VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Logic Design Project

Group 04

SMART HOME

Advisor(s): Nguyen Thien An

No.	Member	Stu ID	Workload	Contribution
1	Le Manh Quan	2352993	Project Manager, 4 sensors, Up MQTT (CoreIot), Video	100%
2	Vo Trung Thanh	2353108	2 sensors, AI, AES, MQTT Broker	100%
3	Nguyen Cong Thanh	2353100	Draw + Real Home, Report	100%
4	Ho Cong Thanh	2353097	Dashboard + Alarms (CoreIot), Video	100%

HO CHI MINH CITY, DECEMBER 2025

Contents

1 Introduction

1.1 Introduction and Objectives

The rapid expansion of the Internet of Things (IoT) has introduced significant challenges regarding data privacy and real-time control. This project aims to address these challenges by designing a secure, hybrid IoT architecture. The primary objective is to create a seamless bridge between local edge devices (ESP32 microcontrollers) and a cloud platform (Core IoT), ensuring that all sensitive telemetry data is encrypted before transmission.

The specific technical goals of this system are:

- **Edge Security:** Implementation of AES-128 encryption on the microcontroller to secure sensor data.
- **Protocol Interoperability:** Development of a Python-based middleware to translate between the lightweight MQTT protocol used locally and the JSON-RPC format required by the Cloud.
- **AI Integration:** Incorporation of a Face Recognition module that bypasses the cloud for immediate, low-latency actuation of physical security barriers (doors).

1.2 System Architecture and Data Flow

The system architecture is organized into three distinct layers: the Perception Layer (Sensors/MCUs), the Network/Gateway Layer (Broker/Python Bridge), and the Application Layer (Cloud Dashboard).

As illustrated in Figure 1.1, the data flows through the following stages:

1. **Data Collection & Encryption:** MCU 1 and MCU 2 collect environmental data. Before the data leaves the device, it is serialized into JSON and encrypted using AES-128-ECB.
2. **Local Transmission:** The encrypted payload is published to the local MQTT broker on topics `home/nodes/node1` and `home/nodes/node2`.
3. **Gateway Processing:** A Python subscriber acts as the decryption gateway. It listens to the local topics, decrypts the payload using the shared secret key, and parses the original JSON data.

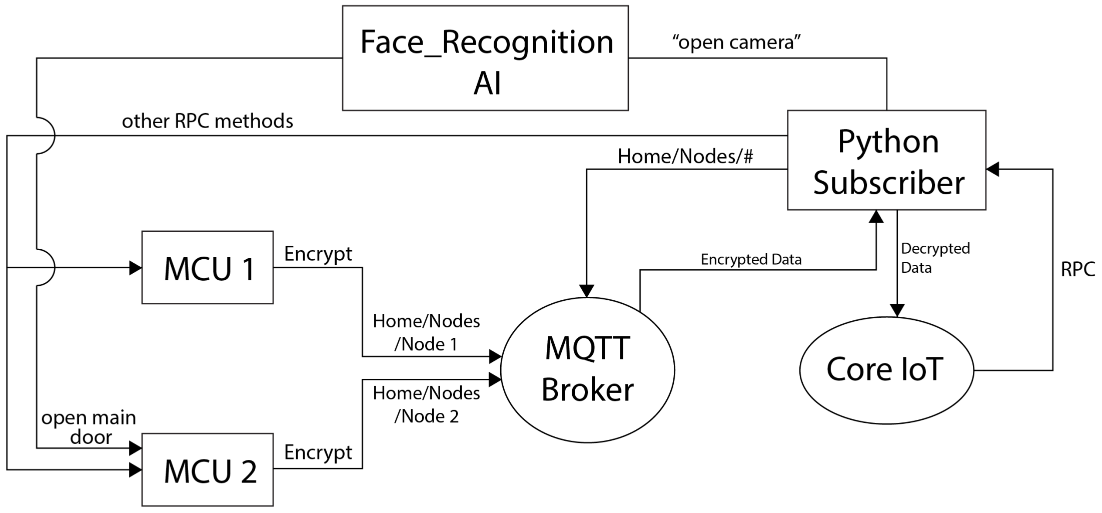


Figure 1.1: System Algorithm: From Edge Encryption to Cloud Visualization

4. **Cloud Forwarding:** The decrypted data is forwarded to the Core IoT platform via the topic `v1/devices/me/telemetry` for visualization.

5. **Cloud-to-Device Control (RPC):** User commands from the dashboard (e.g., "Turn on Pump") travel in reverse. The Python bridge receives the RPC request and publishes a specific command to the local MQTT network (e.g., `home/pump/cmd`) to trigger the actuator.

1.3 Theoretical Framework: The MQTT Protocol

The system relies on MQTT (Message Queuing Telemetry Transport), a standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport.

- **Publish/Subscribe Model:** Unlike the client-server model (HTTP), MQTT decouples the creator of a message (Publisher: MCU 1) from the consumer (Subscriber: Python Bridge). This ensures that if the Python bridge is temporarily offline, the MCU can still operate and publish, or the broker can retain the message.

- **Topic Hierarchy:** The system uses a structured topic tree. The Python bridge subscribes to `home/nodes/#`. The wildcard `#` allows the gateway to receive messages from all current and future nodes without modifying the code, enhancing scalability.

1.4 Source code

Link github: Click here

1.5 House schematic

1.5.1 2D schematic

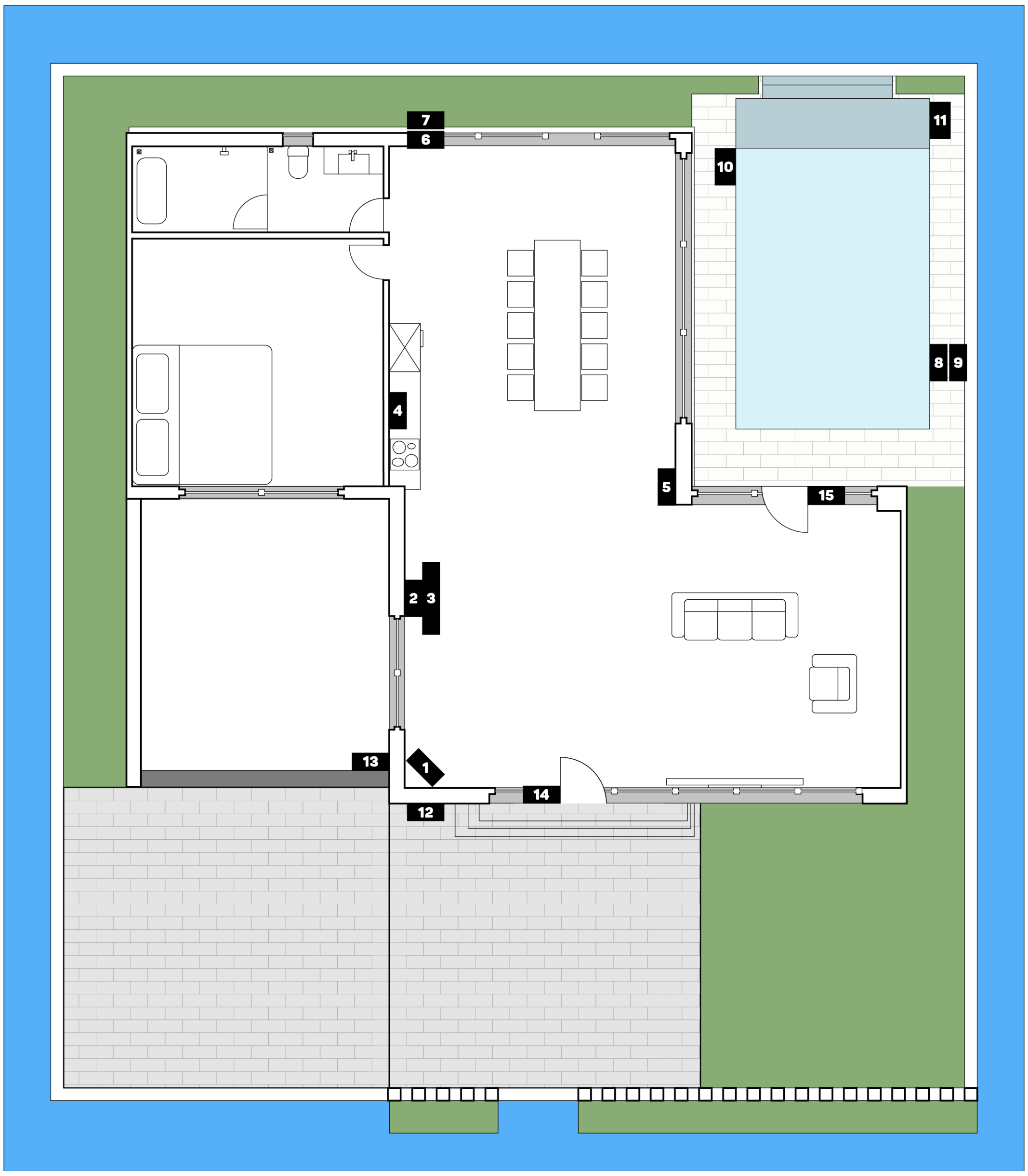


Figure 1.2: 2D House schematic

As illustrated in Figure 1.4:

1. Passive Infrared (PIR) Motion Sensing
2. DC Fan Control (Relay Driver)
3. Liquid Crystal Display (LCD) via I2C
4. Metal Oxide Semiconductor Gas Sensing (MQ-135)
5. DHT20 Temperature and Humidity Sensing
6. Electrical Fan
7. Electromechanical Relay
8. Addressable LED Protocol (NeoPixels)
9. Photoresistor (LDR)
10. Resistive Soil Moisture Sensing
11. Water Pump
12. Ultrasonic Distance Measurement (HC-SR04)
13. Ultrasonic Servo (Garage Door)
14. Button Servo - (Main Door)
15. Button Servo - (Garden Door)

