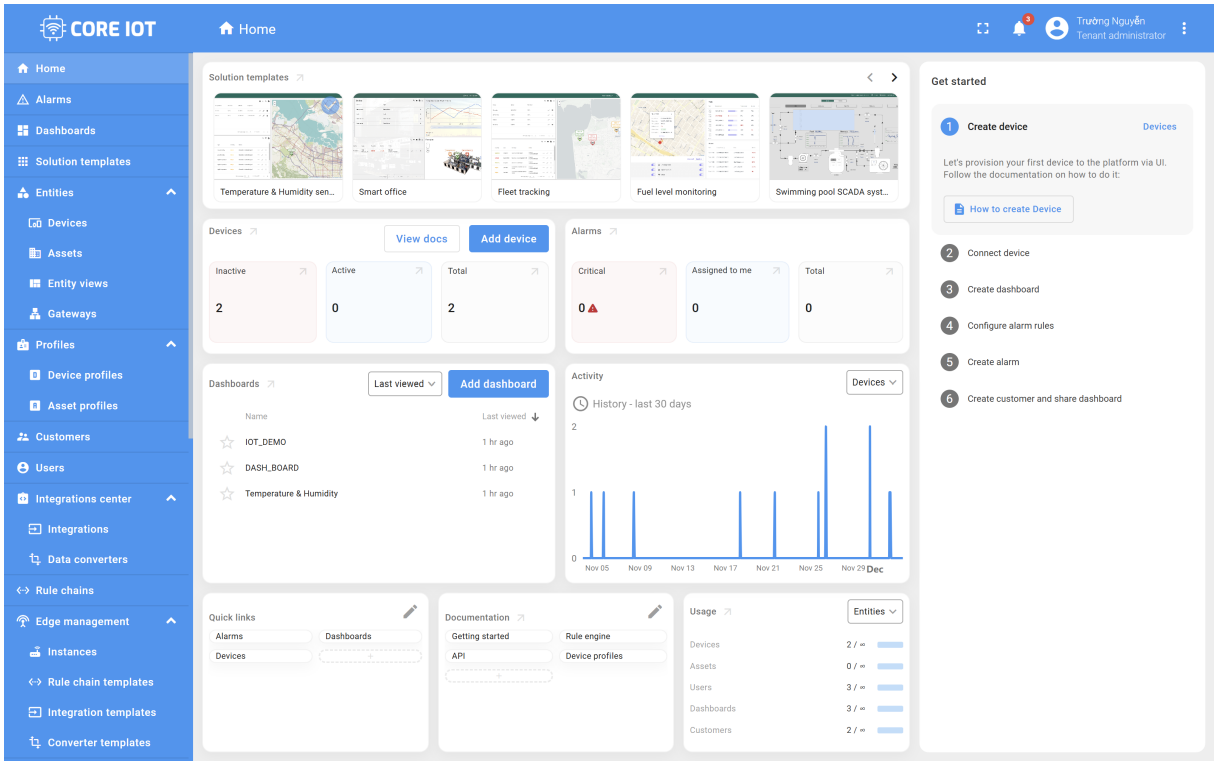

Figure 12: CoreIOT home page

To the backend of this web, there will be some **Key Implementation Highlights:**

- **Taking data:** In Task 5, we define a global MLResult struct in global.h to store the output of the Machine Learning task, including temperature, humidity, the inference score, and anomaly information. In mainserver.cpp, we use xQueueReceive(xQueueForMainServer, ml_result, portMAX_DELAY) to receive this struct from Task 2.3.4, then copy its fields into the local variables temperature, humidity, percent (converted from inference_result * 100), and result (the anomaly_type string), which are later used for displaying data and sending it to CoreIoT.

  This is the code in global.h

```
typedef struct
{
    float temperature;
    float humidity;
    float inference_result;
    bool anomaly_detected;
    String anomaly_type;
    /* data */
} MLResult;

```

  This is the code in mainserver.cpp

```
float temperature = 0;
```

```
  float humidity = 0;
  float percent = 0;
  String result = "";

  MLResult ml_result;

  if (xQueueReceive(xQueueForMainServer, &ml_result, portMAX_DELAY) == pdTRUE) {
    // Process the received sensor data
    // For example, you can print it to the Serial Monitor
    temperature = ml_result.temperature;
    humidity = ml_result.humidity;
    percent = ml_result.inference_result * 100;
    result = ml_result.anomaly_type;
  }

```

- **Controlling Leds:** The handleToggle() function is responsible for controlling LED1 and LED2 through HTTP requests received from the web server. When the client sends a request containing the led parameter, the function determines which LED to toggle.

```
void handleToggle()
{
  int led = serverAP.arg("led").toInt();
  if (led == 1)
  {
    led1_state = !led1_state;
    digitalWrite(47, led1_state);
    Serial.println("State led1: ");
    Serial.println(led1_state ? "ON" : "OFF");
  }
  else if (led == 2)
  {
    led2_state = !led2_state;
    digitalWrite(38, led2_state);
    Serial.println("State led2: ");
    Serial.println(led2_state ? "ON" : "OFF");
  }
  serverAP.send(200, "application/json",
              "{\"led1\":\"" + String(led1_state ? "ON" : "OFF") +
                  "\",\"led2\":\"" + String(led2_state ? "ON" : "OFF") + "\"}");
}
```

- **Updating data using setInterval() function and setupServer() function**

  Beside that, we will using setInterval(), the browser sends a request to the /sensors endpoint every 3 seconds. The returned JSON object contains the latest sensor readings, including temperature, humidity, inference percentage, and anomaly result.

```
    setInterval(()=>{
          fetch('/sensors')
            .then(res=>res.json())
            .then(d=>{
              document.getElementById('temp').innerText=d.temp;
              document.getElementById('hum').innerText=d.hum;
              document.getElementById('percent').innerText=d.percent;
              document.getElementById('result').innerText=d.result;
            });
        },3000);
```

The setupServer() function defines all HTTP routes handled by the ESP32 web server. And we will use it to map to a specific handler function

```
  void setupServer()
  {
    serverAP.on("/", HTTP_GET, handleRoot);
    serverAP.on("/toggle", HTTP_GET, handleToggle);
    serverAP.on("/sensors", HTTP_GET, handleSensors);
    serverAP.on("/settings", HTTP_GET, handleSettings);
    serverAP.on("/connect", HTTP_GET, handleConnect);
    serverAP.begin();
  }

```

- **Settings Page and Mode Transition:** Eventually, a **Settings** page is implemented to allow users to reconfigure the network mode of the ESP32. Through this interface, the user can manually input the **Wi-Fi SSID** and **password** of any available network directly from their phone or laptop.

