Swinburne University of Technology
COS80029 Technology Application Project
Project Repot

Project Code: MA4

Cost – Effective IOT-Based Fire and Gas Detection System

Team Members:
Vaishnavi Adhikari (104318778)
Ian Lau (104439984)
Rakesh Saka (104765251)
Jamal Yami (101013575)
Janith Samaraweera Kiribandara Appuhamillage Don (10485364)

Table of Contents

1. Abstract 3
2. Introduction 3
  2.1 Background 3
  2.2 Problem Statement 3
  2.3 Project Objectives 4
3. Literature Review 4
  3.1 Existing Solutions 4
  3.2 Gaps in Existing Systems 5
  3.3 Justification for the Proposed System 5
4. System Design 7
  4.1 Scope and Limitations 7
  4.2 Overview of the System Architecture 7
  4.3 System Flow Diagram 8
  4.4 Hardware Components 12
  4.5 Software Components 13
    4.5.1. ESP32 13
    4.5.2 Mobile App 13
  4.6 Cloud architecture 14
5. Implementation 14
  5.1 Hardware Setup 14
  5.2 Cloud integration 14
  5.3 Mobile App development 15
    5.3.1 App to IoT communication 15
    5.3.2 App to AWS communication 16
6. Testing and Results 17
  6.1 Testing Procedure 17
  6.2 Test Scenarios and Observations 18

6.2.1 WebSocket and Wi-Fi connection 18
6.2.2 Hardware & Sensor Functions 20
6.3 Performance Analysis 20
7. Challenges and Solutions 20
7.1 Hardware Issues 20
7.3 App Development Hurdles 21
8. Conclusion 21
8.1 Summary of Achievements 21
8.2 Lessons Learned 22
9. Future Work 22
10. References 23

1. Abstract

Gas leaks and fire accidents are serious safety risks, especially in places that cannot afford expensive safety systems. This project introduces a low-cost, IoT-based fire and gas detection system that can monitor and alert users in real-time. It uses an ESP32 microcontroller connected to gas, flame, and temperature sensors to detect hazards. The device gives alerts through a buzzer and LED and sends sensor data to a mobile app using AWS IoT Core and MQTT. The app allows users to connect the device to Wi-Fi and view live readings. Built with affordable components and simple software, this system is designed to improve safety in homes, especially in low-income or remote areas. The result is a working prototype that proves low-cost technology can help protect people and property effectively.

2. Introduction

2.1 Background

Gas leaks and hidden fire threats continue to pose significant safety risks in residential, industrial, and commercial settings. Although traditional detection devices are available, most current systems are either focused on a single event, do not support remote monitoring, or are too expensive. These shortcomings hinder emergency responses and elevate the chances of serious damage and loss.

This project seeks to overcome these shortcomings by creating an economical, Internet of Things (IoT)-based system for detecting gas and fire. Combined with commercially available microcontroller and sensors, the envisioned product comprehensively monitors multiple fire threats factors to safeguard users and the neighborhood life and property.
Furthermore, the system functions locally and remotely. Alerts are cascade to the user via the physical IoT device locally and remotely via a mobile app to help user to monitor anywhere.

2.2 Problem Statement

In environments where flammable gases and fire risks are significant, timely notification to users is crucial to avert loss of life, property destruction, and environmental damage.
Conventional detectors frequently detect a single factor that limits overall effectiveness. Furthermore, many detection systems operate as standalone units and do not have remote

connections. In critical moments where prompt warnings are crucial, the restriction of notifications to the local vicinity is a significant drawback. The market does have detection systems that allow remote connection or integration as part of the smart-home IoT ecosystem. However, those options are usually very expensive.
Safety is a basic need, not a luxury. Given the continuous urbanization around the globe, especially in the third world counties, there is a solid demand for an affordable, network-connected, and easily implementable solution that offers real-time monitoring and detection of residential fire risks.

2.3 Project Objectives

This project intends to tackle the stated problem by utilizing IoT technology to create a safety system that identifies gas leaks and fire threats in real-time, while smoothly integrating into mobile and cloud platforms. Cost-effective hardware is sourced to ensure user's affordability. Deliverables include an IoT device integrated with an IoT ecosystem and a mobile App to enable remote monitoring.

3. Literature Review

3.1 Existing Solutions

The analysis uses the Australian market as a reference to compare existing product prices and features. It should be noted that the product is intended to be a general-purpose, cost-effective solution for home safety, with affordability and accessibility as key goals.

In Australia, several home safety systems are available, but most are expensive or only cover one type of hazard, such as gas leaks or fire. For example, JC Instruments offers a gas detector for around AU$79.99, which can detect methane, butane, and LPG (JC Instruments, 2025). Another brand, Discount Instruments, sells a smart gas detector for about AU$84.11, which connects to the Tuya mobile app via Wi-Fi. However, these devices focus only on gas detection and do not include flame or temperature monitoring.

For flame detection, one of the few options available is *The Flame Detector™* by Forge Technologies, which uses UV light and costs around AU$262. This price is not affordable for many families. Visual-based detection through security cameras like Arlo is also expensive, starting at AU$219, and not designed for hazard alerts.

For temperature monitoring, most devices in the market are smart home products like the Apple HomePod Mini (AU$149) and Eve Room (AU$183.25), which offer air quality features but are not intended for safety alerts. More affordable options like Aqara sensors (AU$39) exist but do not combine with gas or flame detection.

Integrated systems that combine gas, flame, and temperature detection are rare and costly. For example, Nest Protect and Owl Wired offer smart smoke and CO alarms but do not support detection for methane, LPG, or flame. This shows a clear gap in the market for affordable, all-in-one safety solutions that cover multiple hazards at once.

3.2 Gaps in Existing Systems

Most home safety devices in the Australian market are either expensive, single-purpose, or lack smart integration. For example, many gas detectors can detect only one or two gases but do not include flame or temperature sensors (SunFounder, 2024). Flame detectors are usually built for industrial use and cost over AU$200, making them impractical for families with limited budgets. There is a major gap in the availability of multi-sensor systems that are both affordable and designed for home use.

