The University of A Coruña

Faculty of Engineering

Department of Naval Architecture and Marine Engineering

Integrated IoT and Smart Contract Framework for Ship Stabilization and Real-Time Data Visualization

---

Course Title: Industry 4.0 Enabling Technologies

Submitted by:

Momen Masturi
Lemuel Hornsby
Ebrahim Tahmasebi Boldaji

Supervisor:

Prof. Paula Fraga Lamas
Prof. Tiago M. Fernández Caramés

Date of Submission:
May 30, 2025

Contents

1 Introduction

1.1 Overview of the Project

This project presents the design and implementation of an integrated system that combines mechanical stabilization, IoT-based data communication, and blockchain-enabled smart contracts for real-time monitoring and control. A physical stabilizer was developed and programmed using an Arduino microcontroller to react to ship movement. Sensor data was transmitted via the MQTT protocol to a cloud platform, where it was visualized and optionally recorded through a smart contract deployed on the Ethereum blockchain. The system is modular, scalable, and designed with real-world applications in maritime and industrial IoT environments.

1.2 Objectives

The main objectives of the project are:

- To design a functional **mechanical stabilizer** that can respond to motion or instability.
- To develop a **custom electronic circuit** for sensor-actuator integration using Arduino.
- To implement **real-time data transmission** using the MQTT protocol between the device and the cloud.
- To enable **data logging and transaction verification** through a smart contract on a blockchain network.
- To demonstrate the integration of hardware, cloud, and blockchain technologies in a single unified application.

This system has potential use cases in **ship stabilization**, **remote equipment monitoring**, **maritime automation**, and **secure industrial IoT** systems.

1.3 Technologies Used

The following tools, platforms, and technologies were used in the project:

- **CAD Software**: For 3D design of the stabilizer structure.
- **Fritzing**: For circuit layout and schematic design.
- **Arduino IDE**: For programming the microcontroller.
- **Arduino Uno and ESP8266 (Wemos D1 R1)**: As the core IoT device for connectivity.
- **GY-521 (MPU6050)**: Gyroscope/accelerometer module for motion detection.
- **MQTT Protocol**: Lightweight communication protocol for IoT data.

- **Mosquitto Broker**: Public MQTT broker (`test.mosquitto.org`) used for data exchange.
- **Remix IDE and Solidity**: For writing and deploying the smart contract on Ethereum.

2 Mechanical Design of the Stabilizer

2.1 Description of the Stabilizer Mechanism

The stabilizer mechanism is designed to maintain balance and reduce unwanted motion, especially roll, in a dynamic environment such as on a ship. The mechanism is actuated based on feedback from a gyroscope sensor, which detects angular motion. A servo motor then adjusts the stabilizer's position in real-time to counteract the detected movement. The stabilizer assembly includes a base platform, a moving arm, and a mounting point for the servo motor.

2.2 CAD Design Process

The 3D design of the stabilizer components was created using **Onshape**, a cloud-based computer-aided design (CAD) software. Onshape was chosen for its flexibility, collaborative features, and ease of access from any device without local installation. The modeling process included creating the base platform, arm segments, and servo mounting brackets. Each part was carefully dimensioned to fit standard servo motor sizes and to allow free movement during actuation. Figure 1 presents a 3D view of the base platform, illustrating the arrangement of all integrated electrical components.

2.3 Design Considerations

Several factors were taken into account during the design process:

- **Size**: Compact design to fit on a small testing platform and to minimize material use.
- **Mounting**: Holes and slots were included in the base for secure attachment to a surface or test rig.
- **Movement Range**: The arm was designed to move within a 0–180° range, matching the servo's rotation capabilities while ensuring stability during operation.

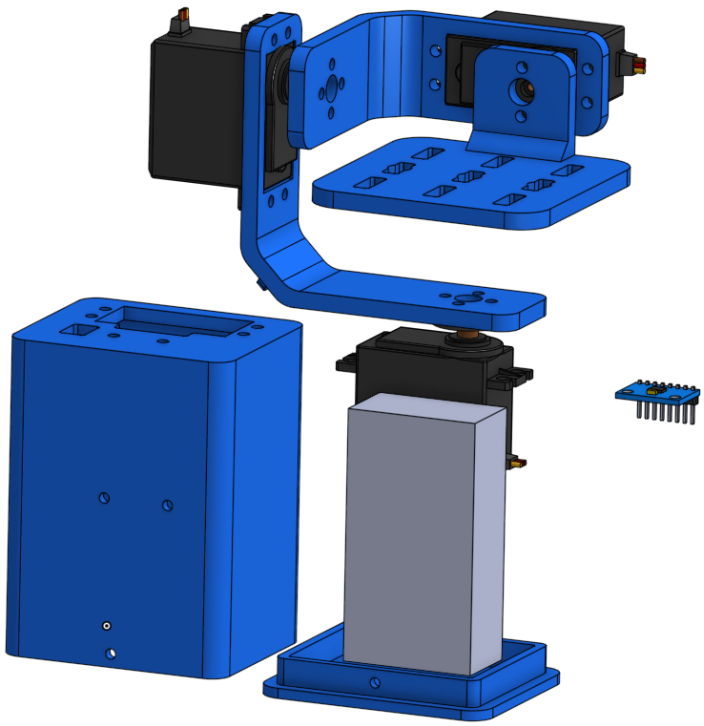


Figure 1: Base platform design of the stabilizer (created in Onshape)

3 Electronic Circuit Design

3.1 Circuit Layout and Components

The electronic system was designed to control the stabilizer mechanism and facilitate communication between sensors, actuators, and the microcontroller. The core of the circuit is the **Arduino Uno** and **Wemos D1 R1** board, based on the ESP8266 microcontroller, which provides built-in Wi-Fi capability. A **GY-521 (MPU6050)** sensor module is used to measure angular velocity and acceleration, while a **servo motor** is used as the actuator to adjust the stabilizer's position.

The following components were used in the circuit:

- **Wemos D1 Mini** – a compact microcontroller board based on the ESP8266, with built-in Wi-Fi support, ideal for IoT applications and wireless data transmission.
- **Arduino Uno** – a popular microcontroller board based on the ATmega328P, used for controlling and powering components in embedded systems. It does not include built-in Wi-Fi but can interface with other boards for wireless communication.
- **GY-521 (MPU6050)** – gyroscope and accelerometer sensor.
- **SG90 Servo Motor** – used for mechanical actuation.
- **Resistors and Capacitors** – for signal stability and protection.