  This configuration enables the ESP32 to switch seamlessly between two operating modes:

  - **Access Point (AP) Mode**: The ESP32 creates its own Wi-Fi network (e.g., `ESPREE!!`), allowing users to connect directly to the device and control all functionalities without requiring an external router.
  - **Station (STA) Mode**: The ESP32 disconnects from AP mode and instead connects to the user-specified Wi-Fi network using the credentials provided on the Settings page. This allows the device to operate as a normal client within an existing home or office network.

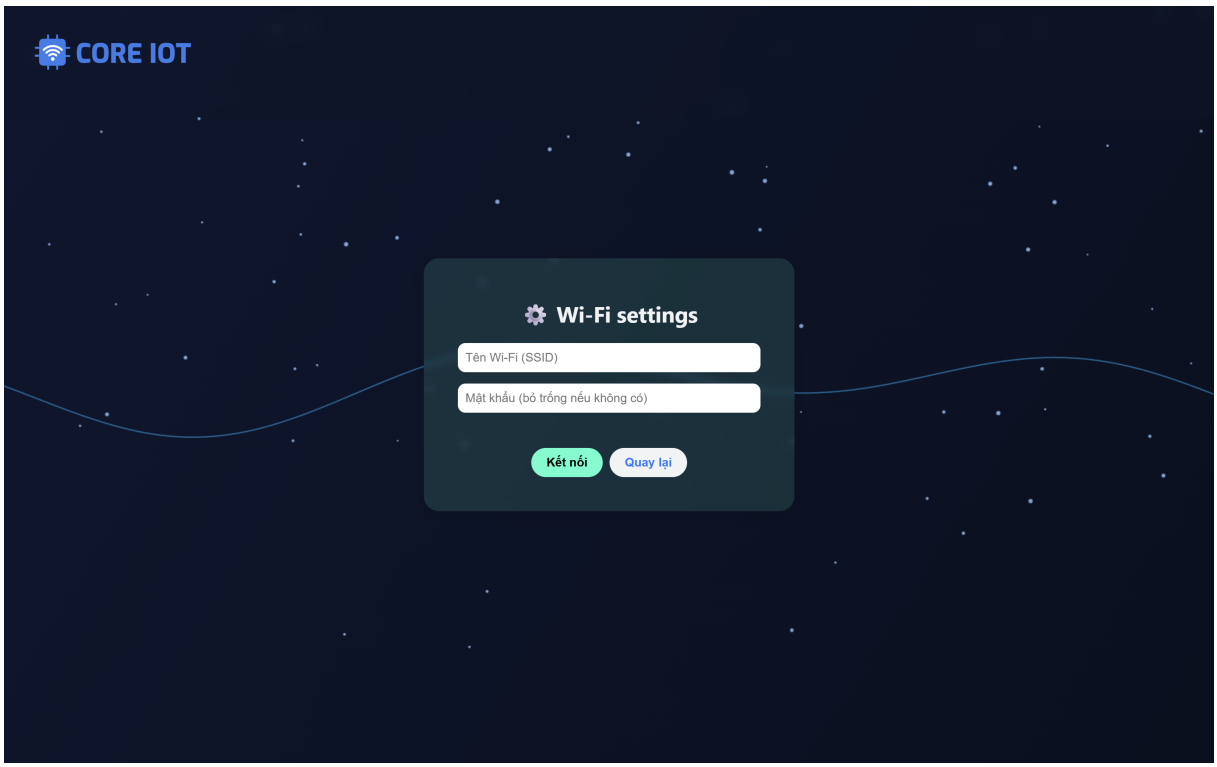

Figure 13: SetiingPage

- **Controlling Leds:** The handleToggle() function is responsible for controlling LED1 and LED2 through HTTP requests received from the web server. When the client sends a request containing the led parameter, the function determines which LED to toggle.

```
1       <h1> Wi-Fi settings</h1>
2         <form id="wifiForm">
3           <input name="ssid" id="ssid" type="text" placeholder="Tên Wi-Fi (SSID)" required><br>
4           <input name="password" id="pass" type="password" placeholder="Mật khẩu (bỏ trống nếu khô
5           <button type="submit" onclick="window.location='https://app.coreiot.io'">Kết nối</button
6           <button type="button" id="back" onclick="window.location='/'">Quay lại</button>
7         </form>
```