1.5.2 3D schematic

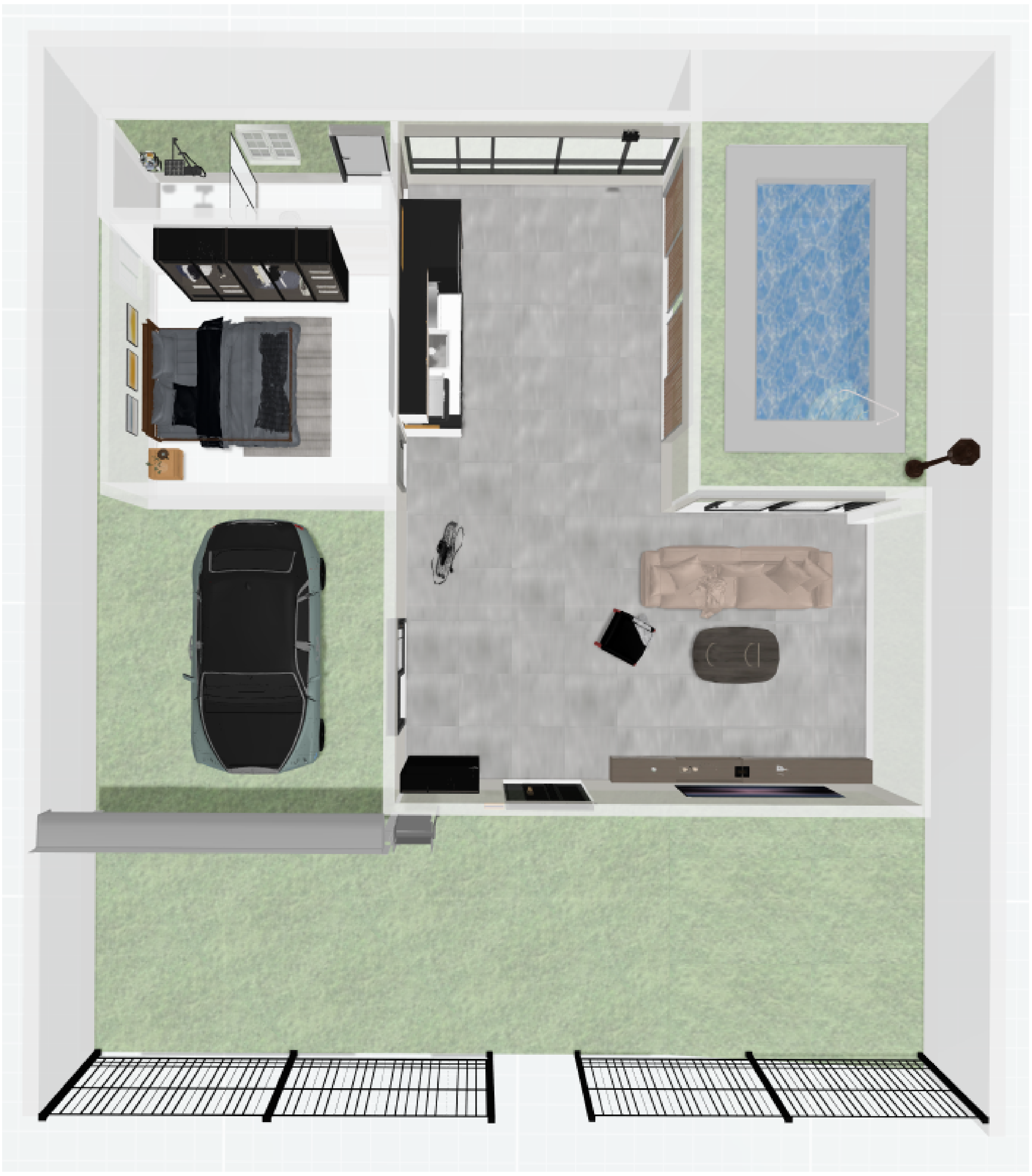


Figure 1.3: 3D House schematic - above view

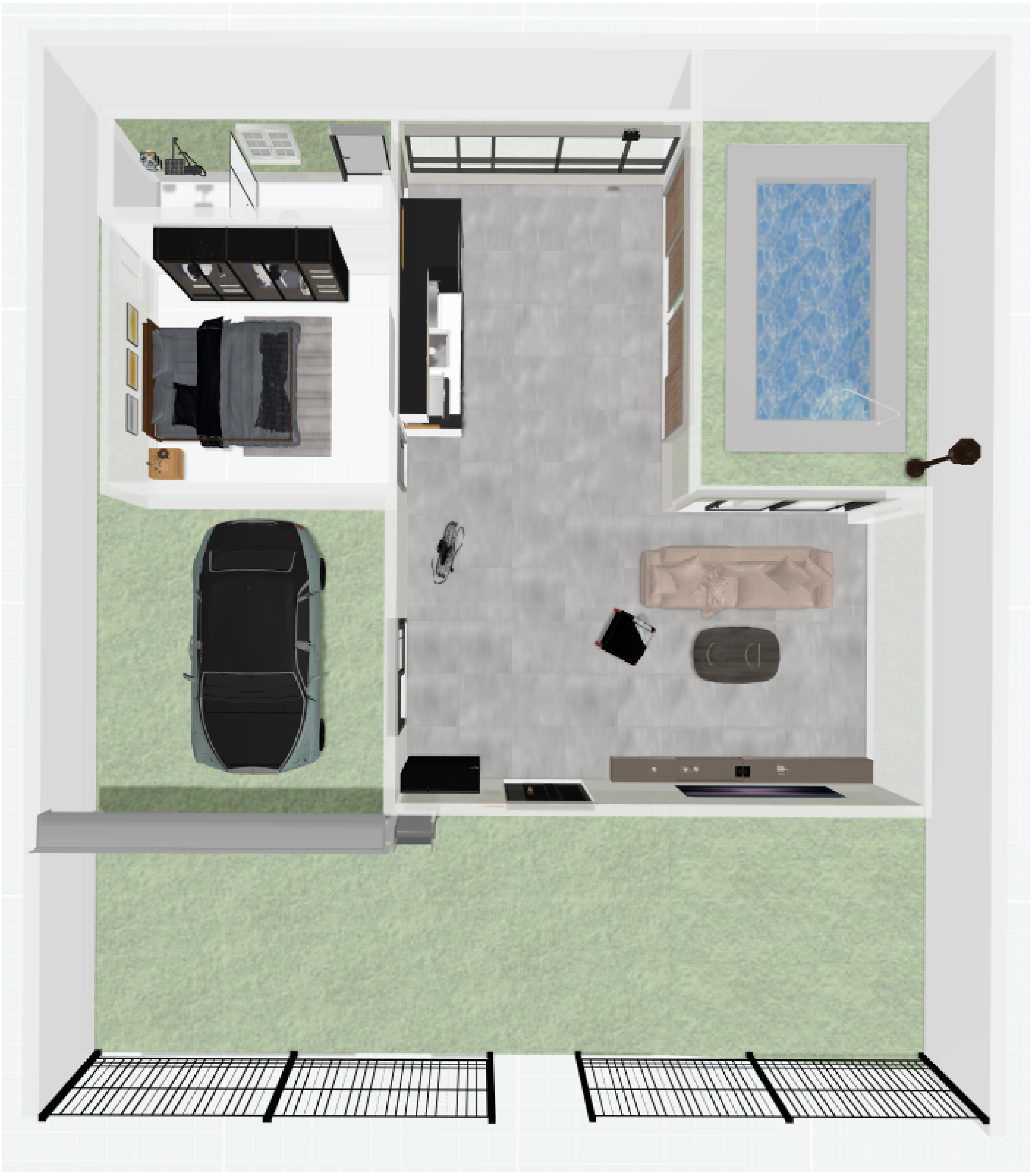


Figure 1.4: 3D House schematic - side view

2 MCU 1 Implementation: Environmental Safety Node

2.1 Assigned Task

The primary objective of the first microcontroller unit (MCU 1) is to function as an autonomous "Environmental Safety Node." It is responsible for monitoring potentially hazardous conditions (gas leaks) and maintaining agricultural parameters (soil moisture) using specific hardware interfaces:

- **Gas Monitoring & Ventilation:** Continuously sample air quality using an MQ-135 sensor. If hazardous levels are detected, trigger a **Relay Module** to activate a high-power Exhaust Fan.
- **Irrigation Control:** Monitor soil moisture levels using a resistive sensor and control a water pump via a feedback loop to prevent "dry-running."
- **Actuation Logic:** Use Digital I/O for the Relay (Fan) and Pulse Width Modulation (PWM) for the Pump.
- **Real-Time Multitasking:** Utilize FreeRTOS to ensure that critical safety checks (Gas) are not blocked by slower tasks (Pump logic).

2.2 Theoretical Framework

To understand the hardware-software integration, we analyze the physical principles of the sensors and the actuator drivers employed.

2.2.1 Metal Oxide Semiconductor Gas Sensing (MQ-135)

The MQ-135 sensor detects gases such as Ammonia (NH_3), Benzene, and Smoke based on the variable conductivity of Tin Dioxide (SnO_2).

- **Mechanism:** In clean air, oxygen adsorbs onto the SnO_2 surface, trapping electrons and increasing electrical resistance. When combustible gases exist, they react with this oxygen, releasing electrons and lowering the resistance.
- **Detection:** The ESP32 measures this resistance drop as an analog voltage increase via the ADC.

2.2.2 Electromechanical Relay Theory (Fan Control)

The Exhaust Fan is a high-power device (typically 12V or 220V) that cannot be driven directly by the ESP32's GPIO (which supplies only 3.3V/40mA). We utilize a **Relay Module** to bridge this gap.

- **Principle:** A relay acts as an electrically operated switch. It consists of a coil and a set of contacts (Common, Normally Open, Normally Closed).
- **Operation:** When the ESP32 sends a `HIGH` signal to the relay driver (transistor/optocoupler), current flows through the coil, generating a magnetic field. This field physically pulls the armature, closing the contacts and completing the high-voltage circuit for the Fan.
- **Isolation:** The relay provides galvanic isolation, protecting the microcontroller from high-voltage spikes generated by the inductive load of the fan motor.

2.2.3 Resistive Soil Moisture Sensing

The system uses a resistive sensor to determine soil water content.