- **External 5V Power Supply** – to ensure sufficient current for the servo motor.
- **Connecting Wires and Breadboard/PCB** – for circuit assembly.

3.2 Breadboard Layout Description

The breadboard layout was created using Fritzing to clearly illustrate the physical connections between all components in the system. The servo motors could not be reliably controlled using only the Wemos D1 Mini due to power limitations. Therefore, an Arduino Uno was used as a dedicated power source to provide stable and sufficient power, ensuring consistent operation of the stabilization platform.

The Wemos D1 Mini was also used to interface with the Arduino Uno via serial communication, reading sensor data which was then transmitted to the FlowFuse platform. The data was visualized in real time through a custom dashboard, enabling effective monitoring of the system's performance.

To enable obstacle detection across a range of angles, an ultrasonic sensor was mounted on a servo motor. However, rotating the sensor for scanning introduced delays in code execution. To mitigate this, an additional Wemos D1 Mini was used exclusively to control the servo motor for ultrasonic scanning. This separation of tasks improved system responsiveness and allowed smoother overall operation.

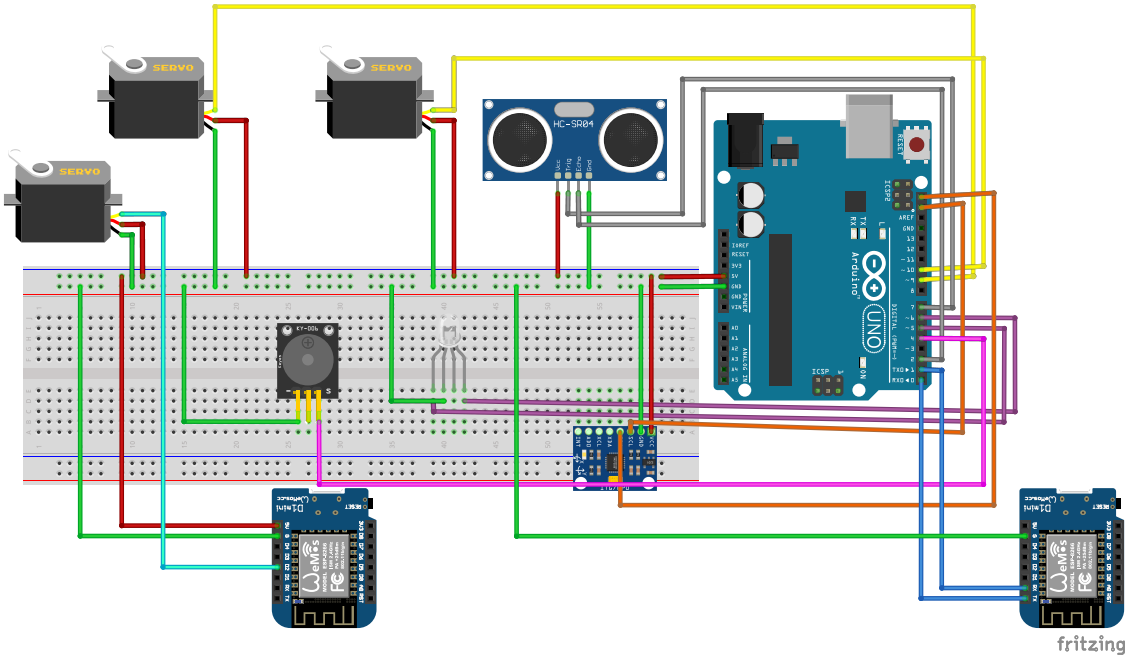


Figure 2: Breadboard Layout of the electronic circuit

3.3 Fritzing Schematic and PCB Design

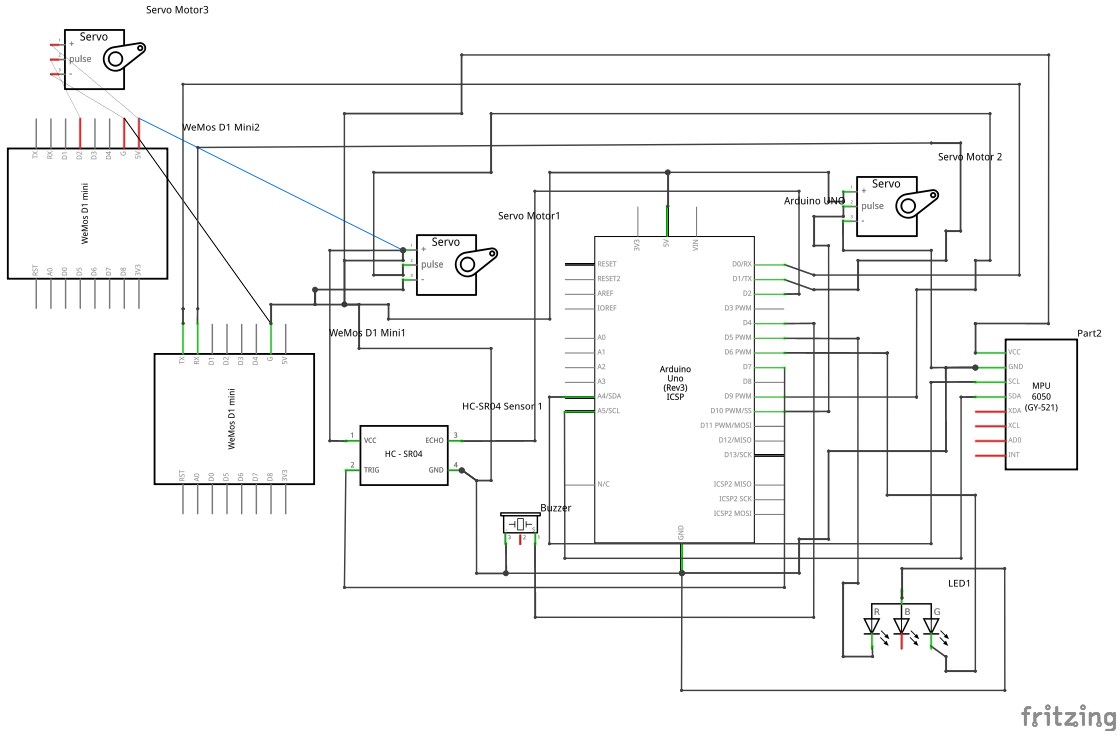


Figure 3: Fritzing schematic of the electronic circuit

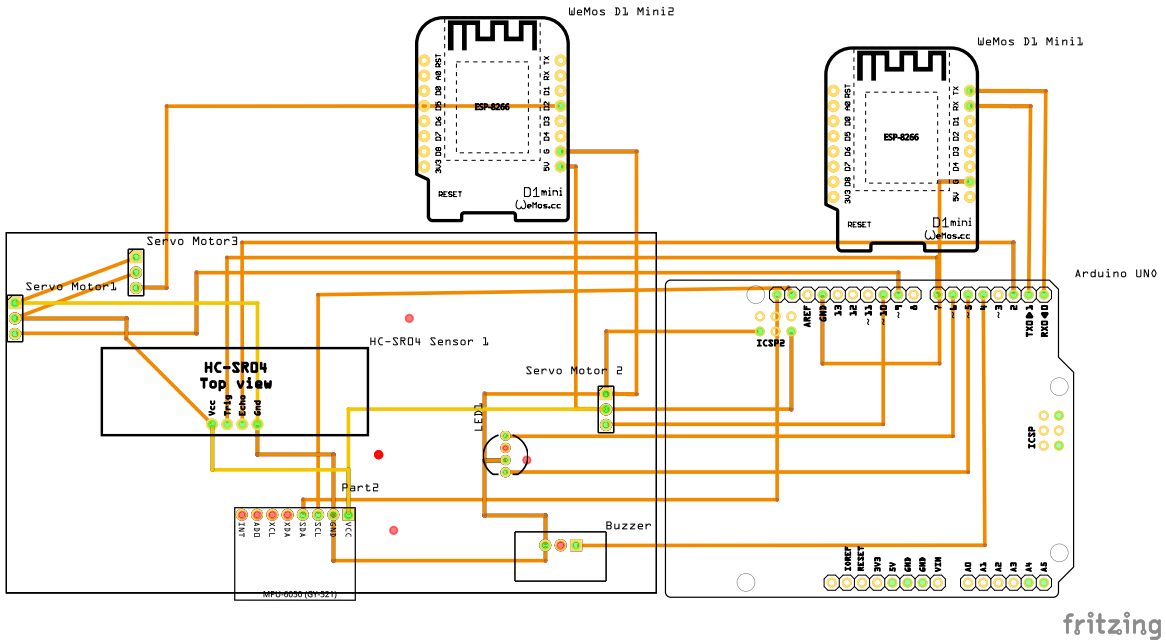


Figure 4: Custom PCB layout designed in Fritzing

4 Embedded Code Implementation

4.1 Arduino UNO Code: Sensor and Actuator Control

This section explains the embedded code running on the Arduino UNO. The UNO is responsible for acquiring motion data from an MPU6050 sensor, measuring distance using an ultrasonic sensor, controlling servo motors for stabilization, and triggering a buzzer for obstacle warnings. Data is sent to the Wemos D1 R1 over serial communication for cloud publishing.

4.1.1 Library Inclusions and Variable Declarations

```
#include <Wire.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Servo.h>

Adafruit_MPU6050 mpu;
const int trigPin = 2;
const int echoPin = 4;
const int buzzerPin = 7;

Servo rollServo;
Servo pitchServo;
```