Another issue is that existing systems often require multiple separate devices to cover different hazards. This increases the total cost and setup difficulty. Some detectors do not support mobile notifications or rely on premium platforms like Apple HomeKit or Nest, which are not suitable for low-income users (Horne, 2024).

There is also a lack of systems that offer customized mobile alerts connected through the cloud. Most solutions do not provide real-time sensor feedback directly to a user's phone. Finally, many existing systems are not designed for easy setup by non-technical users.

These gaps show the need for a low-cost, all-in-one IoT system that detects gas, flame, and temperature, with real-time mobile alerts through the cloud, and an easy-to-use app. Our system fills this gap using ESP32 sensors, cloud messaging via AWS IoT Core, and a custom mobile app built specifically for this project.

3.3 Justification for the Proposed System

The proposed system is designed to solve key issues found in existing safety devices. It combines gas, flame, and temperature detection into one affordable unit. It uses low-cost components like the MQ-2 gas sensor, IR flame sensor, and DHT11 temperature sensor, all connected to the ESP32 microcontroller. These parts are commonly used in low-budget IoT systems and have been proven reliable in similar applications (Ravina et al., 2020). This combination removes the need for separate devices, reducing the cost and complexity for end users.

The chosen temperature sensor is DHT11, which detects temperatures up to 50 degrees Celsius. While the DHT22 is a more advanced option with a range up to 80 degrees, it is not included in the starter kit and is more expensive. Since the project focuses on demonstrating a working IoT architecture using cost-effective tools, the DHT11 was selected as a practical trade-off. It still provides the necessary temperature monitoring to support early fire detection.

For flame detection, the system uses an IR-based sensor. While UV-based detectors are available in the market, they are known for generating false alarms and are significantly more expensive (Prodetec, 2020). The IR flame sensor used in this project detects up to 60 degrees and offers sufficient accuracy for household-level safety detection.

The MQ-2 sensor supports the detection of multiple gas types such as methane, butane, LPG, and smoke. It is capable of sensing concentrations between 200 – 10,000 ppm, while typical gas presence is detected above 400 ppm (Last Minute Engineers, n.d.). This makes it a versatile and reliable option for general gas leak detection.

The system also delivers real-time alerts through a custom-built mobile app connected to AWS IoT Core. When danger is detected, users receive instant notifications on their phones, while LEDs and buzzers provide physical alerts on-site. This setup ensures safety monitoring even when the user is not physically at home. Leveraging AWS also allows for future scalability and upgrades (Amazon Web Services, 2024).

Overall, this system fills a clear market gap. While the Australian market was used for pricing and feature comparisons, the actual goal was to build a functional, affordable, and user-friendly safety system. The integration of cloud alerts, a custom app, Wi-Fi connectivity, and reliable sensors makes this solution suitable for practical home safety applications.

4. System Design

4.1 Scope and Limitations

The sensors used are commercially available from online vendors. The team has sought vendors with high ratings but does not seek to calibrate the hardware performance nor design and manufacture new sensors or parts for the product.

- Hardware used:
    - ESP32 development board
    - MQ-2 for gas and smoke
    - DHT11 for temperature and humidity
    - IR flame sensors
    - OLED display
    - LED and buzzer

The team focused on developing programs that enable fire detection and data transmission. Further effort in product design and packaging is required to manufacture a marketable product.

Remote data transfer is provided by SaaS products or public cloud services. The team does not create a tailored server for the client. Long-term data storage and device management are also not a requirement in the project.

- Platform used:
    - Blynk IoT cloud (The use of SaaS is signed off in the Software Requirements Specification. Upon completion, the client requested to develop another custom app for his product.)
    - Amazon Web Services IoT Core services.

The custom App aims to demonstrate the core function of remote monitoring. It is built on Android Software development kit. Further effort in user registration and management, and security are required to host the App in production.

4.2 Overview of the System Architecture

The IoT detects readings including temperature, flame, and hazardous gas. It has an OLED screen to display the measurement, and an LED buzzer to alert risks when detected.

Furthermore, it is connected to the internet and actively publishes reading to the AWS IoT cloud.

The App serves two purposes. First, it communicates with the IoT and provision Wi-Fi connectivity. Second, it reads the sensor data sent to the cloud and renders it on the App UI in real time.

These functions are powered by three hardware and software components. Figure XXX shows the high-level system architecture. The ESP32 manages the sensors functions and publishes data to the AWS IoT using the MQTT protocol. The AWS IoT is the middleware for receiving and cascading MQTT messages in the cloud. The App uses React Native WebSocket networking support to communicate with the ESP32. The App also utilizes AWS Amplify service to fetch the MQTT messages.

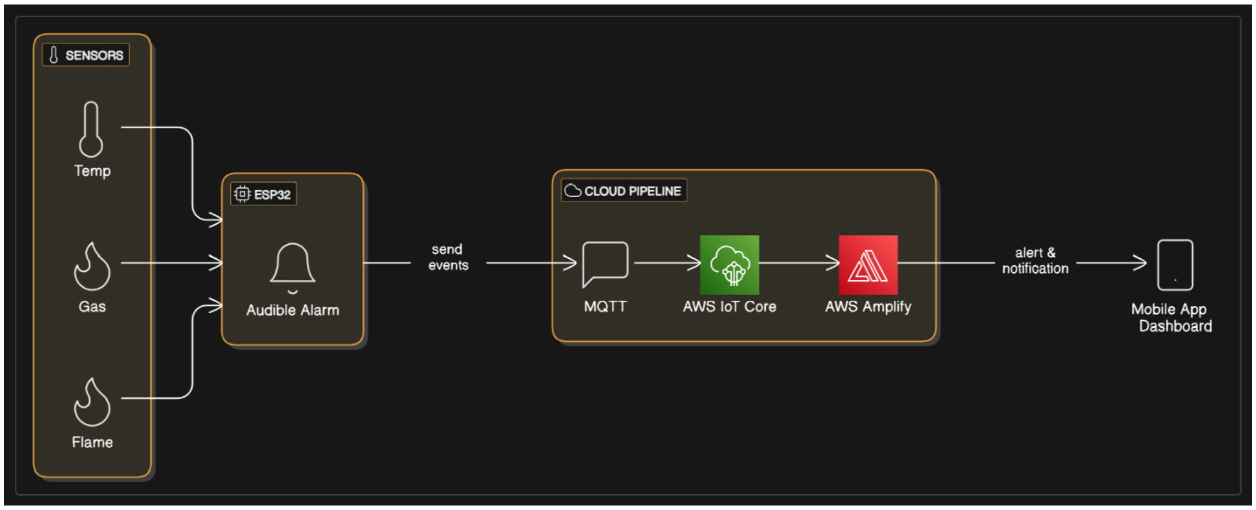


Figure 1. The product is built on ESP32, React Native and AWS.