- **Theory:** Water acts as an electrolyte. The sensor measures the resistance between two probes inserted into the soil.
- **Relationship:** High Moisture $\rightarrow$ High Conductivity $\rightarrow$ Low Resistance $\rightarrow$ Low Voltage Drop (depending on circuit).
- **ADC Conversion:** The ESP32's 12-bit ADC converts the analog signal $(0 - 3.3V)$ into a digital value $(0 - 4095)$. In our code, a raw reading $\geq$ 2400 indicates high resistance (Dry Soil).

2.2.4 DC Motor Control and PWM Theory (Water Pump)

The irrigation system utilizes a 12V DC Water Pump. Similar to the fan, this inductive load requires a driver circuit (typically a MOSFET).

- **PWM Principle:** PWM simulates a variable analog voltage output by rapidly switching a digital pin between HIGH (3.3V) and LOW (0V).
- **Average Voltage Formula:** The effective voltage V_{avg} seen by the motor is calculated as:

$$V_{avg} = V_{supply} \times D = V_{supply} \times \frac{T_{on}}{T_{period}} \tag{2.1}$$

- **Implementation:** The firmware utilizes an 8-bit resolution (0-255). A value of 250 corresponds to a duty cycle of $D \approx 98\%$.

2.3 Software Implementation and Code Analysis

The firmware defines two primary tasks in ‘sensors.cpp‘: `gas_monitor` (for the Fan/Relay) and `pump_control` (for the Pump).

2.3.1 Task 1: Gas Monitoring and Relay Control

The `gas_monitor` task continuously polls the air quality. It controls the Relay on `FAN_PIN` using digital logic.

Listing 2.1: Gas Safety and Relay Logic

```
void gas_monitor(void *pvParameters) {
    // Configure Relay Pin
    pinMode(FAN_PIN, OUTPUT);
    digitalWrite(FAN_PIN, LOW);  // Initial state: Relay Open
        (Fan OFF)

    while (1) {
        // 1. Read Sensor
        int raw = analogRead(GAS_SENSOR_PIN);
        gas_value = raw;

        // 2. Threshold Logic
        if (raw > GAS_THRESHOLD) {
            gas_alert = true;
            // Activate Relay (Close contacts)
            digitalWrite(FAN_PIN, HIGH);
        } else {
            gas_alert = false;
            // Deactivate Relay (Open contacts)
            digitalWrite(FAN_PIN, LOW);
        }

        Serial.printf("MQ135 raw=%d alert=%d\n", raw,
           gas_alert ? 1 : 0);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

Analysis: The code uses `digitalWrite` because a relay is a binary device (ON or OFF). When `gas_alert` is true, the pin goes HIGH, energizing the relay coil and turning on the Fan.

2.3.2 Task 2: Pump Control with Dry-Run Protection

The `pump_control` task manages the irrigation pump. Unlike the relay-driven fan, the pump is driven via PWM (Pulse Width Modulation) to allow for potential speed adjustments, though it currently runs at near-max speed (250/255).

Listing 2.2: Pump Control Logic

```
void pump_control(void *pvParameters) {
  pinMode(PUMP_PIN, OUTPUT);
  digitalWrite(PUMP_PIN, LOW);

  while (1) {
    int raw = analogRead(MOIST_SENSOR_PIN);
    moist_value = raw;

    // SAFETY CHECK: High Resistance = Dry Soil or
       Disconnected Sensor
    if (raw >= 2400) {
      if (pump_on) {
        Serial.println("[PUMP] Moisture high -> auto stop");
      }
      pump_on = false;            // Override global command
      analogWrite(PUMP_PIN, 0); // Stop PWM
    }
    else {
      // Remote Command Processing
      if (pump_on) {
            analogWrite(PUMP_PIN, 250);    // PWM Duty Cycle ~98%
        } else {
            analogWrite(PUMP_PIN, 0);      // Stop
        }
    }
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}
```

3 MCU 2 Implementation: Smart Access and Home Automation

3.1 Assigned Task

The second microcontroller unit (MCU 2) serves as the "Home Automation and Access Control Node." Unlike MCU 1 which focuses on safety, MCU 2 handles user comfort, physical security, and visual feedback. Its specific functional requirements are:

- **Smart Access Control:** Operate three independent servo motors for the Main Door (AI-triggered), Garage Door (Sensor-triggered), and Garden Gate (Remote-triggered).
- **Smart Lighting:** Control a WS2812B (NeoPixel) LED strip for the swimming pool, supporting both automatic (light-sensor based) and manual modes.
- **Environmental Display:** Monitor local temperature and humidity using a DHT20 sensor and display real-time values on a 16x2 I2C LCD.
- **Motion-Activated Cooling:** Control a cooling fan using a PIR motion sensor with remote override capabilities.

3.2 Theoretical Framework

3.2.1 Servo Motor Control (PWM)

To control the physical doors, the system uses positional rotation servo motors (e.g., MG996R or SG90). These actuators are controlled using Pulse Width Modulation (PWM) at a frequency of 50Hz.

- **Signal Protocol:** The servo expects a control pulse every 20ms. The width of this high pulse determines the angular position.
- **Mapping:**
  - Pulse width $\approx$ 0.5ms $\rightarrow$ 0° (Closed)
  - Pulse width $\approx$ 1.5ms $\rightarrow$ 90° (Open)
- **Implementation:** The code uses the `ESP32PWM` library to allocate hardware timers and map these microsecond values to the servo pins.

3.2.2 Ultrasonic Distance Measurement (HC-SR04)

The Garage Door logic relies on an ultrasonic sensor to detect the presence of a car based on the "Time of Flight" principle.

- **Trigger:** The MCU sends a 10μs HIGH pulse to the `ULTRASONIC_TRIG_PIN`.
- **Echo:** The sensor emits an 8-cycle sonic burst at 40kHz. The `ULTRASONIC_ECHO_PIN` goes HIGH for the duration it takes the sound to travel to the object and back.
- **Calculation:** Distance d (cm) is calculated as:

$$d = \frac{\text{Duration}(\mu s) \times 0.0343}{2} \tag{3.1}$$

  where 0.0343 cm/μs is the speed of sound in air.

3.2.3 Photoresistor (LDR) Theory

The automated pool lighting system uses a Light Dependent Resistor (LDR).

- **Principle:** The LDR is made of Cadmium Sulfide (CdS). When photons hit the surface, conductivity increases, causing resistance to drop.
- **Voltage Divider:** The ESP32 measures the voltage at the junction of the LDR and a fixed resistor. A low analog reading indicates high light intensity.

3.2.4 Addressable LED Protocol (WS2812B)

The Pool LED system uses WS2812B LEDs (NeoPixels). Unlike standard LEDs, these allow for individual 24-bit color mixing (Red, Green, Blue).

- **RGB Mixing:** Each LED package contains three dies. By varying the intensity (0-255), over 16 million colors can be produced.
- **Data Protocol:** The ESP32 sends a high-speed data stream (800kHz) where logic '0' and '1' are encoded by varying pulse widths on a single wire.

3.2.5 Passive Infrared (PIR) Motion Sensing

The automatic cooling system utilizes a PIR sensor (e.g., HC-SR501) to detect human presence.

- **Principle:** All objects with a temperature above absolute zero emit infrared (IR) radiation. The PIR sensor contains a pyroelectric crystal that generates a voltage when exposed to IR variations.
- **Fresnel Lens:** A plastic lens focuses IR radiation from different angles onto the sensor, creating "detection zones."
- **Operation:** When a warm body moves across these zones, the sensor detects a differential change in IR levels and outputs a digital `HIGH` signal polled by the MCU.

3.2.6 DC Fan Control (Relay Driver)

The cooling fan is a high-current inductive load (typically 12V DC) that requires isolation from the 3.3V ESP32 logic.

- **Relay Switching:** Unlike the Water Pump which uses PWM, the Fan is operated in a binary state (ON/OFF) using an electromagnetic relay.
- **Logic:** The ESP32 drives a transistor to energize the relay coil, closing the high-voltage contacts.
- **Active State:** The code defines `RELAY_ACTIVE_LOW` to handle specific relay module logic.

3.2.7 DHT20 Temperature and Humidity Sensing

Environmental monitoring is performed by the DHT20, an upgraded I2C version of the classic DHT11/22.

- **Communication:** Uses the I2C bus (SDA/SCL), allowing for robust digital communication without timing-critical pulse reading.
- **Sensing Elements:**
  - **Humidity:** A capacitive sensing element changes capacitance based on water vapor absorption.
  - **Temperature:** An internal band-gap sensor measures thermal changes.
- **Data Output:** The sensor outputs calibrated digital values, which the ESP32 reads to update global variables.

3.2.8 Liquid Crystal Display (LCD) via I2C

The system provides visual feedback using a 16x2 Character LCD.

- **Hardware Abstraction:** A standard HD44780 controller requires 6-10 GPIO pins. To save pins on the ESP32, an I2C I/O expander (PCF8574) is used.
- **Memory Map:** The display has an internal DDRAM. The ESP32 sends ASCII character codes to specific addresses to render text like "Temp: 25.0 C".

3.3 Software Implementation and Code Analysis

MCU 2 is the most complex node, running multiple concurrent FreeRTOS tasks to handle the diverse hardware.

3.3.1 Task 1: Ultrasonic Garage Door Control

The `ultrasonic_task` manages the Garage Door. It implements an automatic entry system with a timeout feature.

- **Auto-Open:** If an object (car) is detected within the `NEAR_CM` range (distance > 0 and distance $<$ `NEAR_CM`), the door opens to 90 degrees.
- **Auto-Close Timer:** Once opened, the system sets a deadline (`s_servo_until_ms`) for 10 seconds. After this period, the door automatically closes to 0 degrees.

Listing 3.1: Ultrasonic Garage Door Logic

```
void ultrasonic_task(void*) {
  // ... Init Servo and Pins ...
  for (;;) {
    // 1. Measure Distance
    long duration = pulseIn(ULTRASONIC_ECHO_PIN, HIGH, 30000)
      ;
    float distance = (duration * 0.0343f / 2.0f);
    glob_distance = distance;

    uint32_t now = millis();