The code includes libraries for I2C communication, sensor access, and servo control. Pins are defined for the ultrasonic sensor and buzzer. Two servo objects are declared to control pitch and roll axes.

4.1.2 Setup Routine

```
void setup() {
  Serial.begin(115200);
  while (!Serial) delay(10);

  Serial.println("Initializing MPU6050...");
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) delay(10);
  }

  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(buzzerPin, OUTPUT);

  mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
  mpu.setGyroRange(MPU6050_RANGE_500_DEG);
  mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);

  rollServo.attach(10);
  pitchServo.attach(9);

  rollServo.write(90);
  pitchServo.write(90);
```

```
24
25    delay(100);
26  }
```

This function initializes the serial monitor, sets up the MPU6050 sensor, and configures pins for the ultrasonic sensor and buzzer. It also attaches and centers the servo motors.

4.1.3 Sensor Data Acquisition and Processing

```
1  sensors_event_t a, g, temp;
2  mpu.getEvent(&a, &g, &temp);
3
4  float pitch = atan2(-a.acceleration.x,
5  sqrt(a.acceleration.y * a.acceleration.y + a.acceleration.z * a.
    acceleration.z)) * 180.0 / PI;
6
7  float roll = atan2(a.acceleration.y, a.acceleration.z) * 180.0 / PI;
```

The MPU6050 sensor provides acceleration and gyroscopic data. The pitch and roll angles are calculated using trigonometric formulas based on the raw acceleration values.

4.1.4 Servo Motor Control

```
1  float pitchServoAngle = map(pitch, 90, -90, 0, 180);
2  float rollServoAngle  = map(roll, 90, -90, 0, 180);
3
4  pitchServoAngle = constrain(pitchServoAngle, 0, 180);
5  rollServoAngle  = constrain(rollServoAngle, 0, 180);
6
7  pitchServo.write(pitchServoAngle);
8  rollServo.write(rollServoAngle);
```

The calculated pitch and roll angles are mapped to a servo-compatible range (0–180°) and constrained to ensure safety. The servo motors are then updated accordingly to stabilize motion.

4.1.5 Ultrasonic Distance Measurement

```
1  digitalWrite(trigPin, LOW);
2  delayMicroseconds(2);
3  digitalWrite(trigPin, HIGH);
4  delayMicroseconds(10);
5  digitalWrite(trigPin, LOW);
6
7  long duration = pulseIn(echoPin, HIGH);
8  float distance = duration * 0.034 / 2;
```

The ultrasonic sensor is triggered, and the echo duration is measured to calculate the distance of an obstacle in centimeters.