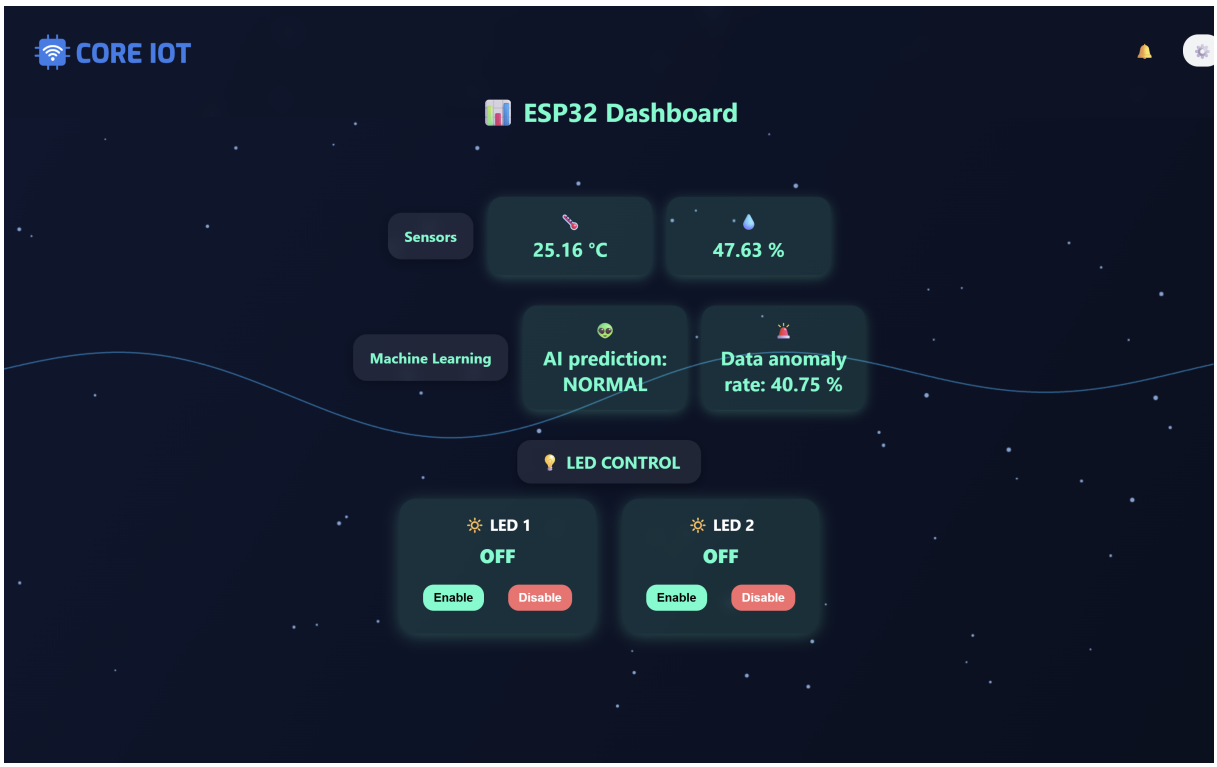

Figure 14: Web after redesign

2.3.5 Task Tiny ML (`tiny_ml`)

The system comprises two distinct phases: an offline training and model conversion pipeline implemented in Python, and an embedded inference engine deployed on an ESP32 microcontroller using TensorFlow Lite for Microcontrollers (TFLite Micro).

1. **Model Training (Python - `TFL_For_MCU.py`)**

   a. **Data Preparation:**

```python
# Load dataset from Kaggle
data = pd.read_csv("synthetic_iot_dataset.csv")

# Filter to 100 devices for manageable dataset size
data_100 = data[data['Device_ID'].str.extract(r'(\d+)', expand=False).astype(int) <= 100]

# Select only predictive features
data_selected = data_100[['Temperature', 'Humidity', 'Anomaly']].copy()
```

   Explainations: Extract only temperature and humidity (two features), binary target: 0 (normal) or 1 (anomaly), reduces dataset to 4800 samples for training

   b. **Feature Standardization**

```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- **Explainations**:
  - Compute mean and standard deviation from training data only.
  - Apply z-score normalization: $z = (x - mean)/std$
  - These mean/std values are embedded in the microcontroller

c. **Model Serialization to C Array**

The quantized TFLite model (bytearray) is converted to a C-compatible hex array format for embedding in firmware:

```
hex_lines = []
for i in range(0, len(tflite_model), 12):  # 12 bytes per line
    chunk = tflite_model[i:i+12]
    hex_line = ', '.join([f'0x{byte:02x}' for byte in chunk])
    hex_lines.append(hex_line)
hex_array = ',\n  '.join(hex_lines)
```

Finally, we create a header file (`dht_anomaly_model.h`) to have the process of model training done.

2. **Embedded Inference Logic**

```
model = tflite::GetModel(dht_anomaly_model_tflite);
```

The quantized model from `dht_anomaly_model.h` is a C byte array.

a. **Global Variables and Configuration:**

```
namespace {
    // TensorFlow Lite components
    tflite::ErrorReporter *error_reporter = nullptr;
    const tflite::Model *model = nullptr;
    tflite::MicroInterpreter *interpreter = nullptr;
    TfLiteTensor *input = nullptr;
    TfLiteTensor *output = nullptr;

    // Memory allocation for model execution
    constexpr int kTensorArenaSize = 8 * 1024;  // 8 KB working memory
    uint8_t tensor_arena[kTensorArenaSize];

    // Adaptive threshold mechanism
    float anomaly_threshold = 0.5;  // Initial threshold (adjusts over time)
    unsigned long last_adjustment = 0;
    int recent_anomalies = 0;       // Count in current window
    int total_readings = 0;         // Total readings in window
    int readings = 10;              // Window size for threshold adjustment
}
```

- **Adaptive Threshold Variables:**
  - `anomaly_threshold`: Dynamic threshold (starts at 0.5)

* `recent_anomalies`: Count of anomalies detected in last 10 readings
* `total_readings`: Total readings processed in current window
* `readings`: Window size (reset every 10 samples)

b. **Adaptive Threshold Algorithm**

- **Algorithm Logic (`SetAnomalyThreshold()`):**

```
Initial state: threshold = 0.5

Reading 1-10:
  Detected 4 anomalies -> ratio = 4/10 = 0.40 (40%)
  40% > 30% -> threshold += 0.05 = 0.55
  Interpretation: Too sensitive, raising bar

Reading 11-20:
  Detected 1 anomaly -> ratio = 1/10 = 0.10 (10%)
  10% in range [0.05, 0.30] -> no change
  Interpretation: Good balance