    // 2. Logic: Auto Open if Close Object Detected
    if (!s_servo_open && distance > 0 && distance < NEAR_CM)
     {
      servo_set_angle(DOOR_OPEN_ANGLE);      // 90 deg
      s_servo_open     = true;
      s_servo_until_ms = now + 10000UL;      // 10 seconds
         timer
      Serial.println("[Door] OPEN (ultrasonic, 10s)");
    }

    // 3. Logic: Auto Close on Timeout
    if (s_servo_open && now > s_servo_until_ms) {
      servo_set_angle(DOOR_CLOSED_ANGLE);    // 0 deg
      s_servo_open     = false;
      Serial.println("[Door] CLOSE (timeout 10s)");
    }
    vTaskDelay(pdMS_TO_TICKS(150));
  }
}
```

3.3.2 Task 2: Smart Lighting Control (LDR & NeoPixel)

The `light_task` handles the automation, while the `set_led` function handles the hardware driving.

Logic Flow:

- **Sensing:** The task polls the LDR every 300ms. If `raw < s_threshold`, it requests the LEDs to turn ON.
- **Manual Override:** A boolean flag `glob_led_override` allows the user to seize control via MQTT. If set, the sensor logic is ignored.

Listing 3.2: Lighting Automation with Manual Override

```
void light_task(void* /*pv*/) {
  const TickType_t period = pdMS_TO_TICKS(300);
  TickType_t last = xTaskGetTickCount();

  while(1) {
    int raw = analogRead(s_pin_adc);
    glob_light_raw = raw;

    // Check Manual Override Flag
    if (!glob_led_override) {
        bool on = (raw < s_threshold); // Dark detection
        if (on != glob_led_state) {
            set_led(on);
        }
    }
    vTaskDelayUntil(&last, period);
  }
}
```

3.3.3 Task 3: AI-Driven Main Door

The Main Door is controlled by the `ai_servo_task`. This task is purely reactive to the MQTT command `OPEN_AI`. When the Face Recognition system identifies a user, it sends this command, triggering the servo to open for 3 seconds.

4 Encryption Implementation

4.1 Assigned Task

To ensure data privacy and integrity within the local IoT network, a security layer was implemented for both Edge nodes (MCU 1 and MCU 2). The specific requirements were:

- **Algorithm:** Implement **AES-128** (Advanced Encryption Standard) with a 128-bit symmetric key.
- **Mode of Operation:** Use ECB (Electronic Codebook) mode for block processing.
- **Data Formatting:** Apply PKCS#7 Padding to handle variable-length JSON payloads and Base64 Encoding for safe transport over MQTT text-based topics.
- **Gateway Decryption:** Develop a Python-based decryption engine on the gateway to restore the original telemetry data before forwarding it to the Cloud.

4.2 Theoretical Framework

4.2.1 Symmetric Encryption (AES)

The system uses Symmetric Key Cryptography, meaning the same key is used for both encryption (at the Edge) and decryption (at the Gateway).

- **Block Size:** AES operates on fixed-size blocks of 128 bits (16 bytes).
- **Key Size:** We utilize a 128-bit key (16 bytes). The key defined in the firmware is: `0x01, 0x02, ..., 0x10`.

4.2.2 PKCS#7 Padding

Since the sensor data (formatted as a JSON string) rarely aligns perfectly with the 16-byte block boundary, padding is required. **Theory:** PKCS#7 adds N bytes, each having the value N, where N is the number of bytes needed to reach the next block multiple.

$$\text{Pad Value} = \text{BlockSize} - (\text{DataLength} \mod \text{BlockSize}) \tag{4.1}$$

Example: If data is 14 bytes, we add two bytes of `0x02`. If data is 16 bytes, we add a full block of sixteen `0x10` bytes.

4.3 Edge Implementation (C++ on MCU 1 & 2)

Both MCUs utilize the `mbedtls` library (standard in the ESP32 SDK) to perform the encryption. The process involves three distinct steps:

1. **Serialization:** Constructing a plain JSON string containing the sensor data.
2. **Encryption:** Passing that string to the `aesEncryptToBase64` function.
3. **Encapsulation:** Wrapping the resulting ciphertext in a final JSON object that identifies the device.

4.3.1 Core Encryption Function

The underlying function used by both nodes is identical. It handles the PKCS#7 padding calculation and the AES-ECB transformation.

Listing 4.1: AES Encryption with Padding and Base64 (C++)

```cpp
#include "mbedtls/aes.h"
#include "mbedtls/base64.h"

// Symmetric Key (Must match Python side)
static const uint8_t AES_KEY[16] = {
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
  0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10
};

String aesEncryptToBase64(const String &plain) {
  // 1. Calculate PKCS#7 Padding
  size_t len = plain.length();
  size_t blockSize = 16;
  size_t paddedLen = ((len / blockSize) + 1) * blockSize;

  std::vector<unsigned char> input(paddedLen);
  memcpy(input.data(), plain.c_str(), len);

  uint8_t pad = paddedLen - len;
  for (size_t i = len; i < paddedLen; i++) {
      input[i] = pad;  // Apply padding bytes
  }

  // 2. AES Encryption (ECB Mode)
  std::vector<unsigned char> output(paddedLen);
  mbedtls_aes_context aes;
  mbedtls_aes_init(&aes);
  mbedtls_aes_setkey_enc(&aes, AES_KEY, 128);

  for (size_t i = 0; i < paddedLen; i += blockSize) {
    mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_ENCRYPT,
                              input.data() + i,
                              output.data() + i);
  }
  mbedtls_aes_free(&aes);

  // 3. Base64 Encoding
  size_t outLen = 0;
  mbedtls_base64_encode(nullptr, 0, &outLen, output.data(),
     paddedLen);

  std::vector<unsigned char> b64(outLen + 1);
  mbedtls_base64_encode(b64.data(), outLen, &outLen, output.
     data(), paddedLen);
  b64[outLen] = '\0';

  return String((char*)b64.data());
}
```

4.3.2 Payload Packing: MCU 1 (Node 1)

MCU 1 monitors soil moisture and gas levels. In the `task_mqtt_publish`, these variables are manually concatenated into a JSON string. Note that the raw sensor values and boolean states are exposed here in plaintext before encryption.

Listing 4.2: MCU 1 Data Packing and Encryption

```
void task_mqtt_publish(void *pv) {
  // ... Loop ...

  // 1. Construct Plaintext JSON
  String plain = "{";
  plain += "\"moist\":"       + String(moist_value) + ",";
  plain += "\"pump_on\":"     + String(pump_on ? "true" : "
     false") + ",";
  plain += "\"gas_raw\":"     + String(gas_value) + ",";
  plain += "\"gas_alert\":"   + String(gas_alert ? "true" : "
     false");
  plain += "}";

  // 2. Encrypt
  String cipher = aesEncryptToBase64(plain);

  // 3. Encapsulate for Transport
  // The 'dev' field allows the gateway to know which key/
     logic to apply
  String payload = "{";
  payload += "\"dev\":\"node1\",";
  payload += "\"cipher\":\"" + cipher + "\"";
  payload += "}";

  mqtt.publish(MQTT_TOPIC, payload.c_str());
}
```

4.3.3 Payload Packing: MCU 2 (Node 2)

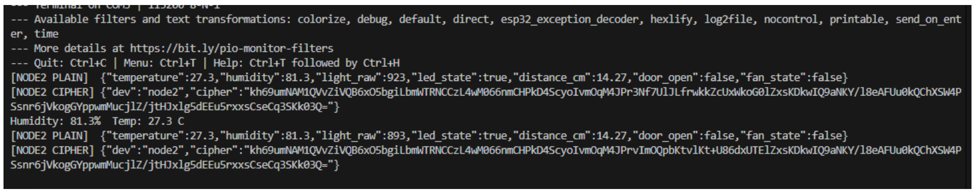
MCU 2 handles a larger dataset including temperature, humidity, lighting, and door status. The packing logic is similar but extends to include the DHT20 and Ultrasonic sensor readings.

Listing 4.3: MCU 2 Data Packing and Encryption

```
void task_mqtt_publish(void *pv) {
  // ... Loop ...

  // 1. Construct Plaintext JSON
  String plain = "{";
  plain += "\"temperature\":" + String(glob_temperature, 1) + ",";
  plain += "\"humidity\":"    + String(glob_humidity,    1) + ",";
  plain += "\"light_raw\":"   + String(glob_light_raw)    + ",";
  plain += "\"led_state\":"   + String(glob_led_state ? "true" : "false") + ",";
  plain += "\"distance_cm\":" + String(glob_distance, 2)  + ",";
  plain += "\"door_open\":"   + String(glob_door_open ? "true" : "false") + ",";
  plain += "\"fan_state\":"   + String(glob_fan_state ? "true" : "false");
  plain += "}";

  // 2. Encrypt
  String cipherB64 = aesEncryptToBase64(plain);

  // 3. Encapsulate for Transport
  String payload = "{";
  payload += "\"dev\":\"node2\",";
  payload += "\"cipher\":\"" + cipherB64 + "\"";
  payload += "}";

  mqtt.publish(MQTT_TOPIC_TELEMETRY, payload.c_str());
}
```

Analysis: This two-stage wrapping technique ensures that the broker only sees the device ID (`"dev": "node..."`), while the actual telemetry remains opaque (`"cipher": "..."`) until it reaches the authorized Python gateway.

Example: When received information from Node 1 and Node 2, it show this on the terminal