4.3 System Flow Diagram

The product requires two simple steps to activate. Figure XXX further describe the comprehensive frontend and backed process flow.

The product function begins when the user powers the IoT device. Once turned on, the ESP32 microcontroller activates Access Point (AP) mode, allowing connection from the mobile app. The user then sends Wi-Fi credentials to the device via WebSocket server. Upon receiving the credentials, the ESP32 attempts to connect to the local Wi-Fi network. Once connected, the device proceeds to establish a connection with the AWS server. If the connection fails, the user can power off the device and then on again to retry.

Once the ESP32 connects to the AWS server, it transmits the temperature, gas, and flame sensors reading to the AWS server. Simultaneously, the mobile app retrieves the data from the server and displays it on a real-time dashboard. This dashboard shows the sensor readings, connection status, and any active alerts.

The system continuously monitors the sensor values. If any of the values exceed predefined safety thresholds, an alert is triggered. This alert is displayed within the app and on the IoT's buzzer and LED. Once the setup is complete and live monitoring is active, the system is fully operational and ready to ensure real-time safety monitoring.

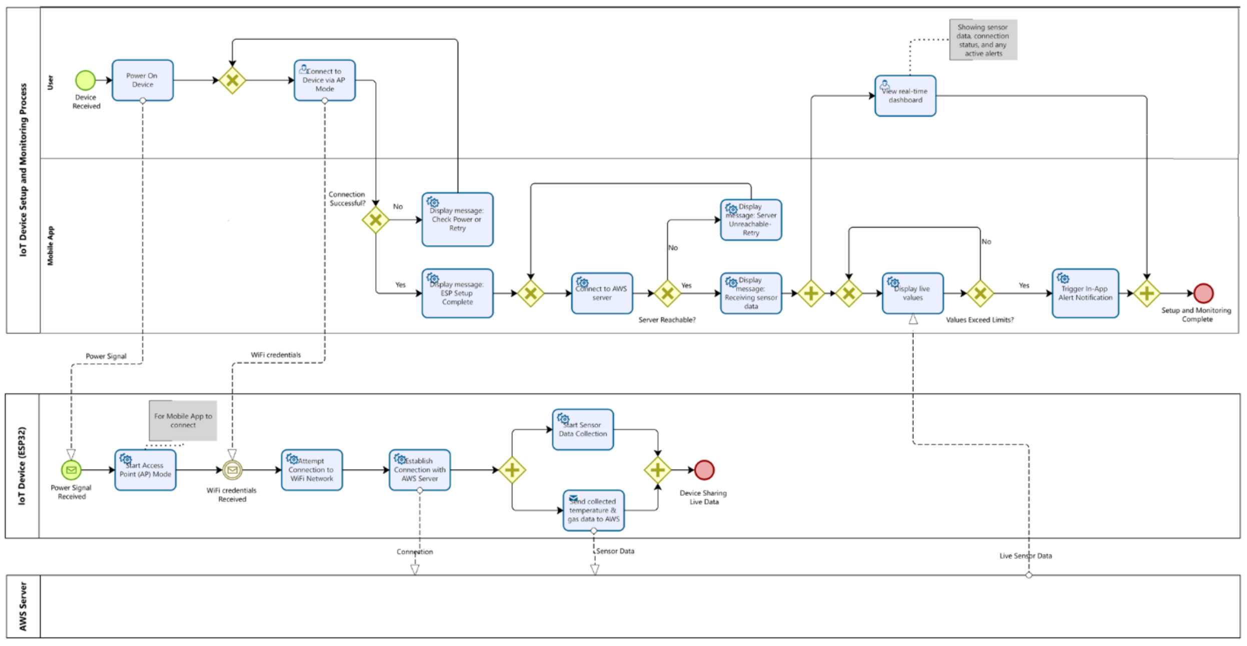

IoT Device Setup and Monitoring Process
User
Mobile App
Device Received
Power On Device
Connect to Device via AP Mode
Connection Successful?
No
Yes
Display message: Check Power or Retry
Display message: ESP Setup Complete
Connect to AWS server
Server Reachable?
No
Yes
Display message: Server Unreachable- Retry
Display message: Receiving sensor data
View real-time dashboard
Showing sensor data, connection status, and any active alerts
Display live values
Values Exceed Limits?
No
Yes
Trigger In-App Alert Notification
Setup and Monitoring Complete
Power Signal
WiFi credentials
IoT Device (ESP32)
For Mobile App to connect
Power Signal Received
Start Access Point (AP) Mode
WiFi credentials Received
Attempt Connection to WiFi Network
Establish Connection with AWS Server
Start Sensor Data Collection
Send collected temperature & gas data to AWS
Device Sharing Live Data
Connection
Sensor Data
Live Sensor Data
AWS Server

Figure 2. The process flow diagram demonstrates the end-to-end process of setting up the product.

Below describes the flow from the user's perspective.