4.1.6 Serial Communication and Obstacle Alert

```
  Serial.print(pitch, 1);
  Serial.print(",");
  Serial.print(roll, 1);
  Serial.print(",");
  Serial.println(distance);

  if (distance < 100) {
    digitalWrite(buzzerPin, HIGH);
  } else {
    digitalWrite(buzzerPin, LOW);
  }
```

Pitch, roll, and distance data are printed as a comma-separated string to the serial interface, which is read by the Wemos D1 R1. A buzzer alert is triggered if an object is detected closer than 100 cm.

Complete Code Listing

The full code for the Arduino UNO is provided in Appendix A.

4.2 ESP8266 Communication Code: Wemos D1 R1

This section explains the code running on the Wemos D1 R1 (ESP8266), which acts as a Wi-Fi-enabled communication bridge between the Arduino UNO and the MQTT broker. Its main responsibilities include establishing a Wi-Fi connection, subscribing to the MQTT server, and publishing data received from the UNO via serial communication.

4.2.1 Library Inclusions and Configuration

```
  #include <ESP8266WiFi.h>
  #include <PubSubClient.h>

  const char* ssid = "Livebox6-B35F";
  const char* password = "4ntkh5hfdHPL";

  const char* mqtt_server = "broker.hivemq.com";
  const int mqtt_port = 1883;
  const char* mqtt_topic = "devices/mpu@001";
```

This part imports essential libraries for Wi-Fi and MQTT communication. The SSID and password are used to connect the Wemos board to a wireless network, while MQTT parameters configure the broker's address, port, and topic.

4.2.2 Wi-Fi Setup Function

```
  void setup_wifi() {
    delay(10);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
    }
    Serial.println("WiFi connected");
    Serial.println(WiFi.localIP());
```

```
}
```

This function handles Wi-Fi initialization. It begins by attempting to connect to the specified SSID and waits in a loop until the connection is successful. Once connected, it prints the local IP address to the serial monitor for debugging.

4.2.3 MQTT Connection Handling

```
void reconnect_mqtt() {
  while (!client.connected()) {
    if (client.connect(mqtt_topic, mqttUser, mqttPassword)) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      delay(5000);
    }
  }
}
```

The function 'reconnect mqtt()' ensures a persistent connection to the MQTT broker. If the connection drops, it retries every 5 seconds until reconnected.

4.2.4 Setup Function

```
void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
  Serial.println("ESP8266 ready to receive and publish pitch and roll
  ...");
}
```

The 'setup()' function initializes serial communication (used to receive data from the Arduino UNO), connects to Wi-Fi, and sets the MQTT server parameters.

4.2.5 Main Loop Logic

```
void loop() {
  if (!client.connected()) {
    reconnect_mqtt();
  }
  client.loop();

  if (Serial.available()) {
    String data = Serial.readStringUntil('\n');
    client.publish(mqtt_topic, data.c_str());
  }
}
```

The 'loop()' function continuously checks MQTT connectivity and reconnects if needed. When serial data is available (sent from the Arduino UNO), it is read, printed for debugging, and published to the MQTT topic for remote visualization (e.g., on a Node-RED dashboard).

Complete Code Listing

The full code for the Arduino UNO is provided in B.

5 Cloud Architecture of the IoT Stabilization System

Figure. 5 illustrates the end-to-end cloud architecture of the IoT stabilization system. The IoT Devices layer includes a microcontroller (Arduino Uno) and an ESP8266 Wi-Fi module. The Arduino handles sensor inputs (pitch, roll, and distance) and communicates the data via serial to the ESP8266. The ESP8266 connects to the cloud via Wi-Fi and publishes the data using the MQTT protocol to a public MQTT broker (HiveMQ). From there, the FlowFuse (Node-RED) client subscribes to the topic and parses the data for real-time visualization on a dashboard interface. This setup enables remote monitoring and enhances situational awareness through a structured, cloud-based pipeline.

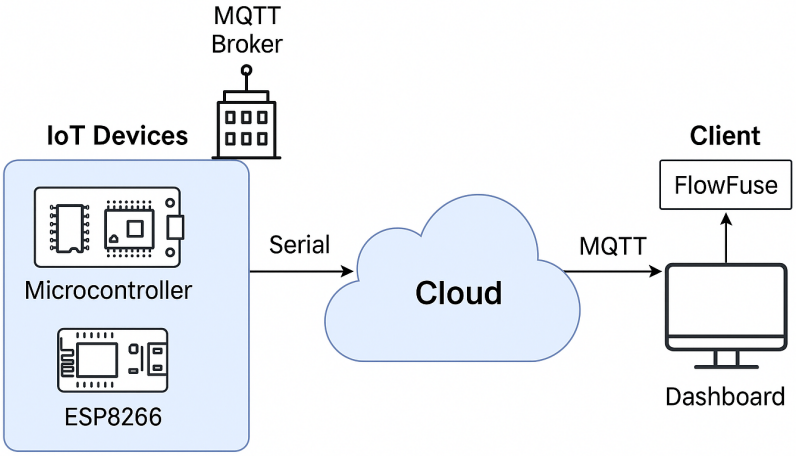


Figure 5: Cloud and network architecture

5.1 Overview of the System Architecture

The system is composed of two separate microcontrollers to balance computation and communication efficiency:

- Arduino Uno is responsible for acquiring motion data from the MPU6050 sensor and driving the servo motors. It was chosen for its stable power delivery and responsiveness in handling servo actuation.
- ESP8266 (Wemos D1 R1) acts purely as a communication module. It handles Wi-Fi connectivity and serves as a bridge between the Arduino Uno and the cloud.

Communication between the Arduino Uno and the ESP8266 is established using serial communication (UART). The ESP8266 reads pitch and roll data transmitted over the serial interface and publishes this data to an MQTT broker, making it accessible for real-time monitoring on a cloud dashboard.

5.2 Device-to-Cloud Communication Flow

The flow of data begins with the MPU6050 measuring pitch and roll angles. The ultrasound sensor is also incorporated to detect an object within 100cm range. These values are calculated and transmitted by the Arduino Uno via its TX pin to the RX pin of the ESP8266. The ESP8266 then:

- Connects to a Wi-Fi network
- Publishes the received pitch and roll data to the *broker.hivemq.com* MQTT broker
- Sends the data under a unique topic: *devices/mpu@001*

This message is then subscribed to by a **FlowFuse (Node-RED)** instance, which visualizes the values on a user dashboard in real time. Figure 6 depicts the connection scheme used to receive data from the Wemos D1 R1 board, perform debugging, and display the processed information on the dashboard.