```
--- Terminal on COM3 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, esp32_exception_decoder, hexlify, log2file, nocontrol, printable, send_on_ent
er, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
[NODE2 PLAIN]  {"temperature":27.3,"humidity":81.3,"light_raw":923,"led_state":true,"distance_cm":14.27,"door_open":false,"fan_state":false}
[NODE2 CIPHER] {"dev":"node2","cipher":"kh69umNAM1QVvZiVQB6xO5bgiLbmWTRNCCzL4wM066nmCHPkD4ScyoIvmOqM4JPr3Nf7UlJLfrwkkZcUxWkoG0lZxsKDkwIQ9aNKY/l8eAFUu0kQChXSW4P
Ssnr6jVkogGYppwmMucjlZ/jtHJxlg5dEEu5rxxsCseCq3SKk03Q="}
Humidity: 81.3%  Temp: 27.3 C
[NODE2 PLAIN]  {"temperature":27.3,"humidity":81.3,"light_raw":893,"led_state":true,"distance_cm":14.27,"door_open":false,"fan_state":false}
[NODE2 CIPHER] {"dev":"node2","cipher":"kh69umNAM1QVvZiVQB6xO5bgiLbmWTRNCCzL4wM066nmCHPkD4ScyoIvmOqM4JPrvImOQpbKtvlKt+U86dxUTElZxsKDkwIQ9aNKY/l8eAFUu0kQChXSW4P
Ssnr6jVkogGYppwmMucjlZ/jtHJxlg5dEEu5rxxsCseCq3SKk03Q="}
```

Figure 4.1: Example terminal

5 MQTT Broker Configuration

5.1 Assigned Task

The MQTT Broker acts as the central communication hub (or "Fog Node") for the local IoT network. Its primary responsibilities are:

- **Decoupling:** Enable asynchronous communication between the Edge nodes (MCU 1, MCU 2) and the Gateway (Python Bridge).
- **Routing:** Efficiently route telemetry data from publishers to subscribers using a hierarchical topic structure.
- **Reliability:** Maintain persistent connections with the ESP32 devices to minimize latency during command execution.

5.2 Theoretical Framework

5.2.1 The Publish/Subscribe Model

Unlike the Request/Response model (HTTP) where the client waits for the server, MQTT uses a **Publish/Subscribe** pattern.

- **Broker:** The server (Eclipse Mosquitto) that filters messages. It is the only component that knows about all clients.
- **Publisher:** A device (e.g., MCU 1) that sends data to a specific topic (e.g., `home/nodes/node1`). It does not know who will receive it.
- **Subscriber:** A device (e.g., Python Bridge) that expresses interest in a topic. It receives all messages sent to that topic.

This architecture allows the Python Bridge to be restarted without crashing the MCUs, ensuring system resilience.

5.2.2 Quality of Service (QoS)

The system utilizes **QoS 0 (At most once)** for high-frequency sensor telemetry to reduce overhead, and **QoS 1 (At least once)** for critical control commands (like "Open Door") to ensure the action is performed.

5.3 Implementation Details

5.3.1 Broker Configuration

The broker is hosted on a local server (Raspberry Pi or PC) within the LAN.

- **Software:** Eclipse Mosquitto (Open Source MQTT Broker).
- **IP Address:** `192.168.120.250`.
- **Port:** 1883 (Unencrypted TCP/IP for local traffic).

5.3.2 Topic Topology Map

The following table defines the communication architecture established in the source code.

Table 5.1: System MQTT Topic Topology

Topic Path	Publisher	Subscriber	Function
`home/nodes/node1`	MCU 1	Python Bridge	Encrypted Soil/Gas Telemetry
`home/nodes/node2`	MCU 2	Python Bridge	Encrypted Temp/Door Telemetry
`home/pump/cmd`	Python Bridge	MCU 1	Water Pump Control (R PC)
`home/door/ai`	AI / Bridge	MCU 2	Face Recognition Trigger
`home/door/garage`	Python Bridge	MCU 2	Garage Door Override
`home/door/garden`	Python Bridge	MCU 2	Garden Gate Control
`home/fan/cmd`	Python Bridge	MCU 2	Fan Mode (ON/OFF/AUTO)
`home/pool_led/cmd`	Python Bridge	MCU 2	Pool LED Mode (ON/OFF/AUTO)

5.4 Instrution for MQTT Broker

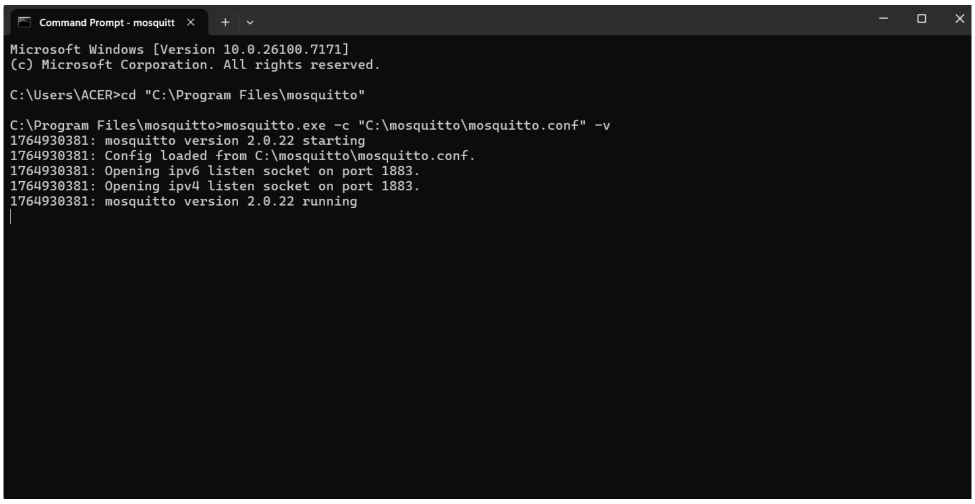
Instruction: find ip config then download mosquito to create mosquito.config file containing

```
listener 1883
allow_anonymous true
```

Then run this command on terminal to run MQTT Broker

```
cd "C:\Program Files\mosquitto"
mosquitto.exe -c "C:\mosquitto\mosquitto.conf" -v
```

If it looks like this, it is ready to receive information from MCU 1 and MCU 2



```
Microsoft Windows [Version 10.0.26100.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ACER>cd "C:\Program Files\mosquitto"

C:\Program Files\mosquitto>mosquitto.exe -c "C:\mosquitto\mosquitto.conf" -v
1764930381: mosquitto version 2.0.22 starting
1764930381: Config loaded from C:\mosquitto\mosquitto.conf.
1764930381: Opening ipv6 listen socket on port 1883.
1764930381: Opening ipv4 listen socket on port 1883.
1764930381: mosquitto version 2.0.22 running
```

Figure 5.1: Example terminal for encrypting

5.5 Integration Analysis

The role of the Broker is verified by examining the connection logic in the edge devices.

MCU Connection Logic: Both MCUs initiate a connection to the broker in their `setup()` function. If the connection fails, they enter a retry loop to prevent system hang.

Listing 5.1: MCU Reconnection Logic

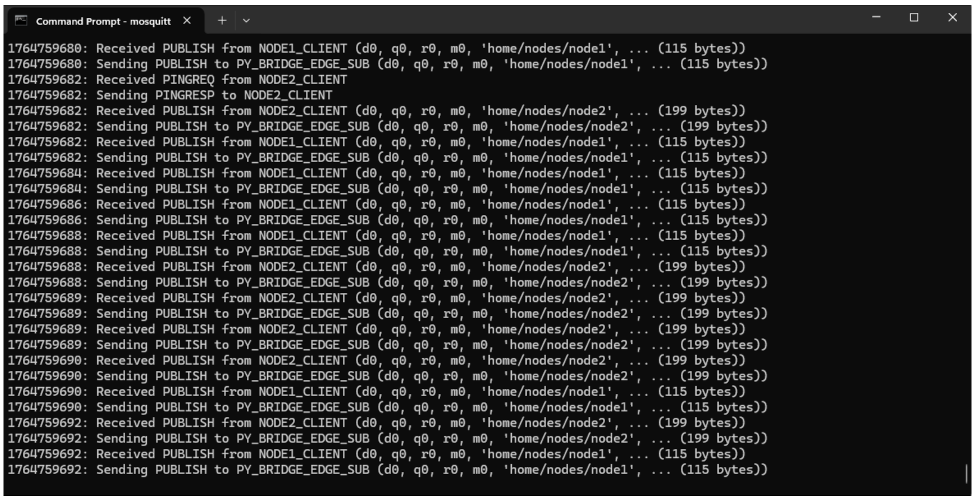
```
void mqttReconnect() {
  if (mqtt.connected()) return;
  Serial.print("MQTT connecting ... ");
  // Attempt to connect with Client ID
  if (mqtt.connect("NODE2_CLIENT")) {
    Serial.println("OK");
    // Resubscribe to command topics
    mqtt.subscribe("home/door/ai");
    // ... other subscriptions ...
  } else {
    Serial.printf("fail rc=%d\n", mqtt.state());
  }
}
```

Bridge Subscription Logic: The Python script uses a wildcard subscription (#) to capture data from all nodes simultaneously. This demonstrates the scalability of the broker architecture—adding "Node 3" would require no changes to the Python receiver code.

Listing 5.2: Python Wildcard Subscription

```python
def on_sub_connect(client, userdata, flags, rc):
    print("EDGE SUB connected to broker", EDGE_BROKER)
    # Subscribe to ALL node topics
    client.subscribe("home/nodes/#")
```

Example: When received information from Node 1 and Node 2, it show this on the terminal