1. Power up the IoT device to start the device in Access Point mode. It is ready to be connected.

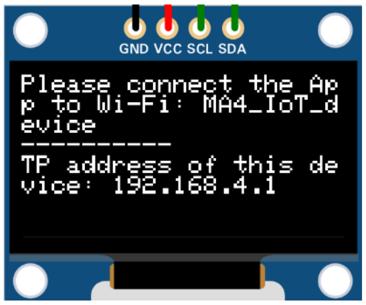


Figure 3. OLED message

2. Via the App, the user connects to the IoT Access Point IP address and sends Wi-Fi credentials to the IoT.

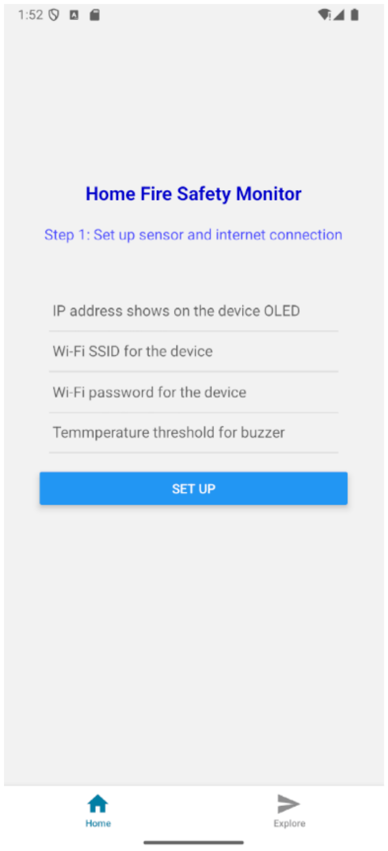

1:52
Home Fire Safety Monitor
Step 1: Set up sensor and internet connection
IP address shows on the device OLED
Wi-Fi SSID for the device
Wi-Fi password for the device
Temmperature threshold for buzzer
SET UP
Home
Explore

Figure 4. Setup Screen

2.1. Once the IoT WebSocket server receives a connection, Wi-Fi mode is triggered using the credentials received.

2.2. Once the internet is connected, the IoT starts publishing MQTT messages to AWS.

3. Via the app, the user links to the IoT using the AWS IoT “Thing” ID to monitor sensor readings remotely.

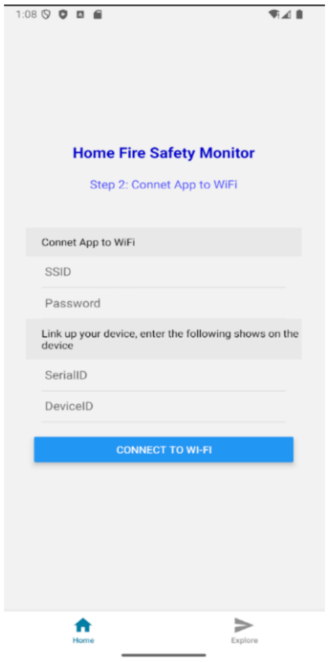


Figure 5, WiFi Setup

4. The App is actively receiving the IoT sensor readings and alert notifications.

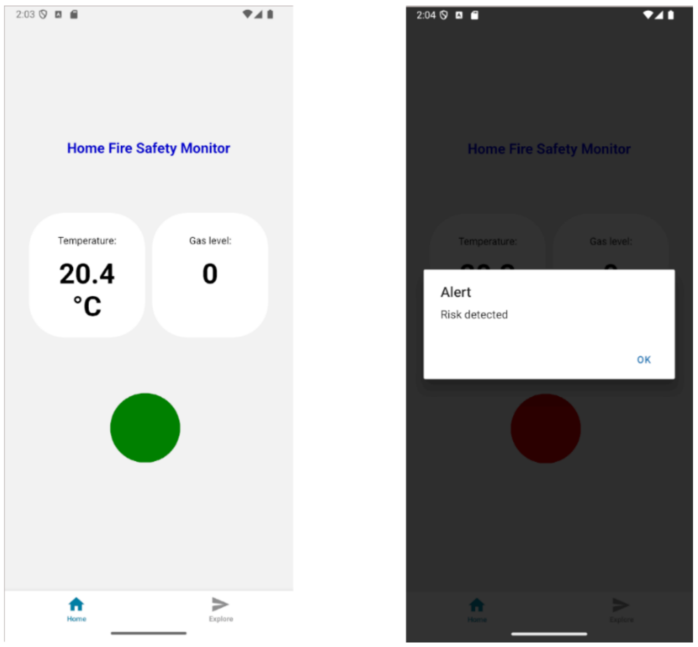


Figure 6. The App subscribes to the sensor reading and alert from the cloud.

4.4 Hardware Components

The hardware implementation centers around the ESP32 microcontroller, chosen for its built-in Wi-Fi, dual-core processing power, and GPIO flexibility. The prototype integrates three key sensors for environmental monitoring, along with alert components and a display module for real-time data visualization on the IoT.

Key Hardware Components:

- **ESP32 Development Board**: Acts as the main processing and communication unit.
- **MQ-2 Gas Sensor**: Detects combustible gases such as methane, butane, and smoke.
- **DHT11 Sensor**: Measures ambient temperature and humidity.
- **IR Flame Sensor**: Detects the presence of a flame using infrared light.
- **Buzzer**: Provides audible alerts when thresholds are breached.
- **LED (Onboard)**: Offers a visual indication of alerts.
- **OLED Display (SSD1306 128x64)**: Displays real-time sensor readings and system status.
- **Power Supply:** The system is powered through a 5V USB connection, which supplies the ESP32 and all connected peripherals. While sufficient for testing and prototyping, future versions may incorporate a battery backup or voltage regulation for standalone operation.