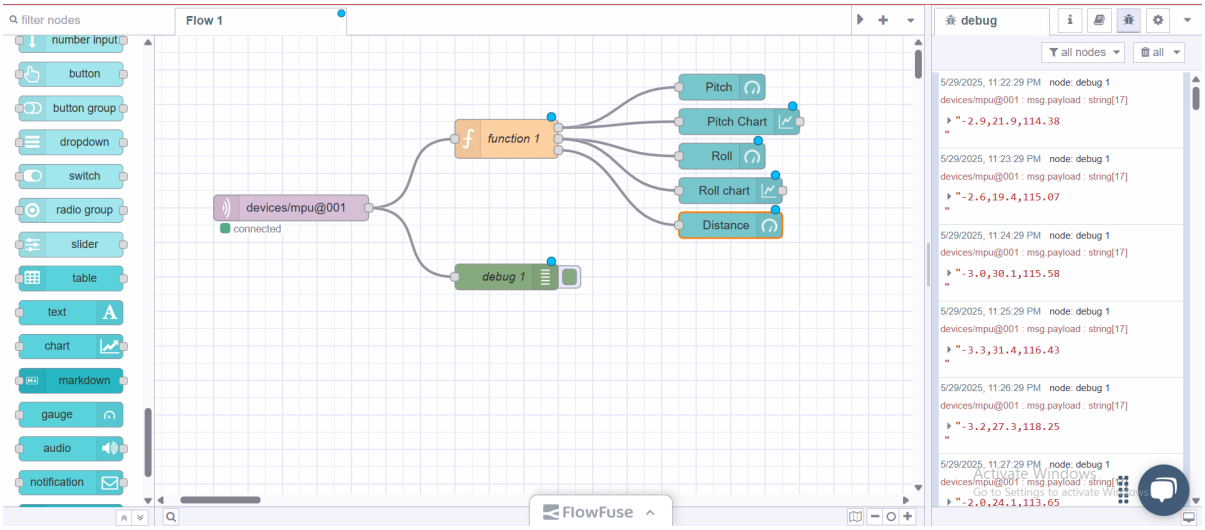


Figure 6: Connection setup for receiving, debugging and presenting data

5.3 MQTT-In and Function Node Customization

To visualize the pitch and roll data from the mpu and the distance from the ultrasound on the FlowFuse dashboard, the mqtt-in node was configured to subscribe to the custom topic: *devices/mpu@001*. Because of previous instability with *test.mosquitto.org*, the team switched to *broker.hivemq.com*, a free and stable alternative. A unique topic name was chosen to ensure consistent connectivity and prevent interference from other public users. Once the mqtt-in node receives the message (a CSV string like "2.13, -0.78, 54"), it is passed to a JavaScript function node that parses and separates the values into three distinct outputs — one for pitch, one for roll and one for distance respectively. This allows the dashboard to display each value in separate widgets in dashboard.

5.4 Debugging MQTT Payloads

During the testing phase, the debug node in FlowFuse was essential for verifying the structure and consistency of incoming MQTT messages. As shown in the Figure. 7,

the MQTT-in node successfully receives comma-separated string payloads representing pitch and roll angles. Each line in the debug console confirms that the payload is being published correctly from the ESP8266. However, minor inconsistencies (e.g., extra quotes or spacing) were observed in some payloads due to how the Arduino formatted the strings. These were handled in the JavaScript function node using .split(",") and parseFloat() to ensure reliable parsing. The corresponding code is provided in the C for reference This debugging step was crucial to confirming the system's end-to-end reliability from the sensor readings, through the serial link, to the MQTT message published and visualized in the dashboard. The debug node output, shown in the Figure. 7 below, confirms the new payload structure and demonstrates that values for pitch, roll, and distance are successfully received and parsed.

Figure 7: Debugging data before presenting them via dashboard

5.5 Dashboard Visualization

A web-based dashboard was developed using Node-RED (FlowFuse) to provide real-time visualization of sensor data transmitted via MQTT. The dashboard displays three key parameters using dynamic gauge widgets:

- Pitch
- Roll
- Distance to Obstacle

Each parameter is updated live as data is received from the ESP8266 microcontroller, allowing for intuitive monitoring of the system's orientation and proximity to nearby objects. The use of gauge-style displays provides a clear and immediate understanding

of values, making the system more user-friendly for operators who may need to assess stabilization performance at a glance. This visual feedback supports effective decision-making and enhances situational awareness during operation. Figure 8 illustrates the dashboard interface, showing the real-time pitch, roll, and distance readings via gauges.

