

158335 Assignment 2

Name: Meixian Shi

ID: 22009303

Contents

1	Problem Identification and Research	2
1.1	Problem Statement	2
1.2	Background Analysis	2
1.3	IoT-Cloud Justification	2
2	System Design	3
2.1	Architecture Overview	3
2.2	Component Selection	3
3	Prototype Implementation	4
3.1	Cloud Backend Architecture	4
3.2	Data Ingestion	4
3.2.1	MQTT Protocol	4
3.2.2	IoT Rule Configuration	5
3.3	Data Processing	5
3.4	Security Implementation	7
3.4.1	Device Authentication and Communication Security	7
3.4.2	Cloud-Side Access Control	7
3.5	UI Features	8
3.5.1	Real-Time Data Visualization	8
3.5.2	Alert Status Indication	8
3.5.3	User Authentication System	9
3.6	Data Analytics	10
3.7	Real-Time Trend Visualization	10
4	Deployment Guide	11
4.1	Hardware Configuration	11
4.2	Cloud Deployment	11
4.3	Frontend Deployment	12
4.4	Verification Process	12
5	User Manual	12
5.1	System Startup	12
5.2	Accessing the Dashboard	12
5.3	Interface Features	12
5.4	Alert Handling	13
5.5	Troubleshooting	13

1 Problem Identification and Research

1.1 Problem Statement

In modern households and industrial settings, abnormal temperature and humidity levels can cause serious issues:

Health risks: High humidity levels (above 80%) can lead to mold growth, increasing the risk of respiratory illnesses. High temperatures (above 50°C) may accelerate equipment degradation and even cause fires.

Limitations of traditional solutions: Commercial temperature and humidity meters typically offer only local, real-time data, lacking features such as remote monitoring, historical data analysis, and automated alerts. As a result, they are inadequate for responding to unexpected anomalies in a timely manner.

1.2 Background Analysis

Standalone IoT devices: Some solutions (e.g., Arduino with DHT11 sensors) only support local storage, making long-term data retention unfeasible.

Commercial cloud platforms: Certain services (e.g., ThingSpeak) have limited functionality or high costs, making them unsuitable for flexible alert logic or scalable use.

1.3 IoT-Cloud Justification

Real-time responsiveness: The MQTT protocol ensures that device data is transmitted to the cloud within 5 seconds, meeting the timing demands for early fire detection.

Scalability: DynamoDB provides auto-scaling storage capabilities, supporting future expansion to multiple devices.

Remote accessibility: A web-based dashboard removes geographical barriers, allowing users to monitor environmental status from any device.

Cost-effectiveness: The AWS free tier covers initial usage needs, while the hardware cost is significantly lower compared to commercial systems (e.g., Honeywell solutions).

2 System Design

2.1 Architecture Overview

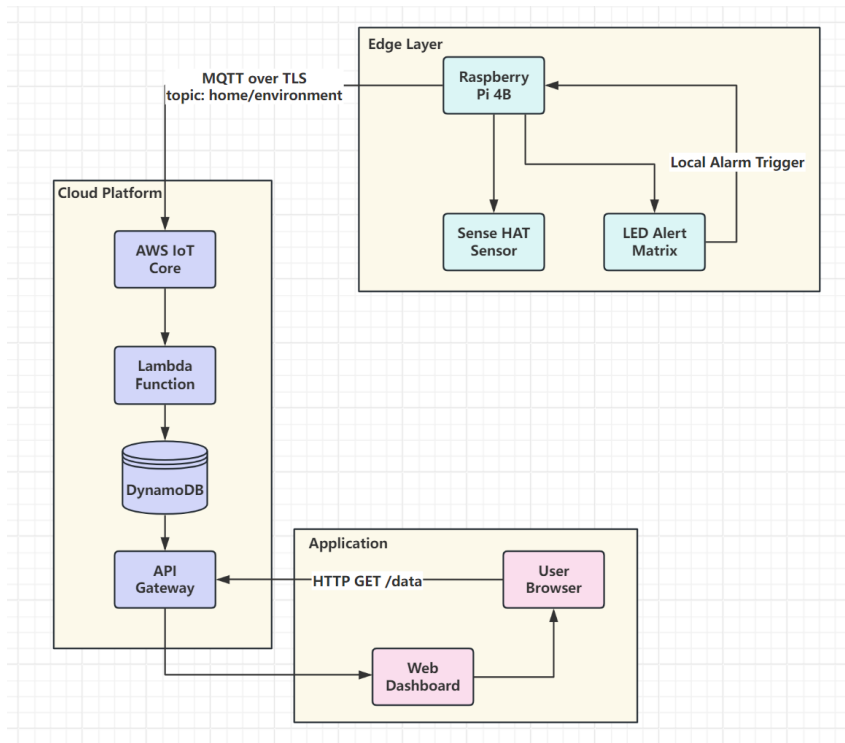


Figure 1: System Architecture

2.2 Component Selection

Components:

1. Raspberry Pi 4B
2. Sense HAT Simulator

Cloud Service Configuration: The cloud infrastructure is built using the AWS IoT technology stack, with each component working together to enable end-to-end data processing:

- **Device Connectivity Layer:** AWS IoT Core serves as the MQTT message broker, receiving data from edge devices. A minimum-privilege policy named `EnvironmentMonitorPolicy` is configured to restrict devices to publishing messages only to the `home/environment` MQTT topic.
- **Data Storage Layer:** A DynamoDB database is deployed, using timestamp (in UNIX format) as the partition key and `device_id` as the sort key. Five read/write capacity units are provisioned to support basic throughput performance.
- **Data Processing Layer:** A Lambda function named `SaveEnvironmentData` handles real-time data processing and persistence. It runs in a Python 3.9 environment and is granted permissions via an execution role named `labRole`.

- **API Service Layer:** AWS API Gateway is used to expose a RESTful endpoint for cross-origin data access. A GET method is configured to retrieve the latest 20 environmental records.

3 Prototype Implementation

3.1 Cloud Backend Architecture

The edge device publishes sensor data every 5 seconds to the *home/environment* topic via the MQTT protocol. The AWS IoT Rules Engine listens to this topic in real time and triggers a Lambda function. After data cleansing, the Lambda function stores the processed data into a DynamoDB table. An API Gateway endpoint provides a GET /data interface to retrieve the latest 20 records. The frontend polls this API using AJAX to support real-time visualization.

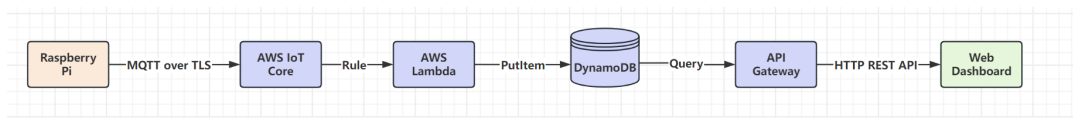


Figure 2: Cloud Backend Architecture

3.2 Data Ingestion

3.2.1 MQTT Protocol

Device policy strictly allows publishing to the specified topic only.

```

# mqttToAWS.py core code snippet
def collect_and_publish():
    while True:
        # Sensor data collection
        raw_temp = sense.get_temperature()
        raw_humi = sense.get_humidity()

        # Data cleaning and formatting
        payload = {
            "device_id": "rasip",
            "temperature": round(raw_temp, 2),
            "humidity": min(max(raw_humi, 0), 100)
        }

        # MQTT message publishing
        mqtt_connection.publish(
            topic="home/environment",
            payload=json.dumps(payload),
            qos=mqtt.QoS.AT_LEAST_ONCE
        )
        time.sleep(5) # 5-second data collection cycle
  
```

Listing 1: mqttToAWS.py snippet

3.2.2 IoT Rule Configuration

Rule Name: EnvironmentDataRule
SQL Statement: SELECT * FROM 'home/environment'
Purpose: Forwards environmental sensor data to a Lambda function for processing.