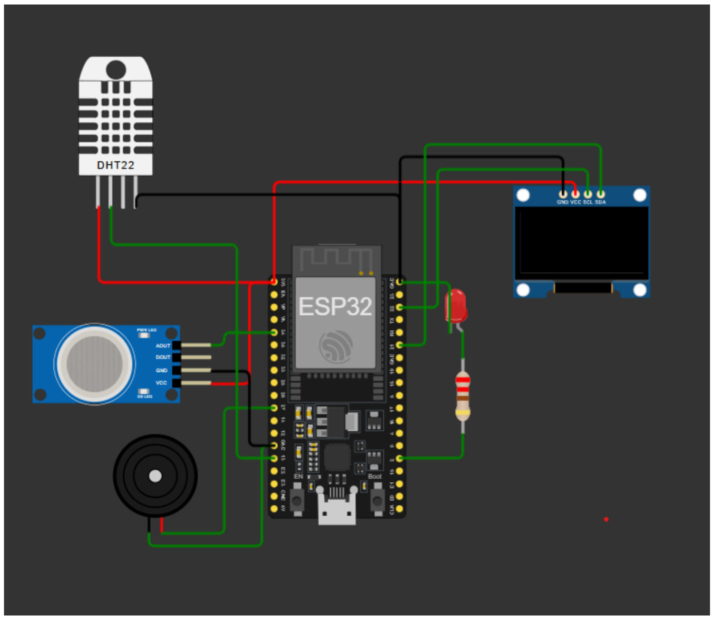


Figure 7. Circuit Design and Connections

4.5 Software Components

4.5.1. ESP32

The ESP32 is programmed with an Arduino Integrated Development Environment (IDE). It is an open-source software used to write, compile, and upload code to multiples microcontroller types, including ESP32 and others. Arduino has an active community producing various packages and libraries to configure the microcontroller and peripherals. The key libraries used, and their function are:

- WiFiClientSecure.h: Secure connection between the device and AWS IoT.
- PubSubClient.h: Implement MQTT protocol to publish sensor data to AWS IoT.
- ArduinoJson.h: Create and process JSON data. Specifically, it parses the SSID and Wi-Fi password received from the App and generates the MQTT message sent to AWS IoT.
- WiFi.h: Supports Wi-Fi connectivity for the device. It enables Access point (AP) mode and WiFi mode to connect to the App and internet, respectively.
- WebSocketsServer.h: Implements a WebSocket server on the ESP32, enabling real-time bidirectional communication with the App for receiving credentials and commands.
- Wire.h: Transfer data between the device and the OLED
- Adafruit_GFX.h and Adafruit_SSD1306.h: Control the UI of the OLED.
- DHT.h: Enable the temperature and humidity detection functions of the DHT11.

4.5.2 Mobile App

The mobile app is developed with the React Native framework. It is an open-source JaveScript library for building Android and iOS functions and UI. With regards to this project, the React useSate() and useCallback() hook are essential to enable the app to dynamically listen to user action from the frontend (such as Wi-Fi and device ID input), and then initiate further functions and API calls (e.g. subscribe to MQTT broker messages) in the backend and render the result interactively back on the frontend.
The useState() hook enabled the App to manage the state of the variables and actively share them accross the functions and components.

Secondly, a key function of the App is to provision the device Wi-Fi connectivity. The React Native supports the WebSockets protocol, thus enabling communication and data transfer between the App to the ESP.

4.6 Cloud architecture

The mobile App remote data streaming feature is provided by AWS IoT service. AWS IoT is a fully managed service that connects IoT devices and routes their data from and to the cloud. Each IoT device is registered as a "Thing" in the AWS IoT registry. Each Thing has a certificate to identify the device and a private key for authorized connection. It also provides secure features, data processing, and device management capabilities, enabling businesses to build platforms and fleets of IoT applications.

Communication between AWS IoT is supported by protocols like MQTT, HTTPS, and LoRaWAN. In this production, MQTT is used.

MQTT (Message Queuing Telemetry Transport) is a publish and subscribe model. It uses an MQTT broker to manage the sending and receiving of data between publishers (IoT) and subscribers (App users). MQTT protocol offers an efficient and scalable solution to manage multiple publishers and subscribers concurrently.

5. Implementation

5.1 Hardware Setup

The firmware of the ESP32 development board is managed using the manufacturer Espressif's own package. The JSON package "https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json" is added to the Arduino board manager to enable ESP32 board support in the Arduino IDE.

5.2 Cloud integration

An AWS IoT "Thing" is pre-registered to generate the device certicate and security keys. These credentials are used by the ESP32 to assign itself as the "Thing" using the connectAWS() function.

```
void connectAWS() {
  net.setCACert(AWS_CERT_CA);
  net.setCertificate(AWS_CERT_CRT);
  net.setPrivateKey(AWS_CERT_PRIVATE);
  client.setServer(AWS_IOT_ENDPOINT, 8883);
  client.connect(/**The name of the "Thing" setup on AWS side*/)
  ...
}
```

5.3 Mobile App development

5.3.1 App to IoT communication

The App communicates with the IoT via ESP32 WebSocket server to set up IoT Wi-Fi connection.

```
ws.current = new WebSocket(`ws://${/ESP Access Point IP}:81`);
ws.current.onopen = () => {
  ws.current?.send(JSON.stringify({ ssid: espSsid, password: espPassword,
  tempThre: tempThre }));
};
```

On the ESP side, the webSocketEvent() function handler parsed the JSON message from the App and triggers an internet connection.

```
void webSocketEvent(uint8_t num, WStype_t type, uint8_t * payload, size_t length)
{
  switch(type) {
    case WStype_DISCONNECTED:
      ...
    case WStype_CONNECTED:
      ...
    case WStype_TEXT:
      {
        Serial.printf("[%u] Get text: %s\n", num, payload);
        String msg = String((char*)payload);
        // Extract SSID
        int ssidKey = msg.indexOf("ssid");
        int ssidQuote1 = msg.indexOf("\"", ssidKey + 5);
        int ssidQuote2 = msg.indexOf("\"", ssidQuote1 + 1);
        ssidFromApp = msg.substring(ssidQuote1 + 1, ssidQuote2);
        // Extract Password
        int passKey = msg.indexOf("password");
        int passQuote1 = msg.indexOf("\"", passKey + 9);
        int passQuote2 = msg.indexOf("\"", passQuote1 + 1);
        passwordFromApp = msg.substring(passQuote1 + 1, passQuote2);
        // Extract Temp
        int tempKey = msg.indexOf("tempThre");
        int tempQuote1 = msg.indexOf("\"", tempKey + 9);
        int tempQuote2 = msg.indexOf("\"", tempQuote1 + 1);
        tempThresFromApp = msg.substring(tempQuote1 + 1, tempQuote2);
        int convertTempToInt = tempThresFromApp.toInt();
        if (ssidFromApp.length() > 0 && passwordFromApp.length() > 0 &&
        convertTempToInt > 0) {
            receivedWiFiCredentialFromApp = true; // trigger WiFi mode
            TEMP_THRESHOLD = convertTempToInt;
            Serial.println("Temp threshold updated to: " + TEMP_THRESHOLD);
        }
      }
```