Reading 21-30:
  Detected 0 anomalies -> ratio = 0/10 = 0.00 (0%)
  0% < 5% -> threshold -= 0.05 = 0.50
  Interpretation: Missing anomalies, lowering bar
```

- **Benefits:**
  * **Self-calibrating**: Adapts to actual data distribution
  * **Prevents false positives:** Raises threshold if too many alerts
  * **Prevents false negatives:** Lowers threshold if missing anomalies
  * **Bounded:** Always stays between 0.3 and 0.8

c. **Main Inference Loop (`tiny_ml_task`)**

- **Queue Reception - Blocking Synchronization**
  * When `xQueueReceive()` is called: If queue is empty (no sensor data yet), TinyML task enters BLOCKED state, Task consumes 0% CPU (not spinning), FreeRTOS scheduler runs other tasks (LED, NeoPixel, etc.)
  * When sensor task enqueues data (every 5 seconds): TinyML task transitions to READY state, Scheduler selects it for execution, Task resumes from the `xQueueReceive()` call with received data.
- **Adaptive Threshold Counter Update**

```
1 total_readings++;
2 if (total_readings >= readings) {  // readings = 10
3     SetAnomalyThreshold();
4 }
```

Example execution:

```
Reading 1:  total_readings = 1  (< 10, skip adjustment)
Reading 2:  total_readings = 2  (< 10, skip adjustment)
...
Reading 10: total_readings = 10 (= 10, call SetAnomalyThreshold())

Inside SetAnomalyThreshold():
  anomaly_ratio = recent_anomalies / total_readings
  Example: 3 anomalies / 10 readings = 0.30 (30%)

  Decision:
  IF ratio > 0.30:  threshold += 0.05  (too sensitive)
  IF ratio < 0.05:  threshold -= 0.05  (not sensitive enough)
  ELSE: no change (5-30% is optimal)

  Reset counters:
  total_readings = 0
  recent_anomalies = 0

Reading 11: total_readings = 1  (new window starts)
```

– **Z-Score Normalization**

```
float temperature_normalized = (temperature - temperature_mean) / temperature_std;
float humidity_normalized = (humidity - humidity_mean) / humidity_std;
```

* **Explainations:** This is the most critical step because:
  · Neural network was trained on normalized data, not raw values
  · Must use exact same mean and std as training phase
  · Mismatch causes catastrophic prediction errors

Then writes normalized values into the TensorFlow Lite input tensor. The tensor is a memory buffer pre-allocated during `setupTinyML()`.
After putting the normalized values to the model, through the process of analyizng, we finally have the output is the single probability value between 0.0 and 1.0.

– **Three-Tier Classification**

```
if (result > anomaly_threshold + 0.2) {
    anomaly_type = "CRITICAL";
    anomaly_detected = true;
    recent_anomalies++;
}
else if (result >= anomaly_threshold) {
    anomaly_type = "WARNING";
    anomaly_detected = true;
```

```
9       recent_anomalies++;
10  }
11  else {
12      anomaly_type = "NORMAL";
13      anomaly_detected = false;
14  }
```

* **Classification rules (threshold = 0.5)**
  - `NORMAL: result < 0.50`

    Low confidence in anomaly ($<$ 50% probability)
  - `WARNING: 0.50 ≤ result < 0.70`

    Moderate confidence in anomaly (50–70% probability).

    Borderline case, needs investigation.
  - `CRITICAL: result ≥ 0.70`

    High confidence in anomaly ($\geq$ 70% probability).

    Clear anomaly, immediate action required.

  Example execution:

```
Example 1: result = 0.234, threshold = 0.50
  0.234 < 0.50 -> NORMAL
  Interpretation: Only 23.4% anomaly probability, definitely normal
  Action: No alert, continue monitoring

Example 2: result = 0.587, threshold = 0.50
  0.587 > 0.50 AND 0.587 < 0.70 -> WARNING
  Interpretation: 58.7% anomaly probability, borderline
  Action: Log for investigation, no immediate alarm

Example 3: result = 0.912, threshold = 0.50
  0.912 > 0.70 -> CRITICAL
  Interpretation: 91.2% anomaly probability, clear anomaly
  Action: Trigger alarm, alert operator, log event
```

– **Result Packaging and Mutex-Protected Serial Output**

```
1  MLResult ml_result;
2  ml_result.temperature = temperature;           // 28.5
3  ml_result.humidity = humidity;                 // 45.2
4  ml_result.inference_result = result;           // 0.234
5  ml_result.anomaly_detected = anomaly_detected; // false
6  ml_result.anomaly_type = anomaly_type;         // "NORMAL"
```

Example execution:

* Original raw sensor values (for logging/debugging)
* ML inference score (0.0-1.0 probability)
* Classification result (NORMAL/WARNING/CRITICAL)

* Boolean flag for quick anomaly check

Mutex-Protected Serial Output: Outputs results to serial console with mutex protection to prevent message interleaving from concurrent tasks.

– **Final Output:**

```
 Normal reading:
[TinyML] T:28.5 H:45.2 | Result:0.234 | Threshold:0.50 | Level:NORMAL

Warning reading:
[TinyML] T:32.5 H:65.8 | Result:0.587 | Threshold:0.50 | Level:WARNING

Critical reading:
[TinyML] T:42.3 H:88.5 | Result:0.912 | Threshold:0.50 | Level:CRITICAL

After threshold adjustment:
[TinyML] T:28.5 H:45.2 | Result:0.234 | Threshold:0.45 | Level:NORMAL
```

2.3.6 Task: Data Publishing to CoreIOT Cloud Server (`taskCoreIOT`)

After being created by the Web Server Task, the CoreIOT Data Publishing Task uses the message queue received from the `temp_hum_monitor` to publish data to the CoreIOT Cloud Server. In this implementation, we use the ThingsBoard library to establish two-way connectivity with the service.

For the code of this task, we created a new file, `taskCoreIOT.cpp`, and implemented it from scratch. We utilize functions provided by `task_core_iot.cpp` and `task_wifi.cpp` to streamline the implementation. Below is the source code for `taskCoreIOT.cpp`:

```
1  #include "taskCoreIOT.h"
2  SensorData data_receive;
3
4  void taskCoreIOT(void *pvParameters) {
5    pinMode(LED_PIN, OUTPUT);
6    xQueueReceive(xQueueForCoreIOT, &data_receive, portMAX_DELAY);
7
8    while(1) {
9      float temperature, humidity;
10     if(xQueueReceive(xQueueForCoreIOT, &data_receive, portMAX_DELAY) == pdTRUE) {
11       temperature = data_receive.temperature;
12       humidity = data_receive.humidity;
13     }
14     if(!isWifiConnected) {
15       Wifi_reconnect();
16     }
17     CORE_IOT_reconnect();
```

```
      CORE_IOT_sendata("telemetry", "temperature", String(temperature));
      CORE_IOT_sendata("telemetry", "humidity", String(humidity));
      if (attributesChanged) {
        CORE_IOT_sendata("attribute", "ledState", String(digitalRead(LED_PIN)));
        attributesChanged = false;
      }
      vTaskDelay(5000);
    }
  }
```

Key Implementation Highlights:

- **Queue Reception:** Similar to other tasks receiving data from the `temp_hum_monitor` task, this task utilizes an event-driven consumer pattern. The function `xQueueReceive(xQueueForCoreIOT, &data_receive, portMAX_DELAY)` is used to block execution when no data is available in the queue and resume processing once new data is enqueued.
- **Compulsory WiFi connection:** To successfully transport data to the cloud, a WiFi connection is mandatory. After the Web Server task switches the device to Station mode, we must verify the WiFi connection before uploading data. If the board loses connectivity, `Wifi_reconnect()` is called repeatedly until the connection is restored.
- **Upload data to the cloud:** The `CORE_IOT_sendata(...)` function from `task_core_iot` is used to send telemetry (temperature and humidity) and attributes (LED_PIN state) to the CoreIOT platform. Before uploading, we must also connect to the device on app.coreiot.io (using `CORE_IOT_reconnect`).
- **Solution template:** Based on the specification requirements, the "Temperature & Humidity Sensors" template (see Figure 15) is the most suitable for this project. This template allows us to create a new device with options such as high temperature and low humidity thresholds, as well as longitude and latitude for the device's location. We configured the device with thresholds of 30°C for high temperature and 60% for low humidity, located at the Ly Thuong Kiet campus. Furthermore, devices created with this template use a profile named `Temperature Sensor`, where the storage structure for published telemetry and attributes differs from the default profile (see Figure 16). Consequently, we needed to modify the ThingsBoard library to correctly publish the data.