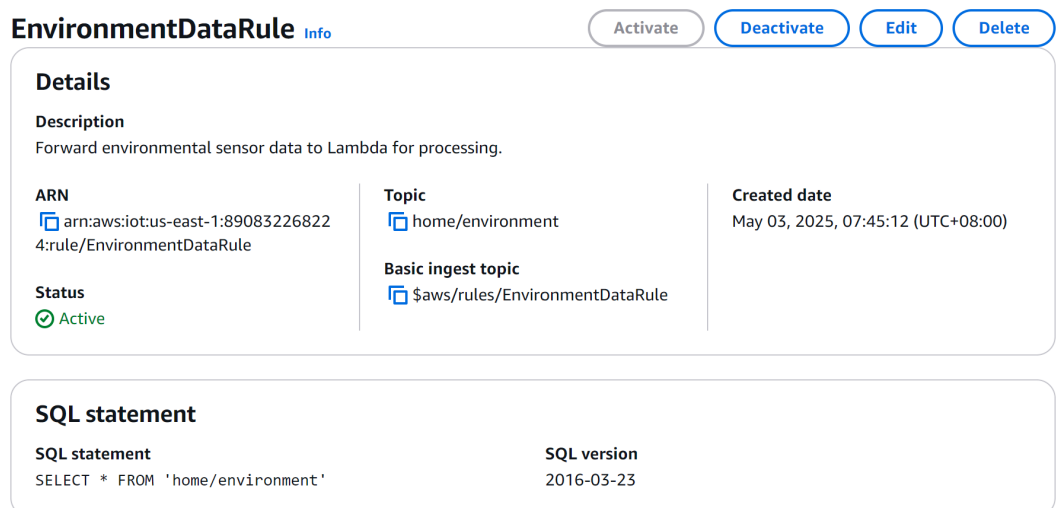


Figure 3: IoT Rule

3.3 Data Processing

The core logic of the Lambda function SaveEnvironmentData is as follows:

- 1. Receive Raw Data:** Receives MQTT message payload (event) from AWS IoT Core in JSON format, containing temperature, humidity, and *device_id*.
- 2. Type Conversion and Cleaning:** Converts temperature and humidity to float, ensuring humidity values are within a physically valid range (0%-100%) to avoid sensor outliers.
- 3. Precision Handling:** Converts numerical values to the DynamoDB-compatible Decimal type, rounding to two decimal places (e.g., 52.30°C) to prevent precision loss from floating-point values.
- 4. Data Storage:** Stores the processed record, including *device_id* and a server-side generated timestamp, into the EnvironmentData.Maeve DynamoDB table. If any exception occurs (e.g., due to invalid data format), the function returns an HTTP 500 status code and logs the error.

```
import json
import boto3
```

```

import time
from decimal import Decimal

def lambda_handler(event, context):
    print("Received raw event:", json.dumps(event))

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('EnvironmentData_Maeve')

    try:
        payload = event if isinstance(event, dict) else json.loads(
event)

        temperature = float(payload.get('temperature', 0.0))
        humidity = float(payload.get('humidity', 0.0))
        device_id = payload.get('device_id', 'rasip')

        humidity = min(max(humidity, 0.0), 100.0)

        item = {
            'timestamp': int(time.time()),
            'device_id': device_id,
            'temperature': Decimal(str(round(temperature, 2))),
            'humidity': Decimal(str(round(humidity, 2)))
        }


        table.put_item(Item=item)
        print("Successfully saved:", item)
        return {'statusCode': 200}

    except Exception as e:
        print("Fatal error:", str(e))
        return {'statusCode': 500}




```

Listing 2: Code for Data Processing

Table: EnvironmentData_Maeve - Items returned (50)


[Actions](#)
[Create item](#)

Scan started on May 09, 2025, 08:00:25


1



<input type="checkbox"/>	timestamp (Number)	device_id (String)	humidity
<input type="checkbox"/>	1746263753	rasip	44.87
<input type="checkbox"/>	1746312174	rasip	94.98
<input type="checkbox"/>	1746264843	rasip	44.78
<input type="checkbox"/>	1746261004	rasip	54.61

Figure 4: EnvironmentData Table Items

3.4 Security Implementation

3.4.1 Device Authentication and Communication Security

Implementation Details:

The Raspberry Pi establishes a secure connection with AWS IoT Core using a pre-configured X.509 certificate (located in the *certificates/* directory). Each certificate contains a unique device identifier and is associated with the AWS IoT policy `EnvironmentMonitorPolicy`.

MQTT communication enforces TLS 1.2 encryption over port 8883. Secure communication is achieved using `mqtt_connection_builder.mtls_from_path` in the code.

Certificate Rotation:

New certificates can be generated via the AWS IoT Console and used to replace the old ones on the device. The outdated certificate is deactivated to prevent any risk of data leakage.

3.4.2 Cloud-Side Access Control

Principle of Least Privilege:

1. Lambda Execution Role:

The Lambda function uses the `labRole` execution role, which is granted only the `dynamodb:PutItem` permission (via the managed policy `AmazonDynamoDBFullAccess`). This ensures that the function can only write to the designated table `EnvironmentData_Maeve`.