5.3.2 App to AWS communication

The App to AWS connection is authentic using AWS Amplify service. AWS Amplify connects backend and frontend App applications by providing authentication and API management. The identity created has an IAM policy attached to allows access to the registered AWS IoT "Thing".

```
const endpoint = // the AWS IoT endpoint;
Amplify.configure({
  Auth: {
    identityPoolId: // IAM roles that allow access to the registered AWS IoT
"Thing",
    region: // AWS region,
  },
});
Amplify.addPluggable(new AWSIoTProvider({
  aws_pubsub_region: // AWS region,
  aws_pubsub_endpoint: // the MQTT endpoint where the data is sent
}));
```

Once authenticated, the App fetch sensor readings using Amplify's PubSub function. PubSub() is a real-time, event-driven function that continuously listens to new messages. When new message arrives to the React DOM and updated the alertStatus value. The new data is immediately parsed to the App UI via the useEffect() hook.

```
useEffect(() => {
  ...
  const subscription = PubSub.subscribe(/** MQTT topic*/).subscribe({
    next: (data) => {
      const payload = typeof data.value === 'string' ? JSON.parse(data.value) :
      data.value;
      ...
      setAlertStatus(
        ...
     );
    }
  })
}, [alertStatus]);
```

6. Testing and Results

6.1 Testing Procedure

Hardware Function:

The Arduino `Serial.print()` function was utilized extensively to monitor and debug the internal state of the ESP32 microcontroller. During each test cycle, sensor readings, threshold checks, and activation of outputs (LED, buzzer, etc.) were printed to the Serial Monitor, allowing us to confirm proper logic of flow and hardware responses.

Sensor functions:

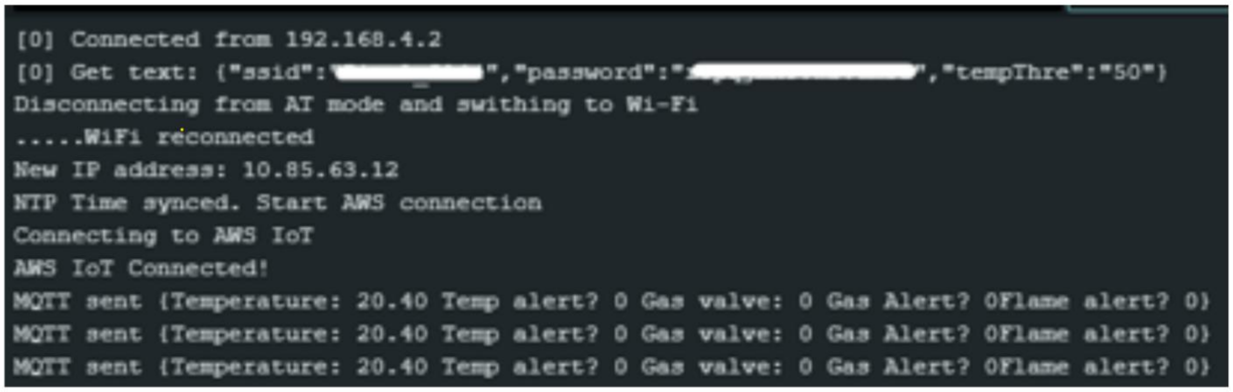
We recorded a series of videos capturing the live response of the system to real-world test conditions. This includes exposure to a flame, a gas leak simulation, and heat sources. These videos serve as proof-of-concept material for clients and stakeholders and were also used in formal project presentations.

App: The UI is used to test the behavior of the app.

6.2 Test Scenarios and Observations

6.2.1 WebSocket and Wi-Fi connection

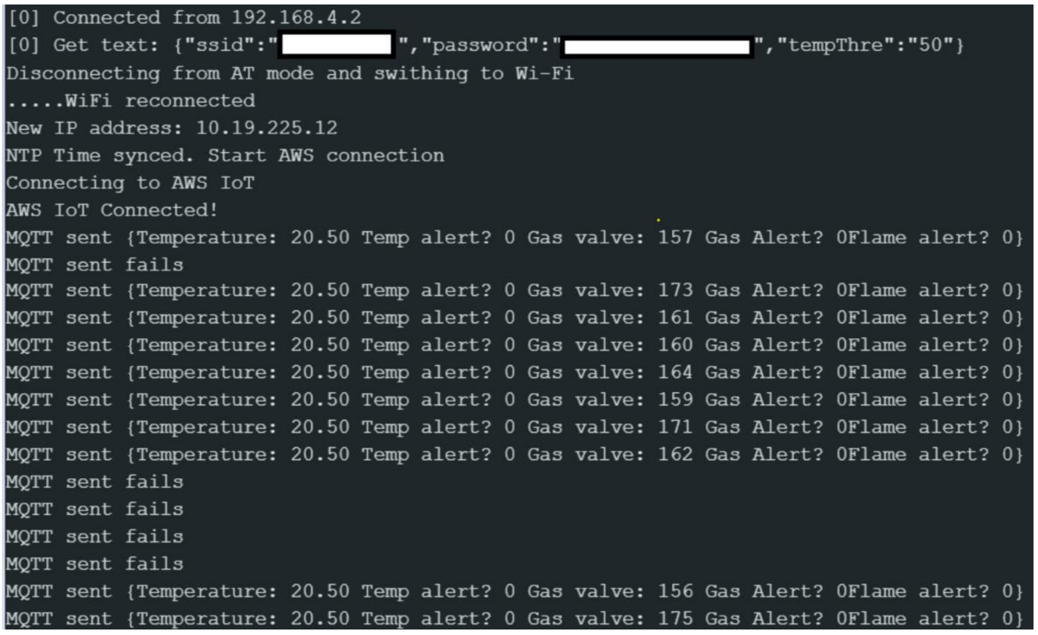
Test case 1: An success end-to-end process where the IoT connect to internet and AWS.



```
[0] Connected from 192.168.4.2
[0] Get text: {"ssid":"[redacted]","password":"[redacted]","tempThre":"50"}
Disconnecting from AT mode and swithing to Wi-Fi
.....WiFi reconnected
New IP address: 10.85.63.12
NTP Time synced. Start AWS connection
Connecting to AWS IoT
AWS IoT Connected!
MQTT sent {Temperature: 20.40 Temp alert? 0 Gas valve: 0 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.40 Temp alert? 0 Gas valve: 0 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.40 Temp alert? 0 Gas valve: 0 Gas Alert? 0Flame alert? 0}
```