```
1764759680: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759680: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759682: Received PINGREQ from NODE2_CLIENT
1764759682: Sending PINGRESP to NODE2_CLIENT
1764759682: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759682: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759682: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759682: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759684: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759684: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759686: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759686: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759688: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759688: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759688: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759688: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759689: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759689: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759689: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759689: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759690: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759690: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759690: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759690: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759692: Received PUBLISH from NODE2_CLIENT (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759692: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node2', ... (199 bytes))
1764759692: Received PUBLISH from NODE1_CLIENT (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
1764759692: Sending PUBLISH to PY_BRIDGE_EDGE_SUB (d0, q0, r0, m0, 'home/nodes/node1', ... (115 bytes))
```

Figure 5.2: Example terminal

6 Decryption Module Implementation

6.1 Assigned Task

The Decryption Module is a specialized software component running on the Gateway (Python Bridge). Its sole responsibility is to ensure that encrypted telemetry data arriving from the edge devices is securely unlocked before being processed or forwarded to the cloud. The specific requirements are:

- **Data Ingestion:** Accept Base64-encoded ciphertext strings from the MQTT subscriber.
- **Cryptographic Engine:** Implement the inverse of the MCU's encryption logic: Base64 Decoding → AES-128-ECB Decryption.
- **Data Sanitation:** Correctly remove PKCS#7 padding bytes to restore the original JSON structure.
- **Error Handling:** Gracefully handle invalid keys or corrupted payloads without crashing the gateway.

6.2 Theoretical Framework: The Decryption Pipeline

The decryption process is strictly linear and must be the exact mathematical inverse of the encryption steps performed on the ESP32.

6.2.1 Step 1: Base64 Decoding

The payload arrives as an ASCII string (e.g., `"yK9x..."`). The system must first convert this back into the raw binary byte array (ciphertext) that the AES algorithm expects.

6.2.2 Step 2: AES-128-ECB Decryption

Using the shared symmetric key (K), the algorithm processes the ciphertext in 16-byte blocks.

$$P_i = D_K(C_i) \tag{6.1}$$

Where C_i is the ciphertext block and P_i is the resulting plaintext block (still containing padding).

6.2.3 PKCS#7 Unpadding

The decrypted binary data will include padding bytes at the end. The algorithm must read the value of the very last byte (N) and remove exactly N bytes from the tail of the data. *Validation:* If the last byte value is 0 or greater than 16, the padding is invalid, indicating a wrong key or corrupted data.

6.3 Software Implementation (Python)

The implementation is contained within the `decrypt.py` module, utilizing the `PyCryptodome` library.

Listing 6.1: AES Decryption Logic in Python

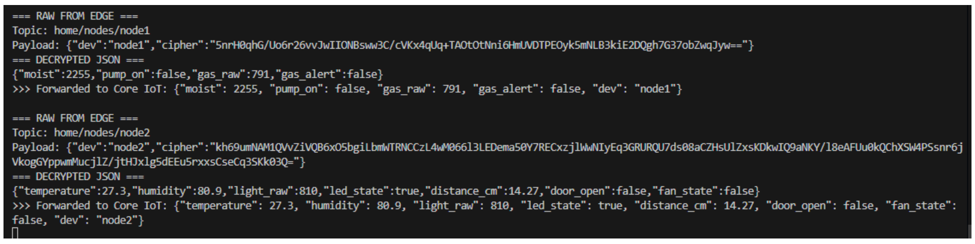
```python
from Crypto.Cipher import AES
import base64
import json

# Shared Symmetric Key (Must match MCU firmware)
AES_KEY = bytes([
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10
])

def aes_decrypt_from_base64(cipher_b64: str) -> str:
    """
    Input: Base64 Cipher String
    Output: Original JSON String
    """
    # 1. Base64 Decode
    ciphertext = base64.b64decode(cipher_b64)

    # 2. AES Decrypt (ECB Mode)
    cipher = AES.new(AES_KEY, AES.MODE_ECB)
    padded = cipher.decrypt(ciphertext)

    # 3. PKCS#7 Unpadding
    # Read the value of the last byte
    pad_val = padded[-1]

    # Validation check
    if pad_val <= 0 or pad_val > 16:
        raise ValueError("Invalid padding - Check AES Key")

    # Slicing: Remove the last 'pad_val' bytes
    plain_bytes = padded[:-pad_val]

    return plain_bytes.decode("utf-8")
```

6.4 Integration with MQTT Bridge

This module is imported directly into the main `mqtt_sub.py` script. When a message arrives on `home/nodes/#`, the bridge isolates the `"cipher"` field and passes it to this function.

- **Success Flow:** The function returns a JSON string, which is then parsed and forwarded to Core IoT.
- **Failure Flow:** If `ValueError` is raised (due to padding errors), the message is discarded, and an error is logged to the console, preserving system stability.

Example: When received information from Node 1 and Node 2, it show this on the terminal



```
=== RAW FROM EDGE ===
Topic: home/nodes/node1
Payload: {"dev":"node1","cipher":"5nrH0qhG/Uo6r26vvJwIIONBsww3C/cVKx4qUq+TAOtOtNni6HmUVDTPEOyk5mNLB3kiE2DQgh7G37obZwqJyw=="}
=== DECRYPTED JSON ===
{"moist":2255,"pump_on":false,"gas_raw":791,"gas_alert":false}
>>> Forwarded to Core IoT: {"moist": 2255, "pump_on": false, "gas_raw": 791, "gas_alert": false, "dev": "node1"}

=== RAW FROM EDGE ===
Topic: home/nodes/node2
Payload: {"dev":"node2","cipher":"kh69umNAM1QVvZiVQB6xO5bgiLbmWTRNCCzL4wM066l3LEDema50Y7RECxzjlWwNIyEq3GRURQU7ds08aCZHsUlZxsKDkwIQ9aNKY/l8eAFUu0kQChXSW4PSsnr6j
VkogGYppwmMucjlZ/jtHJxlg5dEEu5rxxsCseCq3SKk03Q="}
=== DECRYPTED JSON ===
{"temperature":27.3,"humidity":80.9,"light_raw":810,"led_state":true,"distance_cm":14.27,"door_open":false,"fan_state":false}
>>> Forwarded to Core IoT: {"temperature": 27.3, "humidity": 80.9, "light_raw": 810, "led_state": true, "distance_cm": 14.27, "door_open": false, "fan_state":
false, "dev": "node2"}
```

Figure 6.1: Example terminal for decrypting

7 Python Subscriber

7.1 Assigned Task

The Python Subscriber (`mqtt_sub.py`) functions as the intelligent "Gateway Bridge" for the system. Situated on a local server (e.g., Raspberry Pi or PC), it performs three critical functions that the microcontroller nodes cannot handle efficiently on their own:

- **Protocol Bridging:** It maintains simultaneous connections to the Local MQTT Broker (LAN) and the Core IoT Cloud (WAN).
- **Security Offloading:** It acts as the decryption engine. It receives encrypted payloads from the MCUs, decrypts them using the shared AES key, and forwards clean JSON to the cloud.
- **RPC Routing:** It listens for commands from the Cloud dashboard and routes them to the specific local topic required to trigger physical actuators (Doors, Fans, Pumps).

7.2 Theoretical Framework

7.2.1 The Gateway Design Pattern

In IoT architectures, a Gateway is essential for isolating the "Edge" from the "Cloud."

- **Isolation:** The MCUs (Node 1, Node 2) never connect directly to the internet. They only talk to the Local Broker. This reduces the attack surface.
- **Translation:** The Gateway translates the proprietary encrypted format of the local network into the standard JSON format expected by the Core IoT platform API.

7.2.2 Dual-Client Architecture

The script utilizes the `paho-mqtt` library to create two distinct MQTT client instances:

1. **sub_client:** Connects to `192.168.120.250` (Local). Subscribes to `home/nodes/#` to hear all devices.
2. **core_client:** Connects to `app.coreiot.io` (Cloud). Publishes to `v1/devices/me/telemetry`.

7.3 Software Implementation and Code Analysis

7.3.1 Initialization and Connection

The script initializes both clients. The `core_client` requires an Access Token (`CORE_TOKEN`) for authentication with the cloud platform.

Listing 7.1: Dual-Client Initialization

```python
# ====== BROKER CONFIG ======
EDGE_BROKER    = "192.168.120.250"
EDGE_TOPIC     = "home/nodes/#"

CORE_BROKER    = "app.coreiot.io"
CORE_TOKEN     = "A9o87CucucvVNeprWJ3h"    # Device Access
   Token

# Client 1: Listens to Local Nodes
sub_client = mqtt.Client(client_id="PY_BRIDGE_EDGE_SUB")

# Client 2: Talks to Cloud
core_client = mqtt.Client(client_id="PY_BRIDGE_CORE")
```

7.3.2 Uplink Logic: Decryption and Forwarding

The `on_sub_message` function handles the data flow from Edge → Cloud.

- It intercepts the message from the local broker.
- It extracts the `cipher` field.
- It calls the `aes_decrypt_from_base64` function (imported from `decrypt.py`).
- It forwards the decrypted JSON to the Core IoT telemetry topic.

Listing 7.2: Uplink: Decrypt and Forward

```python
def on_sub_message(client, userdata, msg):
    payload_str = msg.payload.decode("utf-8", errors="ignore"
       )
    data = json.loads(payload_str)

    # Check for encrypted field
    if "cipher" not in data:
        return

    try:
        # Decrypt using Shared Key
        plain = aes_decrypt_from_base64(data["cipher"])

        # Parse decrypted string to ensure valid JSON
        obj = json.loads(plain)

        # Add device ID for context
        dev = data.get("dev", "unknown")
        obj.setdefault("dev", dev)

        # Forward to Core IoT
        out = json.dumps(obj)
        core_client.publish("v1/devices/me/telemetry", out,
           qos=1)
        print(">>> Forwarded to Core IoT:", out)

    except Exception as e:
        print("Decryption Error:", e)
```

7.3.3 Downlink Logic: RPC Command Routing

The `on_core_message` function handles the flow from Cloud → Edge. The Core IoT platform sends RPC commands to the topic `v1/devices/me/rpc/request/+`. The script parses the `method` name and routes it to the appropriate local topic.

Listing 7.3: Downlink: RPC Routing Table

```python
def on_core_message(client, userdata, msg):
    data = json.loads(msg.payload.decode("utf-8"))
    method = data.get("method", "")
    params = data.get("params")

    # ==== ROUTING LOGIC ====

    # 1. Camera Control (To AI Module)
    if method == "CamOn":
        ctrl = {"camera_on": True}
        sub_client.publish("ai/camera_control", json.dumps(
            ctrl))

    # 2. Garage Door (To MCU 2)
    elif method == "garage_force":
        sub_client.publish("home/door/garage", "OPEN", qos=1)

    # 3. Main Door (To MCU 2 - Simulating AI unlock)
    elif method == "main_force":
        cmd = {"cmd": "OPEN_AI"}
        sub_client.publish("home/door/ai", json.dumps(cmd),
            qos=1)

    # 4. Fan Control (To MCU 2)
    elif method in ("fan_on", "fan_off", "fan_auto"):
        mode = "auto"
        if method == "fan_on": mode = "on"
        elif method == "fan_off": mode = "off"

        payload = {"mode": mode}
        sub_client.publish("home/fan/cmd", json.dumps(payload
            ))