2. IoT Policy Restrictions:

The `EnvironmentMonitorPolicy` explicitly allows the device to only publish MQTT messages to the topic *home/environment*. All other actions (e.g., subscribe or delete) are denied.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:us-east-1:890832268224:client/rasip"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "arn:aws:iot:us-east-1:890832268224:topic/home/
environment"
    }
  ]
}
```

Listing 3: Policy in Json

3.5 UI Features

3.5.1 Real-Time Data Visualization

The frontend uses the ApexCharts library to dynamically render temperature and humidity line charts, implementing the following features:

Data Fetching: JavaScript calls the *fetchData()* function to retrieve the latest 20 records from the API Gateway endpoint (<https://toxyu2j8cc.../dev/data>), displaying them in reverse chronological order.

Chart Updating: The *chart.updateSeries()* method is invoked to refresh the chart in real-time. If the temperature exceeds 50°C, the temperature line switches to red; if the humidity exceeds 80%, the humidity line switches to blue (as defined in the *updateChartData* function in *script.js*).

Timestamp Handling: The X-axis converts UNIX timestamps (in milliseconds) into local time format to prevent timezone-related confusion.

3.5.2 Alert Status Indication

Status Banner: The top banner dynamically changes style based on alert conditions:

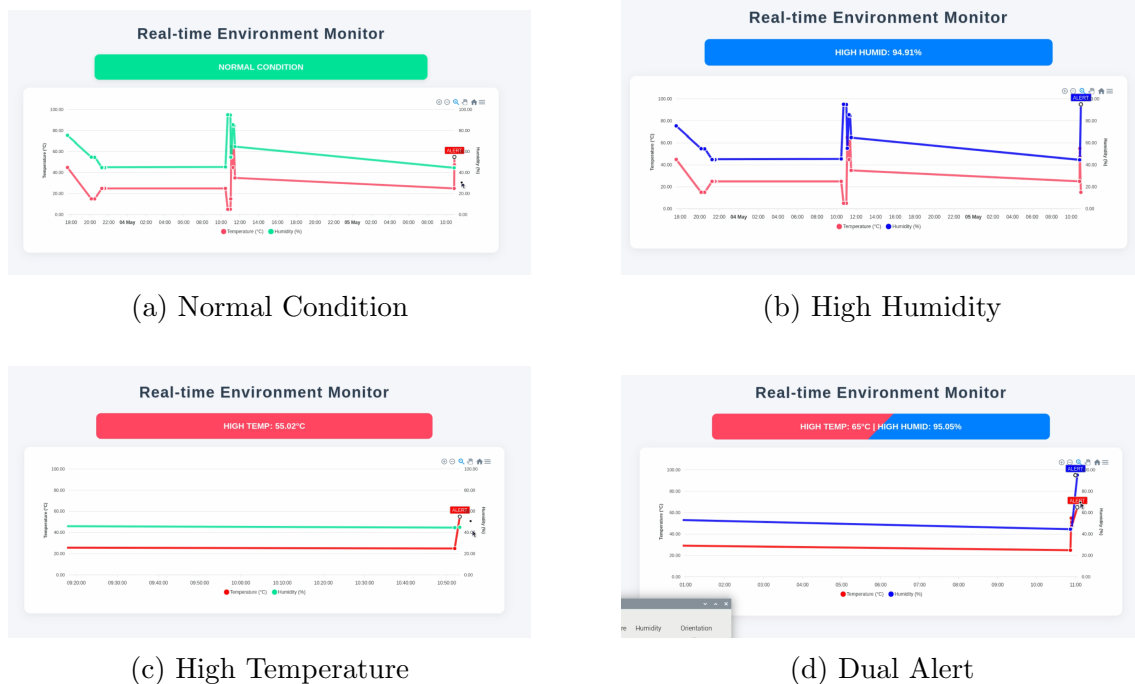


Figure 5: Frontend Alert

Chart Annotation: When the latest data point exceeds the defined threshold, the chart dynamically adds an "ALERT" label using *chart.addPointAnnotation()* (red or blue).

Device-Level Detection: On the device side, *mqttToAWS.py* directly controls the LED alert system via the *set_alarm_led()* function. It activates red or blue LED text alerts when temperature exceeds 50°C or humidity exceeds 80%, respectively.

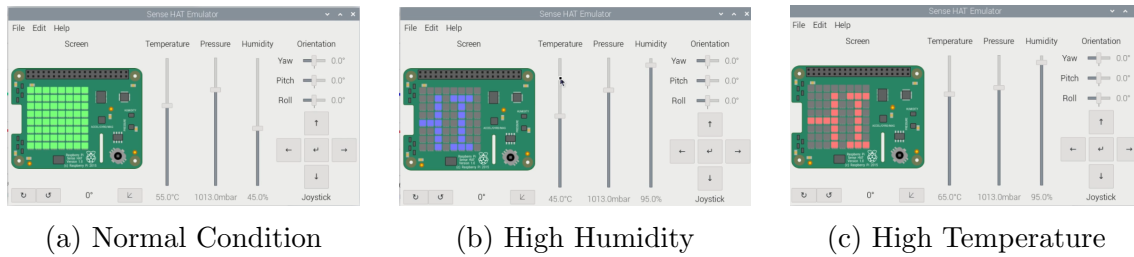


Figure 6: Device Alert

3.5.3 User Authentication System

On first access, a dialog prompts the user to enter a username and password. Upon successful verification, the authentication state is stored via *localStorage.setItem()*.



Figure 7: Authentication Check

The login status is checked using *localStorage.getItem('env_monitor_auth')*. Unauthenticated users are blocked from viewing data and are forced to refresh the page.

This authentication method is intended for development and testing purposes only. Credentials are transmitted and stored in plain text, with no HTTPS encryption or server-side validation implemented.