```
constexpr char ATTRIBUTE_TOPIC[] PROGMEM = "esp/attributes";
constexpr char TELEMETRY_TOPIC[] PROGMEM = "esp/telemetry";
```

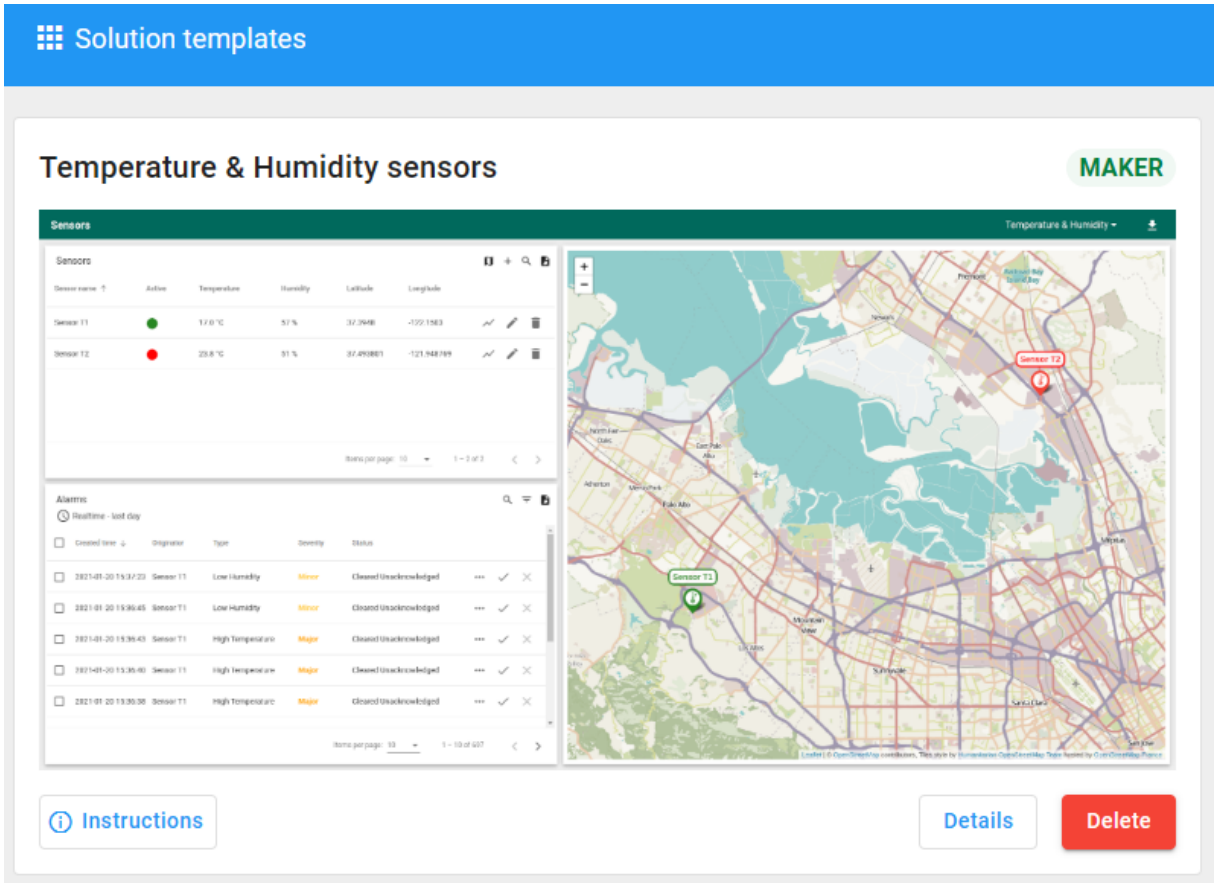

Figure 15: Solution template that we used

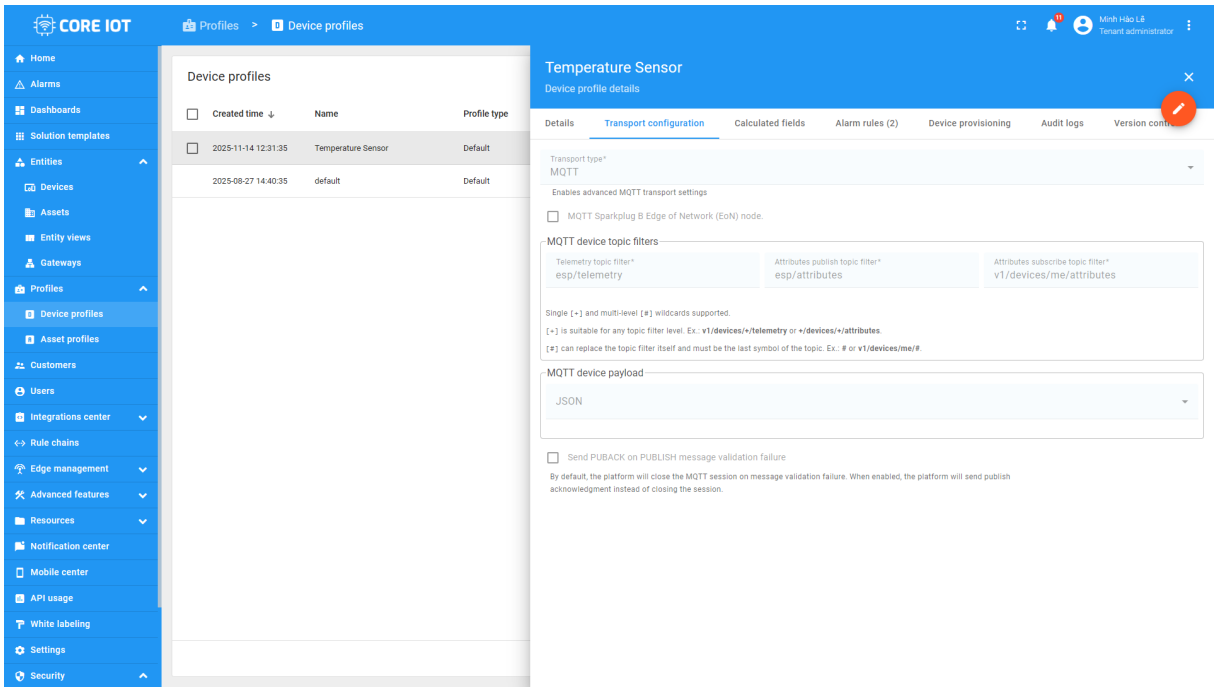

Figure 16: Temperature Sensor profile

– **Main Dashboard:** The main dashboard page (see Figure 17) consists of three sections. The Sensors section allows us to create, delete, visualize data, and manage all devices. The second section is the Map, which displays the location of each device based on its longitude and latitude. The final section is Alarms; as previously mentioned, the solution template supports two thresholds (high temperature and low humidity). If a reading exceeds the set limits, the system triggers an alarm, displays it in this section, and sends a notification.

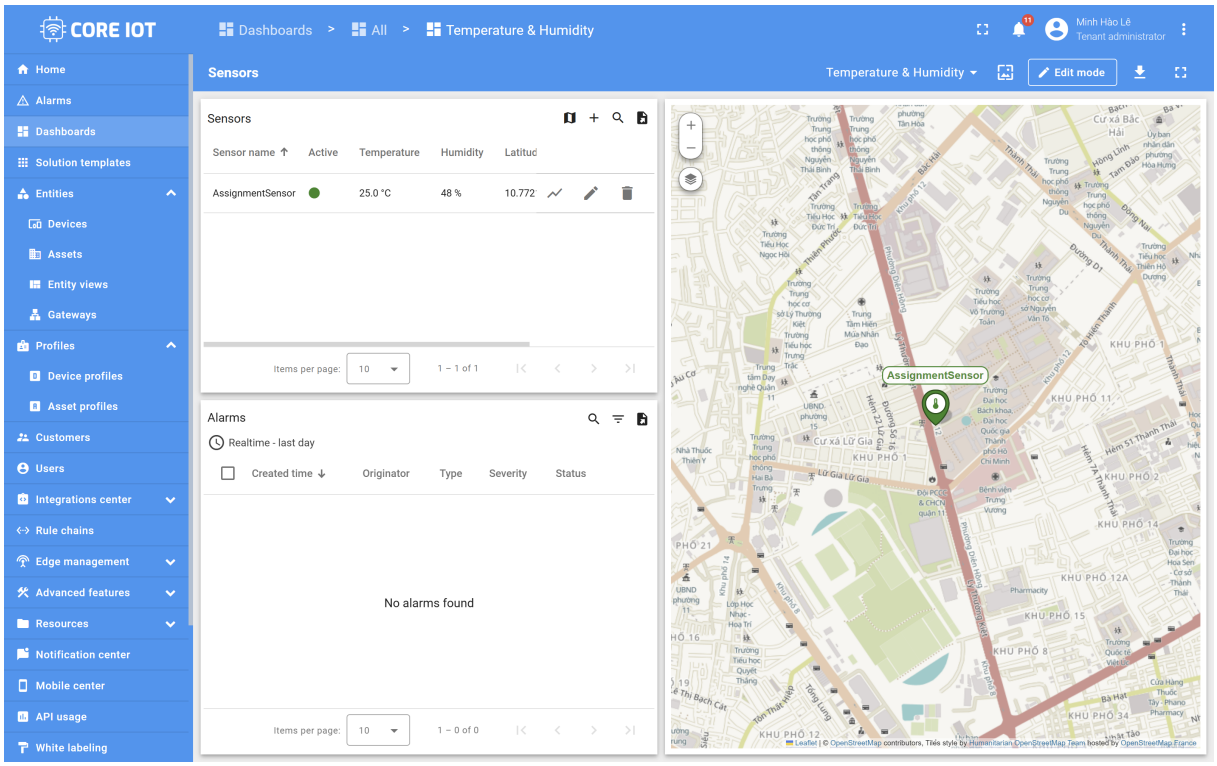

Figure 17: Main Dashboard