Figure 8. Log showing ESP WebSocket receives messages from the App and switches to Wi-Fi mode and publishing data to AWS MQTT broker.

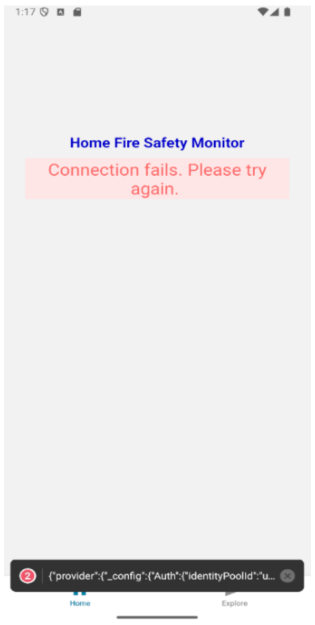
Test case 2: Simulate intermittent Wi-Fi connection between IoT and internet by turn off the WiFi.



```
[0] Connected from 192.168.4.2
[0] Get text: {"ssid":"[redacted]","password":"[redacted]","tempThre":"50"}
Disconnecting from AT mode and swithing to Wi-Fi
.....WiFi reconnected
New IP address: 10.19.225.12
NTP Time synced. Start AWS connection
Connecting to AWS IoT
AWS IoT Connected!
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 157 Gas Alert? 0Flame alert? 0}
MQTT sent fails
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 173 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 161 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 160 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 164 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 159 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 171 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 162 Gas Alert? 0Flame alert? 0}
MQTT sent fails
MQTT sent fails
MQTT sent fails
MQTT sent fails
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 156 Gas Alert? 0Flame alert? 0}
MQTT sent {Temperature: 20.50 Temp alert? 0 Gas valve: 175 Gas Alert? 0Flame alert? 0}
```

Figure 9. Log showing sending MQTT message is not terminated by internet disruption.

Test case 3: Simulate intermittent Wi-Fi connection between App and the internet.

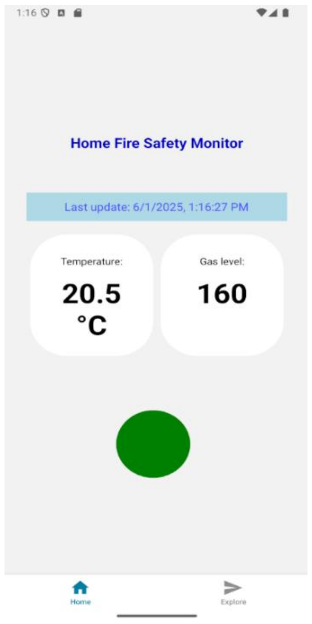


Connection fails. Please try again.

{"provider":{"_config":{"Auth":{"identityPoolId":"u...

Figure 10. The UI alerts App side connection fails message to notify user to reconnect to get latest senor reading.

Test case 4: Simulate App behavior when IoT to internet connection or AWS cloud service is down.



Last update: 6/1/2025, 1:16:27 PM

Temperature: 20.5 °C

Gas level: 160

Figure 11. The App dashboard has a timer showing the latest timestamp of sensor data fetched from AWS. Through the App, it indicates whether the IoT of AWS side is disconnected.

6.2.2 Hardware & Sensor Functions

Test Scenario	Expected Outcome	Actual Observation	Status
Exposure to open flame	Flame sensor detects fire, buzzer and LED activated	Flame detected, buzzer and LED turned on instantly	Pass
Release of smoke near MQ2 sensor	Gas sensor detects concentration > threshold	Gas level exceeded threshold; system triggered alert	Pass
Temperature above 40°C	Temperature reading crosses threshold, warning activated	System displayed high temperature warning	Pass

6.3 Performance Analysis

The system demonstrated high reliability and responsiveness during testing. The integration between hardware and mobile interface was seamless, and all modules operated within acceptable thresholds.

- **Sensor Responsiveness**: All sensors responded promptly to changes in environmental conditions and triggered warnings accordingly.
- **Synchronization**: The video evidence confirmed near-instantaneous data flow between the ESP32 and the mobile app, ensuring users are alerted in real-time.
- **App Performance**: The app maintained a consistent connection with the ESP32, and displayed sensor values without noticeable delay or crashes.

7. Challenges and Solutions

7.1 Hardware Issues

Procurement: The team had to source cost-effective components from online marketplaces, and delivery took approximately three weeks. As a result, actual development work was delayed until the shipment arrived, impacting the project timeline.

Risk of Component Damage:
An additional challenge stemmed from the lack of redundancies in hardware components. With only one unit of each sensor available and long lead times for replacements, there was little room for error. For instance, if a component like the MQ2 sensor was accidentally damaged or "burnt out" during testing, it would take several weeks to receive a replacement, significantly hindering progress. This risk forced the team to adopt very careful handling and conservative testing procedures to avoid unnecessary damage.

7.2 Connectivity and Cloud Challenges

Network Dependency: The WebSocket connection requires both the IoT device and the mobile app to be on the same local network. Since the team had access to only one ESP32 device, members could not conduct parallel testing or programming. This constraint significantly prolonged the development process beyond initial expectations.