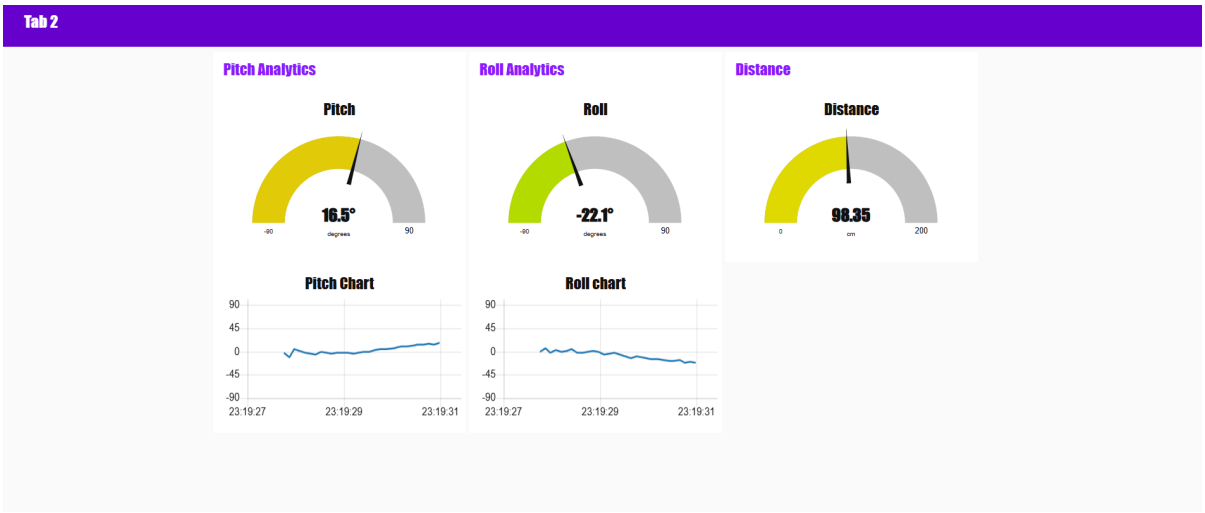


Figure 8: Data visualization via dashboard

6 Smart Contract Development

6.1 Motivation and Purpose

To enhance system accountability and operational control, a smart contract was developed to manage permissions, log missions, and enforce usage limits for the self-stabilizing platform. The contract is designed for use in mission-critical maritime applications such as missile launching or firefighting, where equipment must be tightly controlled and every action transparently recorded.

6.2 Development Environment and Tools

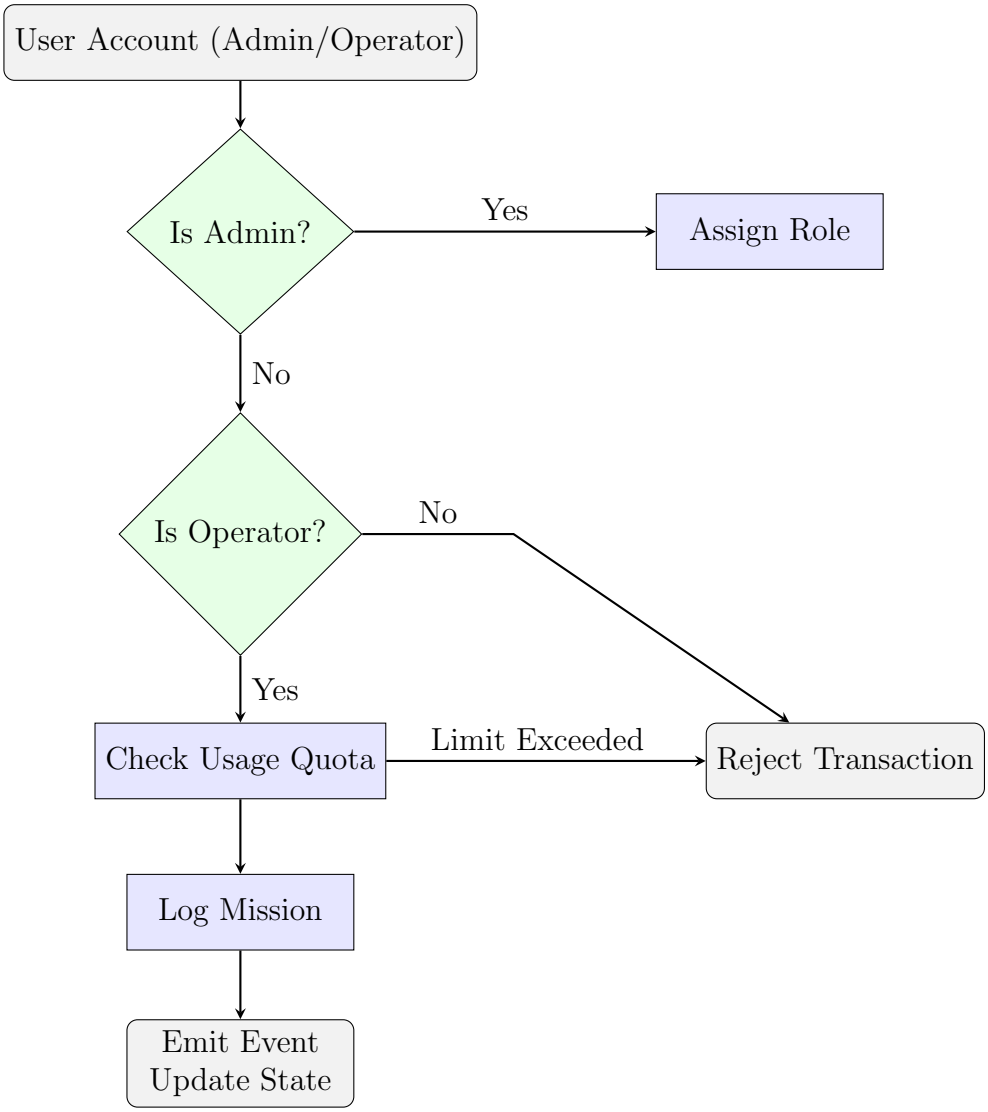
The contract was implemented in `Solidity` and tested using the `Remix IDE`, a browser-based Ethereum development environment. All transactions and logic execution were validated using the `Remix VM (Prague)` local test blockchain.

6.3 Design Approach

The smart contract `PlatformManager` was developed iteratively, with each layer of functionality building upon the previous one. It includes:

- **Role-based Access Control**
- **Mission Logging with Metadata**
- **Daily Usage Quota Enforcement**

Each of these stages is discussed with selected code examples below.



6.3.1 Role-Based Access Control

```
enum Role { None, Operator, Admin }
mapping(address => Role) public roles;

modifier onlyAdmin() {
  require(roles[msg.sender] == Role.Admin, "Not admin");
  _;
}
```

Listing 1: Role definition and storage

Access control is a fundamental concept in both traditional and blockchain-based systems. In this smart contract, we implement a simple yet effective access control mechanism using an enumeration (enum) and a Solidity feature called a modifier.

The enum named `Role` defines three possible user types: `None`, `Operator`, and `Admin`. By default, all users start with the `None` role, which means they are not authorized to perform sensitive actions.

We use a `mapping` to associate each Ethereum address (which identifies a user) with one of these roles. For example, the contract deployer is automatically assigned the `Admin` role. This admin can then assign the `Operator` role to other users via the `assignRole` function.

To enforce role-based permissions, we use a `modifier` called `onlyAdmin`. A modifier is a special construct in Solidity that can be attached to functions to restrict their usage. In this case, it ensures that only users with the `Admin` role can execute certain functions, such as assigning roles.