    # 5. Pool Pump (To MCU 1)
    elif method == "pump":
        payload = {"pump": True}
        sub_client.publish("home/pump/cmd", json.dumps(
            payload))
```

8 Core IoT Platform Integration

8.1 Assigned Task

The final layer of the architecture is the Cloud Platform, implemented using **Core IoT** (based on ThingsBoard). This layer serves as the User Interface (UI) for the system owner. Its specific responsibilities are:

- **Data Visualization:** Render real-time telemetry from MCU 1 (Gas, Soil, Pump) and MCU 2 (Temp, Humidity, Light, Doors) into graphical widgets.
- **Remote Control (RPC):** Provide a control panel with buttons to trigger actions on the edge devices via the MQTT bridge.
- **State Management:** Maintain the "Digital Twin" state of the home (e.g., is the Fan currently ON or OFF?).

8.2 Theoretical Framework

8.2.1 Telemetry vs. RPC

The Core IoT platform communicates using two distinct data streams:

1. **Telemetry (Device $\rightarrow$ Cloud):** Time-series data points (e.g., Temperature = $27.3°C$). These are read-only values used for historical graphing and current status cards.
2. **Remote Procedure Calls (Cloud $\rightarrow$ Device):** JSON-RPC commands sent from the dashboard to the device. The server expects a response confirming the command execution. In our system, the Python Bridge intercepts these RPCs and translates them into local MQTT messages.

8.2.2 The "Digital Twin" Concept

The dashboard does not connect directly to the sensors. Instead, it interacts with a virtual representation (Digital Twin) of the home. When the Python Bridge publishes to `v1/devices/me/telemetry`, it updates the state of this twin. When a user clicks a button, they modify the desired state of the twin, which the platform propagates down to the physical device.

8.3 Dashboard Implementation

The Dashboard is divided into two logical sections: Environmental Monitoring and Actuator Control.

8.3.1 Environmental Monitoring Panel

As shown in Figure 8.1, the system visualizes high-priority environmental data using "Card" widgets with dynamic backgrounds.

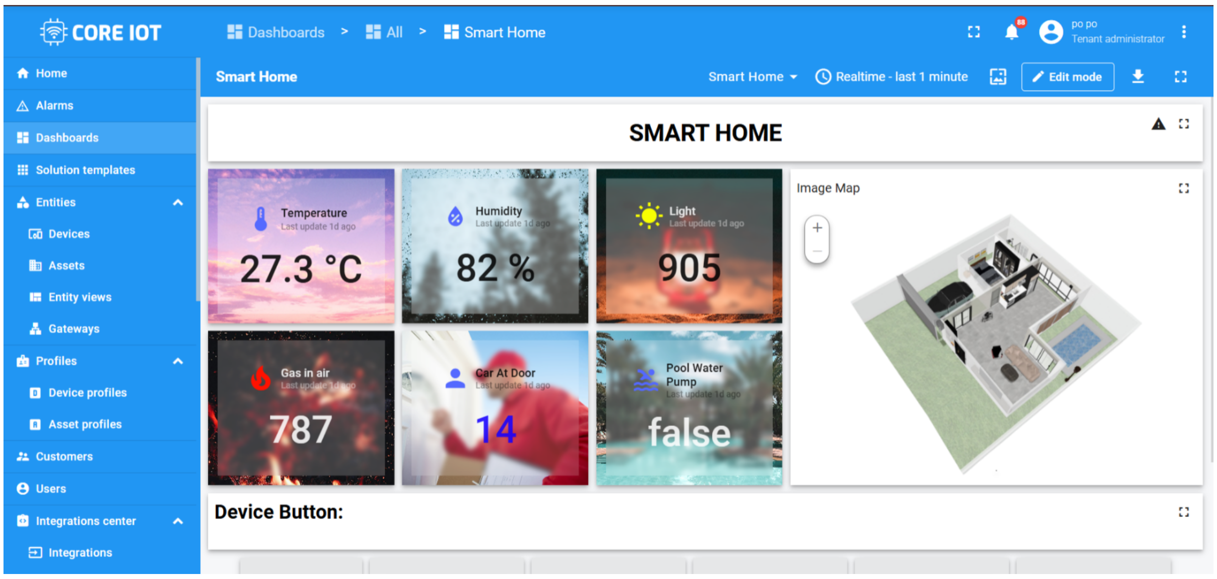


Figure 8.1: Core IoT Dashboard: Real-time Environmental Telemetry

- **Temperature & Humidity:** Displays 27.3°C and 82%. Sourced from the DHT20 sensor on MCU 2.
- **Light Intensity:** Displays raw ADC value (e.g., 905). This value drives the logic for the automated Pool LED system.
- **Gas in Air:** Displays the MQ-135 raw value (787). A high value here correlates with the activation of the exhaust fan.
- **Car At Door:** Displays the ultrasonic distance (e.g., 14 cm). This informs the user if a vehicle is present at the garage entrance.
- **Pool Water Pump:** Indicates the binary state (False/True) of the irrigation pump managed by MCU 1.

8.3.2 Actuator Control Panel

The bottom section of the dashboard (Figure 8.2) contains the Control Logic (RPC) implementation. These buttons trigger specific methods defined in the `mqtt_sub.py` bridge.

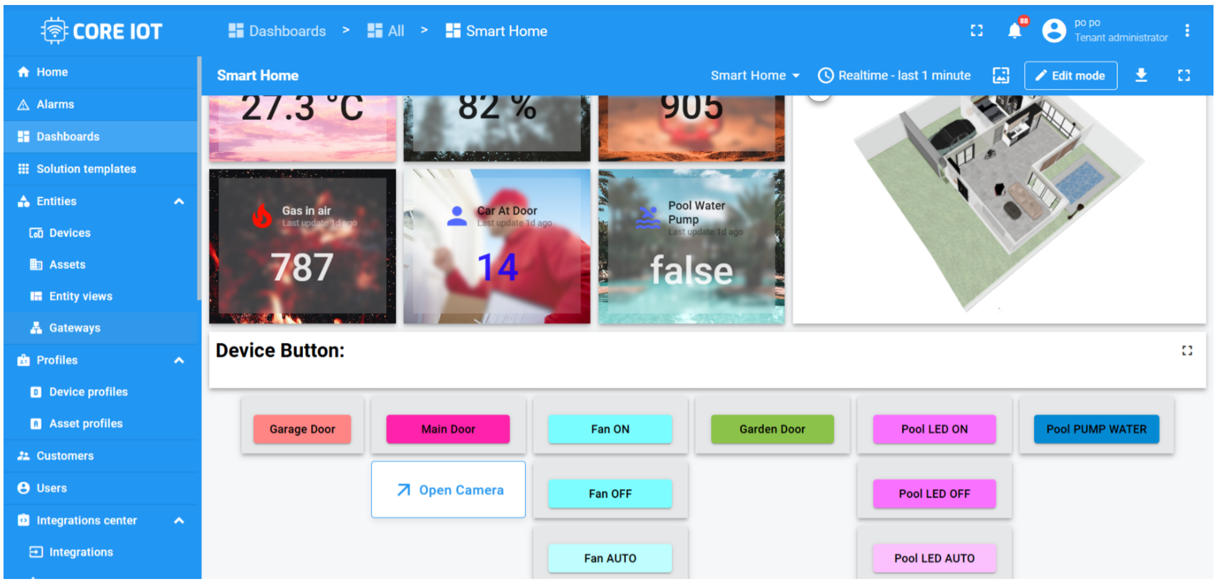


Figure 8.2: Core IoT Dashboard: RPC Control Panel

The mapping between the UI Buttons and the backend logic is detailed below:

Table 8.1: Dashboard RPC Button Mapping

Button Label	RPC Method Name	Action on Edge
`Garage Door`	`garage_force`	Publishes to `home/door/garage` (Open 10s)
`Main Door`	`main_force`	Publishes to `home/door/ai` (Simulate FaceID)
`Garden Door`	`garden_force`	Publishes to `home/door/garden` (Open 3s)
`Fan ON`	`fan_on`	Force Fan Relay HIGH
`Fan AUTO`	`fan_auto`	Revert to PIR Sensor Logic
`Pool LED ON`	`pool_led_on`	Force NeoPixels Orange
`Pool PUMP WATER`	`pump`	Toggles Water Pump (MCU 1)
`Open Camera`	`CamOn`	Triggers AI Camera Stream

8.4 Software Integration (Python Bridge)

The connection between this cloud dashboard and the local hardware is maintained by the `mqtt_sub.py` script. It listens to the topic `v1/devices/me/rpc/request/+` and routes the commands.

Listing 8.1: RPC Routing Logic in Python Bridge

```python
# mqtt_sub.py
def on_core_message(client, userdata, msg):
    # Decode payload
    data = json.loads(msg.payload.decode("utf-8"))
    method = data.get("method", "")

