SAMS

Design MANUAL

Table of Contents

System Scope

Introduction & Background

In Sri Lanka, large-scale agricultural management, particularly in plantation sectors like palm oil, still relies heavily on **manual data collection**, **infrequent lab-based analysis**, and **delayed decision-making**. Traditional methods are not only time-consuming but also prone to errors, leading to ineffective irrigation, nutrient mismanagement, and reduced crop yield.

Furthermore, **laboratory testing of soil nutrients (NPK)** is expensive and slow, and there is often no centralized system for **monitoring real-time environmental conditions** like temperature, humidity, rainfall, and sunlight across multiple fields. This lack of visibility hinders timely interventions.

To address these challenges, our team developed the **Smart Agriculture Monitoring System (SAMS)**, a fully integrated solution combining **IoT-based field sensing**, **mobile and web applications**, and **real-time alerting and analytics** to optimize agricultural management practices.

Features of the System

The SAMS platform is composed of **two hardware setups** and **two software interfaces**; all connected via **Firebase Cloud** for real-time communication and control.

Hardware Components

a) Fixed Monitoring Unit

Mounted in the field for continuous environmental sensing.

- **Tipping Bucket Rain Gauge**: Measures rainfall accumulation in real time.
- **BH1750 Light Sensor**: Captures ambient light (Lux) levels for sunlight exposure analysis.
- **DHT11**: Monitors air **temperature and humidity**.

b) Portable Monitoring Unit

Carried by plantation staff for soil inspection at specific locations.

- **Soil Moisture Sensor**: Checks real-time soil water content.
- **Soil NPK Sensor** (RS485): Measures nutrient levels of **Nitrogen**, **Phosphorus**, and **Potassium**.
- **GPS Module**: Tags each reading with geolocation to match data with field sections.

Both setups are powered by **ESP32 microcontrollers** and support **Wi-Fi connectivity**, with offline fallback logic via SD cards and battery-friendly operation.

Software Interfaces

a) Mobile App (Flutter + Firebase)

Designed for field officers and plantation managers.

- **Login/Authentication System**
  Secure role-based login with Firebase Authentication.
- **All Fields Map View**
  Interactive maps showing plantation layout and section status.
- **Real-Time Data Panels**
  Draggable bottom panel displaying live readings (NPK, soil moisture)
- **Historical Data Charts**
  Graphs for trend analysis of each parameter by time and location.
- **Tree Health Detection Module**
  Upload tree images to detect health status using ML model (YOLOv8). Processed image and stats shown in app.
- **Alert Notifications**
  Automatic alerts for abnormal sensor readings (e.g., low moisture, high NPK).
- **User Profile Management**
  Edit profile, logout, and manage sessions through the app.

b) Web App

Similar interface to the mobile application.

Interfaces

- **Hardware - Software Interface:** Microcontrollers send sensor data to Firebase Cloud.
- **Human Interface:** Flutter-based mobile app and React-based web dashboard.
- **External Interfaces:** Firebase, ML-API and Google Maps JavaScript API

Major Software Functions

- Real-time data monitoring and visualization
- Historical data visualization
- Authentication and access control
- Tree analysis through image upload

Design Constraints

- Power-efficient sensor nodes (portable and fixed)
- Network limitations in rural plantation areas
- Firebase database read/write constraints
- Offline fallback consideration

Module Overview

Hardware Modules

The system is designed to monitor critical soil and environmental parameters, such as Nitrogen, Phosphorus, Potassium (NPK) levels, soil moisture, GPS location, and time, using a combination of sensor modules, microcontroller processing, and cloud integration.

The core controller used is an ESP32-based development board, which collects data from various sensor modules, processes it locally, and communicates with cloud services for remote monitoring and analysis. The system also includes a display module for local visualization and an SD card module for offline data logging.

This document covers the complete hardware setup, pin configuration, power design and interfacing between components. It serves as a reference for assembling, troubleshooting, and extending the system in future iterations.