The role-based access control setup in this contract helps to prevent unauthorized access and maintain the integrity of the system. It ensures that only trusted users can perform key actions like logging missions or configuring user roles, which is critical in a safety-critical application like a maritime stabilizing platform.

```
function assignRole(address _user, Role _role) external onlyAdmin {
  roles[_user] = _role;
  emit RoleAssigned(_user, _role);
}
```

Listing 2: Assigning roles

6.3.2 Mission Logging and Event Emission

```
struct Mission {
  address operator;
  string deviceType;
  uint timestamp;
  string missionType;
  bool success;
}
Mission[] public missions;
```

Listing 3: Mission structure

Mission logging is the core functionality of this smart contract. It enables the system to keep an immutable record of every operation performed by an authorized user (Operator). Each mission is represented as a structured data type called a `struct`, which groups related information under one name. In our case, the `Mission` struct contains details such as the operator's address, the device type used (e.g., missile launcher or water jet), the mission type (e.g., defense or firefighting), the time it was logged, and whether it was successful.

These mission logs are stored in a dynamic array named `missions`, which grows automatically as new entries are added. Because this array is declared `public`, anyone can view the logged missions by querying the contract.

To facilitate real-time interaction with external systems (such as a dashboard or monitoring tool), the contract emits an event named `MissionLogged` every time a new mission is added. Events in Solidity are special constructs that allow contracts to communicate with the outside world. Once emitted, an event can be detected and processed by a front-end application or a blockchain explorer, creating a bridge between blockchain logic and user interfaces.

This mechanism provides a secure and transparent audit trail of all actions performed, making the contract suitable for critical applications where accountability and traceability are essential.

```
function logMission(
string memory _deviceType,
string memory _missionType,
bool _success
) external onlyOperator checkUsageLimit {
  missions.push(Mission(msg.sender, _deviceType, block.timestamp,
 _missionType, _success));
  emit MissionLogged(msg.sender, _deviceType, _missionType, _success);
}
```

Listing 4: Logging a mission

6.3.3 Usage Quota Enforcement

```
mapping(address => uint) public dailyUsage;
mapping(address => uint) public lastUsageDay;
```

Listing 5: Usage tracking state variables

In mission-critical systems—especially those involving defense or emergency response—it is important to prevent overuse or abuse of the platform. To achieve this, the smart contract enforces a daily usage quota. This quota limits how many times a single Operator can log a mission within a 24-hour period.

This feature is implemented using two `mapping` variables:

- `dailyUsage`: Keeps track of how many missions a specific address (user) has logged today.
- `lastUsageDay`: Records the last day (based on the current block timestamp) when the user logged a mission.

Before allowing a user to log a mission, the contract uses a `modifier` named `checkUsageLimit`. This modifier performs the following actions:

1. It checks if the current blockchain day (based on dividing the current timestamp by the number of seconds in a day) is greater than the stored `lastUsageDay`. If so, it resets the user's daily usage count to zero.
2. It then verifies that the user has not exceeded the limit (in this case, 5 missions per day).
3. If the user is still within the limit, the mission is logged and the counter is incremented.
4. If the user has exceeded the quota, the transaction is rejected with a descriptive error message.

This strategy ensures fair and regulated usage of the system, especially in collaborative or resource-limited environments. It also protects against accidental over-logging or potential misuse of operator access. The use of modifiers makes the logic reusable and keeps the contract code clean and readable.

```
modifier checkUsageLimit() {
  if (block.timestamp / 1 days > lastUsageDay[msg.sender]) {
    dailyUsage[msg.sender] = 0;
    lastUsageDay[msg.sender] = block.timestamp / 1 days;
  }
  require(dailyUsage[msg.sender] < 5, "Usage limit exceeded");
  _;
  dailyUsage[msg.sender]++;
}
```

Listing 6: Daily quota enforcement

6.4 Testing and Validation

1. Assigned the Operator role to a test account from the Admin account.
2. Logged five missions with valid parameters from the Operator account.
3. Verified that the sixth attempt was rejected due to quota limits.
4. Retrieved mission logs via `getMissions()` and confirmed correctness.

7 Lessons Learned

- Sensor Integration and Calibration: Successfully interfacing the MPU6050 sensor required understanding I2C communication and proper calibration to get meaningful pitch and roll angles.
- Data Filtering is Essential: Raw accelerometer data is noisy. Applying complementary filters and considering PID control greatly improved stability and responsiveness.
- PWM and Pin Limitations: Learned that not all Arduino pins support PWM, and careful selection is needed when working with servo motors.
- MQTT Communication: Established reliable real-time communication using MQTT and understood topic management and data formatting (e.g., float to string conversion).
- Toolchain Familiarity: Gained hands-on experience using both the Arduino IDE and external apps like IoT MQTT Panel for visualization.

8 Conclusion and Future Work

This project successfully demonstrated a basic 2-axis stabilization system using real-time sensor data from an MPU6050 and servo motor control on an Arduino platform. The system was enhanced with MQTT-based remote visualization, enabling the monitoring of pitch and roll on a mobile dashboard. Through iteration and debugging, stable sensor readings and corresponding servo responses were achieved, simulating a simple gimbal or balance platform. For future work the following items is offered:

- Implement Full 3D Stabilization: Extend the system to include yaw control, using a third servo.
- Kalman Filtering: Improve accuracy and responsiveness using Kalman filters instead of simple complementary filters.
- Wireless Feedback Loop: Allow commands from the MQTT dashboard to adjust PID parameters or control modes remotely.
- Use of IMU Libraries: Migrate to libraries like Madgwick or Mahony for better orientation estimation from the MPU6050.
- Real-World Application: Integrate with a physical model like a robotic arm or RC plane for practical testing of the stabilization logic.