7.3 App Development Hurdles

Lack of Experience: The team had no prior experience with mobile app development. During setup, multiple compatibility issues arose between Android Studio, the Android SDK, and required libraries. A substantial amount of time was spent resolving environmental setup challenges before meaningful development could begin.

8. Conclusion

8.1 Summary of Achievements

This project successfully delivered a working prototype of an IoT-based gas and fire detection system that combines cost-efficiency with real-time monitoring. It integrates the ESP32 microcontroller with the MQ-2 gas sensor, IR flame sensor, and DHT11 temperature sensor. A custom mobile app, built using React Native, allows users to connect to the device locally and remotely to receive real-time alerts through AWS IoT Core.

The key achievements include:

- A complete hardware setup using ESP32, MQ-2, DHT11, and IR flame sensors, with OLED display, LED indicators, and buzzer alerts.
- Wi-Fi provisioning from the mobile app to the ESP32 using WebSocket protocol

- Real-time data transmission and alerting through MQTT over AWS IoT Core with secure certificate-based communication
- A functional React Native mobile app that displays live sensor readings, configures thresholds, and notifies users of alerts.
- End-to-end integration and testing, confirming reliable performance between device, cloud, and mobile app.

8.2 Lessons Learned

During the development process, we learned how to manage cloud-device communication using MQTT, how to configure AWS IoT Core with secured certificates, and how to handle edge cases like failed Wi-Fi connections. Hardware calibration was essential, especially for flame and gas sensors.

We also learned the limitations of low-cost components and how to balance between performance and affordability. Building the mobile app taught us the importance of responsive UI and real-time state handling using React Native. Lastly, clear system flow and component integration played a major role in simplifying user interaction.

9. Future Work

This system can be improved in a number of ways in the future. Adding machine learning (ML) or artificial intelligence (AI) to make the system smarter is one major area. For instance, the system might be able to predict possible gas leaks or fire hazards before they occur by learning from sensor data over time. This would increase safety and provide users with earlier warnings.

The mobile app can also be upgraded to support data logging, graphs, and customizable alert settings, helping users better track trends and control their own thresholds. It would also be useful to introduce SMS or call alerts as a backup in case the internet connection fails. On the hardware side, adding battery backup or solar charging would allow the device to run even during power cuts.

To move toward a production-ready product, we would need to focus on proper hardware design, including custom PCBs, durable casing, and clean internal wiring to make the product compact and reliable.

On the software side, before releasing the mobile app, it's important to plan proper device and user account management, ensure security controls (like login protection and encrypted data), and possibly conduct user testing to improve usability. These steps will help create a complete and user-friendly solution ready for real-world use.

10. References

Alsan Parajuli (2024) IoT Smoke & Gas Detector using ESP8266 & Blynk. Available at: https://iotprojectsideas.com/iot-smoke-gas-detector-using-esp8266-blynk/ [Accessed 1 March 2025].

Amazon Web Services (2024) What is AWS IoT Core?. Available at: https://aws.amazon.com/iot-core/ [Accessed 30 May 2025].

Amazon Web Services, Inc. (no date) Configure Amplify categories. Available at: https://docs.amplify.aws/gen1/react/prev/tools/libraries/configure-categories/ [Accessed: 9 May 2025].

Amazon Web Services, Inc. (no date) Publish/subscribe AWS IoT Core MQTT messages. Available at: https://docs.aws.amazon.com/greengrass/v2/developerguide/ipc-iot-core-mqtt.html [Accessed: 9 May 2025].

DFRobot (2017) ESP32 Arduino Tutorial: Websocket client. Available at: https://www.dfrobot.com/blog-776.html [Accessed 1 May 2025].

Horne, M. (2024) The best smart home devices of 2024. Android Authority. Available at: https://www.androidauthority.com/best-smart-home-devices-3496884/ [Accessed 30 May 2025].

JC Instruments (2025) Gas Leak Detector Combustible Flammable Natural LPG Tester HT601A. Available at: https://www.jcinstruments.com.au/products/gas-leak-detector-combustible-flammable-natural-lpg-tester-ht601a [Accessed 30 May 2025].

Last Minute Engineers (no date) How MQ2 Gas/Smoke Sensor Works? & Interface it with Arduino. Available at: https://lastminuteengineers.com/mq2-gas-senser-arduino-tutorial/ [Accessed 20 March 2025].

Mamtaz Alam (2022) Connecting ESP32 to Amazon AWS IoT Core using MQTT. Available at: https://how2electronics.com/connecting-esp32-to-amazon-aws-iot-core-using-mqtt/ [Accessed 9 May 2025].

Martyn Currey (no date) ESP8266 and the Arduino IDE Part 9: Websockets. Available at: https://www.martyncurrey.com/esp8266-and-the-arduino-ide-part-9-websockets/ [Accessed 1 May 2025].

ProDetec (2020) Flame Detection. Available at: https://prodetec.com.au/wp-content/uploads/2020/04/Flame-Detection-Application-Note.pdf [Accessed 20 March 2025].

RandomNerdTutorials (No data) How to Set an ESP32 Access Point (AP) for Web Server. Available at: https://randomnerdtutorials.com/esp32-access-point-ap-web-server/ [Accessed 1 May 2025].

Ravina, E., Angulo, I., Zuloaga, A., Goiri, I., Ruiz, R. and Ugalde, G. (2020) 'Low-cost IoT prototypes for indoor and outdoor environmental monitoring', Sensors, 20(20), pp. 1–19. doi: 10.3390/s20205774.

React Native (no date) Networking. Available at: https://reactnative.dev/docs/network [Accessed: 7 May 2025].

Shawn Hymel (2019) How to Create a Web Server (with WebSockets) Using an ESP32 in Arduino. Available at: https://shawnhymel.com/1882/how-to-create-a-web-server-with-websockets-using-an-esp32-in-arduino/ [Accessed 1 May 2025].

SunFounder (2024) Lesson 04: Gas Sensor Module (MQ-2). Available at: https://docs.sunfounder.com/projects/umsk/en/latest/02_arduino/uno_lesson04_mq2.html [Accessed 30 May 2025].