    # ==== ROUTING TABLE ====
    if method == "garage_force":
        # Forward to Local Broker
        sub_client.publish("home/door/garage", "OPEN", qos=1)
        print(">>> Garage force OPEN sent to EDGE")

    elif method == "pool_led_on":
        payload = {"mode": "on"}
        sub_client.publish("home/pool_led/cmd", json.dumps(
           payload))

    elif method == "pump":
        payload = {"pump": True}
        sub_client.publish("home/pump/cmd", json.dumps(
           payload))
```

9 Face Recognition and AI Integration

9.1 Assigned Task

To enhance the security of the smart home, a "Contactless Entry" system was developed using Computer Vision. Unlike traditional systems that require keys or keypads, this module allows the owner to unlock the main door simply by standing in front of the camera. The specific requirements were:

- **Dataset Creation:** Develop a tool to capture and standardize facial images of the homeowner.
- **One-Class Classification:** Train a Machine Learning model that learns *only* the owner's features and treats everyone else as an "anomaly" (Stranger).
- **Edge Integration:** The system must run locally on the gateway, triggered via MQTT, and capable of sending commands to the microcontroller nodes.

9.2 Theoretical Framework

9.2.1 Face Detection: Haar Cascades

Before recognition can occur, the system must locate the face. We utilize the **Haar Cascade Classifier** (Viola-Jones algorithm).

- It uses a sliding window approach with simple rectangular features (edges, lines).
- It is extremely fast and lightweight, making it suitable for real-time running on the CPU without requiring a GPU.
- In `AI.py`, the parameters `scaleFactor=1.1` and `minNeighbors=5` are tuned to balance between detection sensitivity and false positives.

9.2.2 Recognition Strategy: Isolation Forest

Instead of a traditional classifier (which requires data for "Owner" vs "Non-Owner"), we frame this as an **Anomaly Detection** problem.

- **Algorithm:** Isolation Forest.
- **Concept:** The model builds an ensemble of random decision trees. Anomalies (Strangers) are "easier to isolate" (require fewer splits to separate) than normal points (The Owner).

- **Training:** We feed the model *only* the owner's face vectors.
- **Inference:** The model outputs `1` for "Normal" (Owner) and `-1` for "Anomaly" (Stranger).

9.3 Software Implementation

The AI module is divided into three distinct scripts representing the machine learning pipeline.

9.3.1 Phase 1: Dataset Generation (facecap.py)

This script captures training data from the webcam. It standardizes inputs to ensure the model trains on consistent data.

- **Standardization:** Every detected face is cropped and resized to a fixed 64×64 pixel resolution.
- **Grayscale:** Images are converted to grayscale to reduce dimensionality (Color is rarely useful for facial geometry).

Listing 9.1: Data Capture and Preprocessing

```python
# facecap.py
    # ... Face Detection ...
    if len(faces) > 0:
        (x, y, w, h) = faces[0]
        # Expand box by 35% to include chin/hair
        pad = int(w * 0.35)
        # ... Coordinate clamping ...

        # Crop and Resize
        face_img = gray[y2:y3, x2:x3]
        face_img = cv2.resize(face_img, (64, 64))

        # Save to disk
        filename = os.path.join(SAVE_DIR, f"owner_{img_count
           :03d}.png")
        cv2.imwrite(filename, face_img)
```

9.3.2 Model Training (loadface.py)

This script loads the images, flattens them into vectors, and trains the Isolation Forest.

- **Vectorization:** A 64×64 image becomes a 1D vector of 4096 features ($64 \times 64 = 4096$).
- **Normalization:** Pixel values (0-255) are scaled to (0.0-1.0).
- **Hyperparameters:** `contamination=0.05` implies we expect very little noise in our training set (since it's all just the owner).

Listing 9.2: Training the Isolation Forest

```python
# loadface.py
# Flatten image to 1D vector
vec = img.flatten().astype("float32") / 255.0

# Train IsolationForest only on Owner's faces
model = IsolationForest(
    n_estimators=100,
    contamination=0.05,
    random_state=42
)
model.fit(X)
dump(model, "face_isoforest.joblib") # Save serialized model
```

9.3.3 Real-Time Inference Engine (AI.py)

This is the main runtime script. It operates as a State Machine controlled via MQTT.
State Logic:

1. **FLOATING:** Camera is OFF. Waiting for `ai/camera_control`.
2. **WAIT_FACE:** Camera ON. Detecting faces.
3. **CHECKING:** Face detected. Run prediction every `CHECK_INTERVAL` (1.0s).

Security Logic (Retry Limit): To prevent brute-force attacks (someone trying to trick the camera repeatedly), the system implements a counter.

- If `fail_attempts >= 10`, the system automatically shuts down the camera and resets.

Listing 9.3: Inference and Actuation Logic

```python
# AI.py
    if pred == 1:
        status_text = "DUNG - Chu nha"
        # 1. Send Unlock Command
        cmd = {"cmd": "OPEN_AI"}
        client.publish(DOOR_AI_TOPIC, json.dumps(cmd))

        # 2. Auto-shutdown after success
        camera_on = False
        current_status = "FLOATING"

    else:
        status_text = "SAI - Nguoi la"
        fail_attempts += 1

        # 3. Security Lockout
        if fail_attempts >= MAX_FAIL_ATTEMPTS:
            print("[AI] Too many attempts -> Camera OFF")
            camera_on = False
```

10 Conclusion and Future Development

10.1 Project Summary

This project successfully designed and implemented a secure, hybrid IoT architecture that bridges local edge computing with cloud-based management. By integrating two ESP32 microcontroller nodes with a Python-based Gateway and the Core IoT platform, the system achieved all primary functional requirements:

- **Environmental Safety:** MCU 1 successfully monitors gas levels and soil moisture, triggering the exhaust fan and water pump autonomously via FreeRTOS tasks.
- **Smart Access & Automation:** MCU 2 effectively manages complex actuation logic, including the AI-driven main door, ultrasonic garage door, and light-sensitive pool illumination.
- **End-to-End Security:** The implementation of AES-128-ECB encryption ensures that telemetry data transmitted over the local network remains opaque to unauthorized listeners. The Python Gateway successfully decrypts these payloads before forwarding them to the cloud.
- **AI Integration:** The Face Recognition module demonstrates that high-level biometric security can be offloaded to a local gateway, reducing latency by processing video streams locally rather than in the cloud.

10.2 System Evaluation

10.2.1 Performance and Reliability

The system demonstrates high reliability due to the "Edge-First" design philosophy.

- **Multitasking Efficiency:** The use of FreeRTOS allows MCU 2 to handle four distinct servo timers and sensor polling loops simultaneously without blocking, as evidenced by the dedicated tasks for `ultrasonic_task`, `garden_servo_task`, and `fan_control_task`.
- **Decoupled Architecture:** The separation of the Local Broker (`192.168.120.250`) from the Cloud Broker (`app.coreiot.io`) ensures that local automation (e.g., Garage Door auto-close) continues to function even if the internet connection is lost.

10.2.2 Security Analysis

While the current implementation significantly improves upon cleartext MQTT, specific constraints were identified:

- **Encryption Mode:** The system uses AES in ECB mode. While efficient for the limited processing power of the ESP32, identical plaintext blocks produce identical ciphertext, which could theoretically be exploited via pattern analysis.
- **Key Management:** The symmetric key is currently hardcoded in both the firmware and the Python script. In a production environment, this poses a risk if the device firmware is extracted.

10.3 Limitations

Despite the successful deployment, the prototype exhibits the following limitations:

- **Lighting Dependencies:** The Isolation Forest AI model, trained on grayscale 64×64 images, is sensitive to lighting changes. Significant shadows can lead to false negatives (refusing entry to the owner).
- **Configuration Rigidity:** Network credentials and Broker IPs are hardcoded in the C++ macros. Changing the network environment requires recompiling and flashing the devices.

10.4 Future Development

To transition this prototype into a market-ready smart home solution, the following enhancements are proposed:

1. **Dynamic Provisioning:** Implement a "Smart Config" or Bluetooth provisioning mode to allow users to set Wi-Fi and MQTT credentials via a mobile app without recompiling code.
2. **Enhanced Cryptography:** Upgrade from AES-ECB to AES-GCM (Galois/Counter Mode) to provide both data confidentiality and integrity authentication. Additionally, implement TLS/SSL for the MQTT connection itself.
3. **AI Optimization:** Replace the statistical Isolation Forest with a lightweight Convolutional Neural Network (CNN) or migrate the inference to a dedicated NPU

(Neural Processing Unit) to improve facial recognition accuracy in variable lighting conditions.

4. **OTA Updates:** Implement Over-The-Air (OTA) firmware updates to allow bug fixes and feature additions to be deployed to MCU 1 and MCU 2 remotely.

10.5 Concluding Remarks

The "Semeter 251" Assignment Report demonstrates a robust application of IoT principles, combining embedded software engineering, network security, and machine learning. The resulting system not only automates daily tasks but does so with a foundational respect for user data privacy and physical security.

11 Conclusion and Future Development

11.1 Project Summary

This project successfully designed and implemented a secure, hybrid IoT architecture that bridges local edge computing with cloud-based management. By integrating two ESP32 microcontroller nodes with a Python-based Gateway and the Core IoT platform, the system achieved all primary functional requirements:

- **Environmental Safety:** MCU 1 successfully monitors gas levels and soil moisture, triggering the exhaust fan and water pump autonomously via FreeRTOS tasks.
- **Smart Access & Automation:** MCU 2 effectively manages complex actuation logic, including the AI-driven main door, ultrasonic garage door, and light-sensitive pool illumination.
- **End-to-End Security:** The implementation of AES-128-ECB encryption ensures that telemetry data transmitted over the local network remains opaque to unauthorized listeners. The Python Gateway successfully decrypts these payloads before forwarding them to the cloud.
- **AI Integration:** The Face Recognition module demonstrates that high-level biometric security can be offloaded to a local gateway, reducing latency by processing video streams locally rather than in the cloud.

11.2 System Evaluation

11.2.1 Performance and Reliability

The system demonstrates high reliability due to the "Edge-First" design philosophy.

- **Multitasking Efficiency:** The use of FreeRTOS allows MCU 2 to handle four distinct servo timers and sensor polling loops simultaneously without blocking, as evidenced by the dedicated tasks for `ultrasonic_task`, `garden_servo_task`, and `fan_control_task`.
- **Decoupled Architecture:** The separation of the Local Broker (`192.168.120.250`) from the Cloud Broker (`app.coreiot.io`) ensures that local automation (e.g., Garage Door auto-close) continues to function even if the internet connection is lost.

11.2.2 Security Analysis

While the current implementation significantly improves upon cleartext MQTT, specific constraints were identified:

- **Encryption Mode:** The system uses AES in ECB mode. While efficient for the limited processing power of the ESP32, identical plaintext blocks produce identical ciphertext, which could theoretically be exploited via pattern analysis.
- **Key Management:** The symmetric key is currently hardcoded in both the firmware and the Python script. In a production environment, this poses a risk if the device firmware is extracted.

11.3 Concluding Remarks

The "Semeter 251" Assignment Report demonstrates a robust application of IoT principles, combining embedded software engineering, network security, and machine learning. The resulting system not only automates daily tasks but does so with a foundational respect for user data privacy and physical security.