```
function checkAuth() {
  if (!localStorage.getItem('env_monitor_auth')) {
    const username = prompt('Enter username:');
    const password = prompt('Enter password:');
    if (username === 'admin' && password === 'admin123') {
      localStorage.setItem('env_monitor_auth', 'authenticated');
    }
  }
}
```

Listing 4: checkAuth Code Snippet

3.6 Data Analytics

To verify the integrity of system-collected data and analyze temperature and humidity trends, this project utilizes the AWS DynamoDB console and CloudWatch Logs Insights for basic querying and visualization.

First go to DynamoDB and run the following query to retrieve the 20 most recent records, ensuring the device is uploading data correctly:

```
SELECT * FROM "EnvironmentData_Maeve"
WHERE device_id = 'rasip'
ORDER BY timestamp DESC
LIMIT 20;
```

Listing 5: query the data

Then go the CloudWatch console and navigate to Logs Insights. Use the query to calculate average temperature and humidity, grouped by 5-minute intervals across the most recent 100 logs.

```
fields @timestamp, temperature, humidity
| sort @timestamp desc
| limit 100
| stats avg(temperature) as avg_temp, avg(humidity) as avg_humi by bin(5
m)
```

Listing 6: query the data

Below is the chart generated by CloudWatch Visualize function.

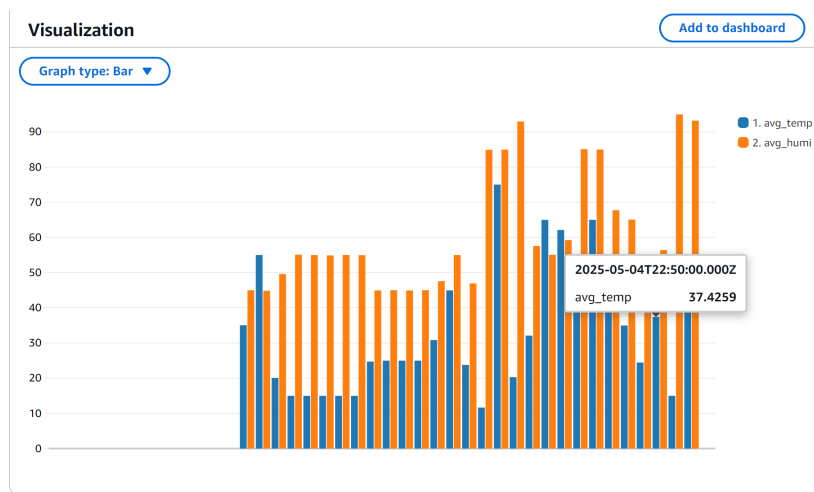


Figure 8: CloudWatch Logs Insight

3.7 Real-Time Trend Visualization

The frontend dynamically displays temperature and humidity trends using Apex-Charts line charts. Users can zoom in on specific time ranges (e.g., past 1 hour or 24 hours) to explore data.

Interactive tooltips show precise values and timestamps on hover, aiding manual analysis of abnormal periods.

4 Deployment Guide

4.1 Hardware Configuration

The Raspberry Pi runs a pre-installed Raspberry Pi OS on a microSD card. It executes the *mqttToAWS.py* Python script for data collection and transmission.

Dependencies: awsiotsdk (install via `pip3 install`)

Sense HAT simulator (install via `sudo apt-get install sense-hat`)

The device certificate files (.pem format) are stored in the *certificates/* directory. The script must explicitly specify the paths to the device certificate, private key, and root CA to ensure a secure MQTT connection over TLS 1.2 to the AWS IoT Core endpoint: `a18s752r9nbkhs-ats.iot.us-east-1.amazonaws.com`.

Below is my project structure:

```
project_root/
|-- mqttToAWS.py          # Data transmission script
|-- certificates/         # Certificate files
|   |-- AmazonRootCA1.pem
|   |-- AmazonRootCA3.pem
|   |-- certificate.pem.crt
|   |-- public.pem.key
|   |-- private.pem.key
|-- dashboard/            # Web dashboard files
|   |-- index.html        # Main interface
|   |-- script.js         # Data fetching logic
```

4.2 Cloud Deployment

Register the device in AWS IoT Core and attach the `EnvironmentMonitorPolicy`, allowing the device to publish only to the *home/environment* topic.

Create a DynamoDB table named *EnvironmentData_Maeve* with *timestamp* (Number) and *device_id* (String) as the primary key.

Set up a Lambda function `SaveEnvironmentData` (runtime: Python 3.9), triggered by an IoT Rule, with execution role `labRole` to enable data insertion.

Use API Gateway to create a REST endpoint: `https://toxyu2j8cc.execute-api.us-east-1.amazonaws.com/dev/data`, integrated with the `SimpleDataQuery` Lambda function to provide data query functionality.

4.3 Frontend Deployment

The static frontend files (*index.html* and *script.js*) can be run locally using Python's built-in HTTP server: `python3 -m http.server 8000`

Alternatively, they can be hosted on any HTTPS-enabled web server.

The frontend fetches data from the hardcoded API Gateway URL and manages authentication status using `localStorage`. The login credentials are fixed as `admin/admin123`.

4.4 Verification Process

After launching *mqttToAWS.py*, verify terminal output for "Connected!" and log entries for data publication on the Raspberry Pi.

In the AWS DynamoDB console, check for new entries.

Open the frontend page to confirm that the charts are updating in real-time and that alert statuses are displayed.

If anomalies are detected, inspect the Lambda logs in CloudWatch and verify the certificate permissions on the device side.

5 User Manual

5.1 System Startup

Power On the Device: Connect the Raspberry Pi to a power source. The system will automatically boot, then run the *mqttToAWS.py* script.

Status Confirmation: Check the Raspberry Pi's LED matrix display:

- Solid Green: System is running normally and data is being uploaded.
- Scrolling Red Text: Temperature exceeds 50°C.
- Scrolling Blue Text: Humidity exceeds 80%

5.2 Accessing the Dashboard

Open *index.html* directly in a browser.

Login Authentication: On first access, enter the username `admin` and password `admin123`.

5.3 Interface Features

Real-Time Chart:

- Red Line: Temperature (°C). Turns red and displays an "ALERT" tag if it exceeds 50°C.

- Blue Line: Humidity (%). Turns blue and displays an "ALERT" tag if it exceeds 80%.
- Zooming: Click and drag the chart area to explore historical data in detail.

Status Panel:

- Green: Environment is normal.
- Red: Temperature alert (shows current value).
- Blue: Humidity alert (shows current value).
- Red-Blue Gradient: Both temperature and humidity exceed thresholds.

5.4 Alert Handling

Local Alert:

1. If the LED matrix displays scrolling text, check for ventilation or dehumidification needs.
2. Once the alert condition is resolved, the LED will automatically return to solid green.

Remote Notification:

1. The top status bar of the dashboard continuously shows active alerts.
2. Press F5 to manually refresh and retrieve the latest status.

5.5 Troubleshooting

Issue	Solution
Cannot log in	Confirm correct credentials (<code>admin/admin123</code>), clear browser cache.
No data in chart	Check device power and internet connection, wait 1–2 minutes.
Status bar shows "Connection Failed"	Verify the configured API Gateway URL is correct.
LED matrix not lit	Restart the Raspberry Pi and ensure <code>mqttToAWS.py</code> is running.

Table 1: Troubleshooting Common Issues