– **Sensor Management:** Clicking on a specific sensor on the dashboard navigates the user to the sensor management page. On this page, we can configure the two thresholds mentioned above. The primary feature of this view is two graphs displaying the temperature and humidity data published by the board. In Figure 18, the time window is set to 2 hours real-time, using average aggregation with a 15-second grouping interval.

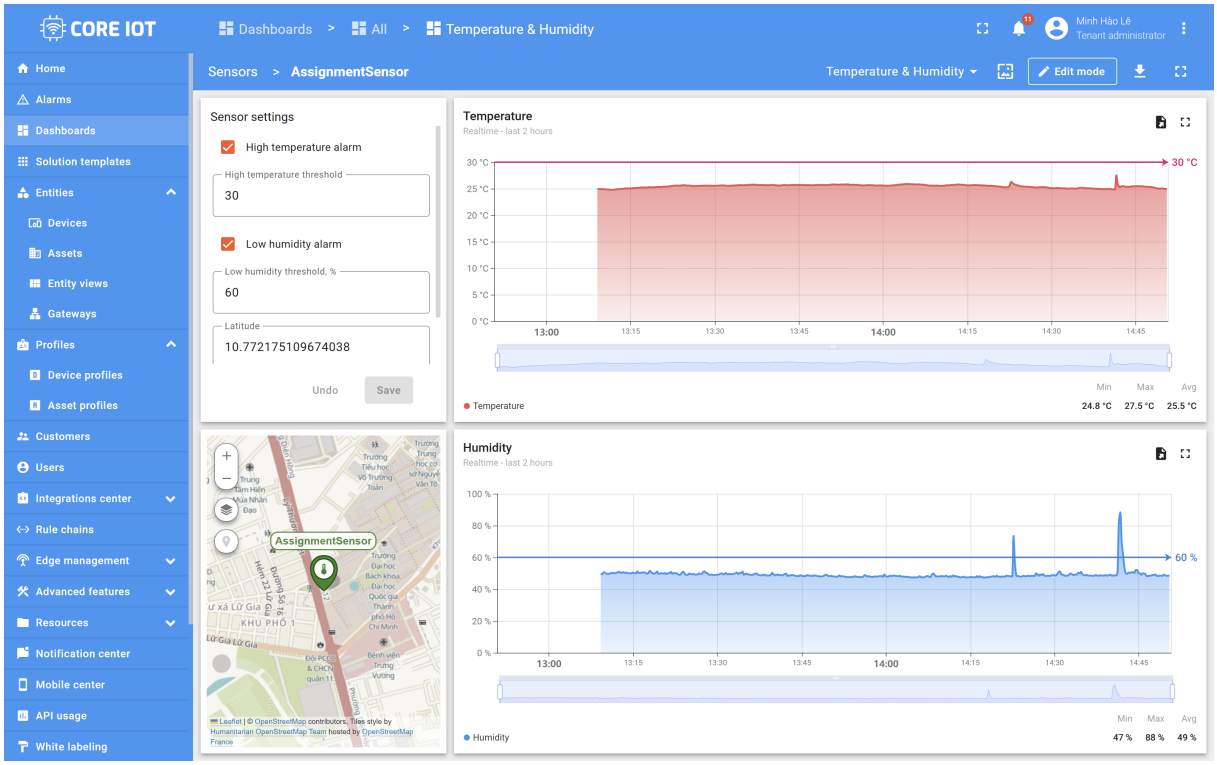

Figure 18: Sensor Management

– **Control LED from CoreIOT:** Similar to Task 2 in the Lab, we created another dashboard with a Switch Control widget to toggle the LED on the physical system (see Figure 19). It uses the `setLedSwitchState` function to handle the RPC request. When we toggle the switch, it toggles the LED and displays the state change in the Serial Terminal (see Figure 20). This action is shown in the last part of the demonstration video at the end of this report.

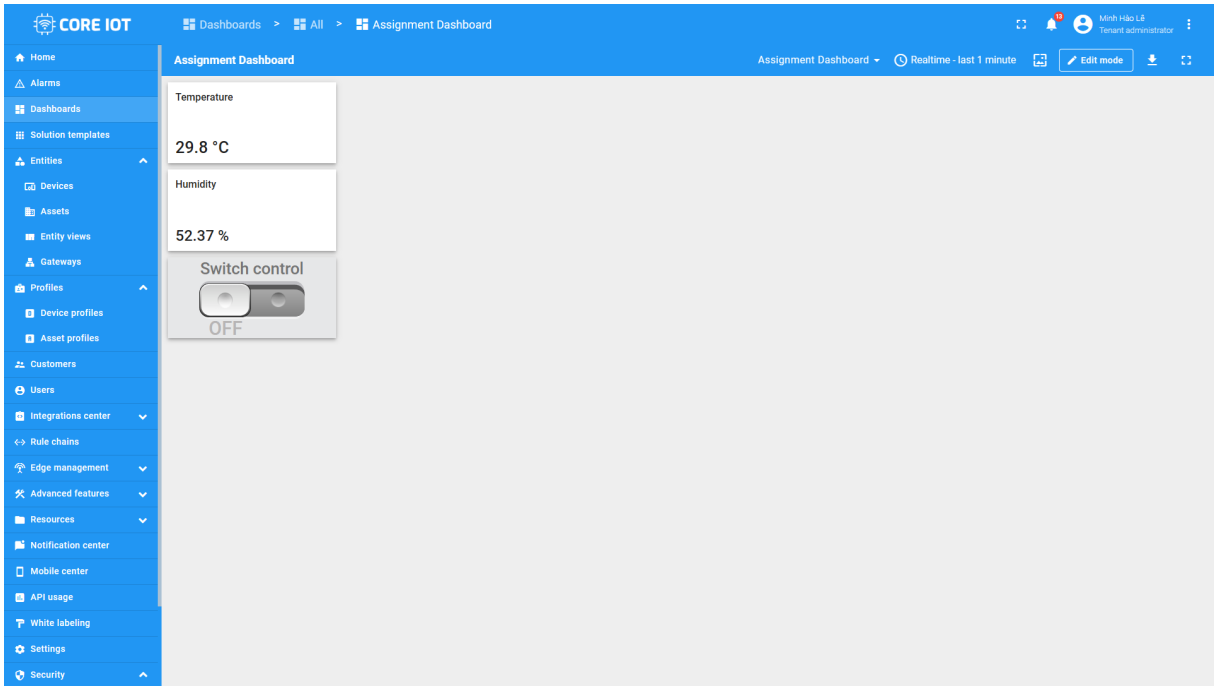

Figure 19: Dashboard with Switch control