9 Validation of Project

To support the validation process, video recordings were made at each stage of the project. The complete set of videos has been uploaded to a Google Drive link for reference.

link: `https://drive.google.com/drive/folders/1FSg8RiB5Dv0DWWHPhLmTaOawbKVdZOgc`

10 Appendix

A Arduino Uno Code

```
#include <Wire.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Servo.h>

Adafruit_MPU6050 mpu;

const int trigPin = 2;
const int echoPin = 4;
const int buzzerPin = 7;

Servo rollServo;
Servo pitchServo;

void setup() {
  Serial.begin(115200);
  while (!Serial) delay(10);

  Serial.println("Initializing MPU6050...");
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) delay(10);
  }
  Serial.println("MPU6050 Connected!");

  pinMode(trigPin, OUTPUT);
```

```
27    pinMode(echoPin, INPUT);
28    pinMode(buzzerPin, OUTPUT);
29
30    mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
31    mpu.setGyroRange(MPU6050_RANGE_500_DEG);
32    mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);
33
34    rollServo.attach(10);
35    pitchServo.attach(9);
36
37    rollServo.write(90);
38    pitchServo.write(90);
39
40    delay(100);
41  }
42
43  void loop() {
44    sensors_event_t a, g, temp;
45    mpu.getEvent(&a, &g, &temp);
46
47    float pitch = atan2(-a.acceleration.x, sqrt(a.acceleration.y * a.
     acceleration.y + a.acceleration.z * a.acceleration.z)) * 180.0 / PI;
48    float roll  = atan2(a.acceleration.y, a.acceleration.z) * 180.0 / PI
     ;
49
50    float pitchServoAngle = map(pitch, 90, -90, 0, 180);
51    float rollServoAngle  = map(roll, 90, -90, 0, 180);
52
53    pitchServoAngle = constrain(pitchServoAngle, 0, 180);
54    rollServoAngle  = constrain(rollServoAngle, 0, 180);
55
56    pitchServo.write(pitchServoAngle);
57    rollServo.write(rollServoAngle);
58
59    Serial.print(pitch, 1);
60    Serial.print(",");
61    Serial.print(roll, 1);
62
63    digitalWrite(trigPin, LOW);
64    delayMicroseconds(2);
65    digitalWrite(trigPin, HIGH);
66    delayMicroseconds(10);
67    digitalWrite(trigPin, LOW);
68
69    long duration = pulseIn(echoPin, HIGH);
70    float distance = duration * 0.034 / 2;
71
72    Serial.print(",");
73    Serial.println(distance);
74
75    if (distance < 100) {
76      digitalWrite(buzzerPin, HIGH);
77    } else {
78      digitalWrite(buzzerPin, LOW);
79    }
80  }
```

Listing 7: Full Arduino UNO Code

B Arduino Vemos mini R1 D1 code

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

// WiFi Configuration
const char* ssid = "Livebox6-B35F";
const char* password = "4ntkh5hfdHPL";

// MQTT Configuration
const char* mqtt_server = "broker.hivemq.com";  // e.g., broker.hivemq.com or IP
const int mqtt_port = 1883;
const char* mqtt_topic = "devices/mpu@001";          // Customize for your FlowFuse dashboard

const char* mqttUser = "";
const char* mqttPassword = "";

WiFiClient espClient;
PubSubClient client(espClient);

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect_mqtt() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(mqtt_topic, mqttUser, mqttPassword ))
    Serial.println("connected");
    else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" retrying in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
```

```
  Serial.begin(115200); // Used for Serial communication with Uno &
    debugging
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
  Serial.println("ESP8266 ready to receive and publish pitch and roll...
    ");
}

void loop() {
  if (!client.connected()) {
    reconnect_mqtt();
  }
  client.loop();

  if (Serial.available()) {
    String data = Serial.readStringUntil('\n');
    Serial.println("Received from Uno: " + data);

    // Publish to MQTT topic
    client.publish(mqtt_topic, data.c_str());
    Serial.println("Published to MQTT: " + data);
  }
}
```

Listing 8: Full Arduino Vemos mini R1 D1 Code

C Java script code for debugging and presenting data in dashboard

```
  let values = msg.payload.split(',');  // Split the string

  let pitch = parseFloat(values[0]);
  let roll = parseFloat(values[1]);
  let distance = parseFloat(values[2]);

  // Send 3 separate messages for each gauge
  return [
  { payload: pitch, topic: "pitch" },
  { payload: roll, topic: "roll" },
  { payload: distance, topic: "distance" }
  ];
```

Listing 9: Daily quota enforcement

D Solidity code for smart contract

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract PlatformManager {

```

```
// ROLES
enum Role { None, Operator, Admin }

// MAPPINGS
mapping(address => Role) public roles;
mapping(address => uint) public dailyUsage;
mapping(address => uint) public lastUsageDay;

// MISSION STRUCT
struct Mission {
  address operator;
  string deviceType;
  uint timestamp;
  string missionType;
  bool success;
}

Mission[] public missions;

// EVENTS
event RoleAssigned(address indexed user, Role role);
event MissionLogged(address indexed operator, string deviceType,
 string missionType, bool success);
event UsageLimitExceeded(address indexed user, uint timestamp);

// MODIFIERS
modifier onlyAdmin() {
  require(roles[msg.sender] == Role.Admin, "Not admin");
  _;
}

modifier onlyOperator() {
  require(roles[msg.sender] == Role.Operator, "Not operator");
  _;
}

modifier checkUsageLimit() {
  if (block.timestamp / 1 days > lastUsageDay[msg.sender]) {
    dailyUsage[msg.sender] = 0;
    lastUsageDay[msg.sender] = block.timestamp / 1 days;
  }
  require(dailyUsage[msg.sender] < 5, "Usage limit exceeded");
  _;
  dailyUsage[msg.sender]++;
}

// CONSTRUCTOR
constructor() {
  roles[msg.sender] = Role.Admin;
  emit RoleAssigned(msg.sender, Role.Admin);
}

// ROLE MANAGEMENT
function assignRole(address _user, Role _role) external onlyAdmin {
  roles[_user] = _role;
  emit RoleAssigned(_user, _role);
}
```

```
63
64    // LOG MISSION
65    function logMission(string memory _deviceType, string memory
       _missionType, bool _success)
66    external onlyOperator checkUsageLimit {
67      missions.push(Mission(msg.sender, _deviceType, block.timestamp,
       _missionType, _success));
68      emit MissionLogged(msg.sender, _deviceType, _missionType, _success);
69    }
70
71    // GET MISSIONS
72    function getMissionCount() external view returns (uint) {
73      return missions.length;
74    }
75
76    function getMissions() external view returns (Mission[] memory) {
77      return missions;
78    }
79  }
```

Listing 10: Daily quota enforcement