```
Received Switch state
Switch state change: 1
[LED] Temperature: Frequency: 1.0 HZ
[TinyML] T:29.38 H:52.97 | Result:0.37 | Threshold:0.35 | Level:WARNING
NeoPixel Update - Humidity: 52.97% | Status: LOW | Color: Red
[LED] Temperature: 29.42°C | Frequency: 1.0 HZ
[TinyML] T:29.42 H:52.74 | Result:0.36 | Threshold:0.35 | Level:WARNING
NeoPixel Update - Humidity: 52.74% | Status: LOW | Color: Red
Received Switch state
Switch state change: 0
```

Figure 20: Serial Monitor showing the state changes

– **Alarm notifications:** As mentioned earlier, we configured two thresholds: one for temperature and one for humidity. During the operation of the system, several alarms were triggered and recorded in the mailbox (see Figure 21).

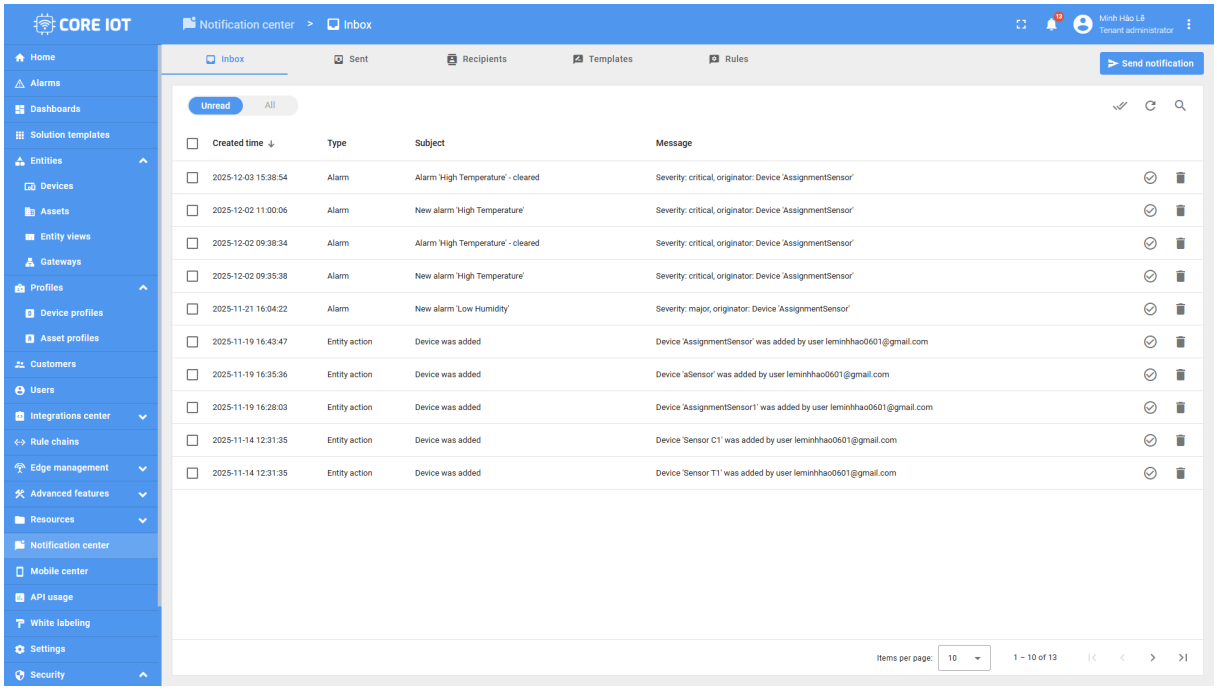

Figure 21: Mail inbox on CoreIOT

3 Source Code and Demonstration

To ensure transparency and reproducibility, the complete implementation of the project has been made publicly available.

3.1 Project Repository

The full source code, including the C++ implementation, the modified libraries, is hosted on GitHub. This repository serves as a comprehensive reference for the project structure and configuration logic. You can access the repository at the following link:
Repository: Source Code (GitHub)

3.2 Demonstration Video

For a visual verification of the system's functionality, a demonstration video has been uploaded to YouTube. This video showcases the fulfillment of all project requirements, including the LED blinking signal triggered by temperature, the NeoPixel LED controlled by humidity, and the LCD displaying sensor information. Additionally, we demonstrate our custom-designed web server, accessible via an Access Point connection, which displays sensor readings and machine learning results. It also features two buttons to control the system's LEDs. Finally, the video also includes the real-time data synchronization between the device and the CoreIOT

platform, as well as the successful execution of RPC commands for LED control.

Watch the demo here: Demo Video