- Sensor Node (Soil Moisture, NPK, Temperature, Humidity, Sun Light, Tipping Bucket)
- Microcontroller (ESP32)
- Power Supply (Rechargeable Battery)
- Communication Modules (Wi-Fi)

Software Modules

- Mobile App (Flutter): User Authentication, Dashboard, Charts, Field Navigation
- Web App (React): Admin Management, Tree Analysis, Visualization, Charts
- Backend (Firebase): Firestore, Authentication, Custom claims
- Google map API: Google map integration for plantation boundary display
- ML API: Tree detection and health assessment

Design Description

Software

Data Description

This section describes **what kind of data** our system handles, **how it's structured**, and **where it flows**.

Types of Data

1. **Soil Condition Sensor Data**:

    - Soil Moisture (%)
    - NPK Levels (mg/kg)
    - Timestamp and GPS coordinates

2. **Environmental Sensor Data**
    - Temperature (°C)
    - Humidity (%)
    - Rainfall (mm)
    - Sunlight/Lux (lux)
    - Timestamp and GPS coordinates

3. **Tree Image Analysis Data (from ML API):**
    - input_image_url: Original uploaded image
    - output_image_url: Image with tree annotations
    - tree_count: Number of trees detected
    - detection_summary: {Healthy: X, Unhealthy: Y }

Data Storage

- Stored in **Cloud Firestore** under structured documents.
- Images stored in **Google Cloud Storage.**
- Firestore has real-time syncing, enabling live dashboards.

Data Format Example

```
{
  "soilMoisture": 40,
  "nitrogen": 24,
  "phosphorous": 4,
  "potassium": 13,
  "timestamp": "2025-07-21T14:25:00Z",
  "geoPoint”:
    "lat": 6.9876,
    "lon": 80.7654
  }
}
```

Program Structure

Architecture Type: ***Layered Modular Architecture***

Our mobile application follows Layered Modular Architecture, which organizes the system into layers, each with clear responsibility and interaction pattern. This structure improves maintainability, scalability and code reusability.

Mobile App

1. Presentation Layer

`lib/presentation/`

- **Responsibility:** UI rendering and user interaction
- **Subfolders:**
  - `screens/` – Full-screen UI pages (e.g., HomeScreen, LoginScreen)
  - `widgets/` – Reusable UI components (e.g., OfflineBanner, buttons, Overlays)
  - `constants/` – centralizes app-wide values (e.g., colors, style)
- **Entry Point:** `main.dart` initializes Firebase, Providers, and routes.

2. Domain Layer

`lib/domain/`

- **Responsibility:** Business logic and core entities
- **Subfolders:**
  - `entities/` – Core app models like `SoliParameters`, `User`, `DeviceGroup`
  - `test/` – Unit and integration tests

- **Characteristics:**

    - Pure Dart classes (no Flutter dependency)
    - Defines contracts/interfaces used across layers
    - Can be easily tested and reused

3. Data Layer

`lib/data/`

- **Responsibility:** Communication with Firebase, APIs, and local storage
- **Subfolders:**
    - `models/` – Data Transfer Objects (DTOs) for Firebase and REST APIs
    - `repositories/` – Repos abstracting Firestore/REST logic (e.g., `SensorRepository`, `UserRepository`)
    - `services/` – Firebase/Auth services, HTTP clients, and helper functions

4. Core Layer

`lib/core/`

- **Responsibility:** App-wide configuration and utilities
- Contains:
    - `routes.dart` – Centralized route definitions

5. Integration Layer

- `firebase_options.dart`: Generated by Firebase CLI, holds Firebase config for initialization
- `main.dart`: Bootstraps the app, sets up Firebase, Providers, routes

Web app

1. **assets**: Static files like images and icons.

2. **components**: Reusable building blocks for the UI. (e.g., Estate Card, Estate Side Bar, Stat Card).

3. **layouts**: Common page structures and wrappers (e.g., Side Bar).

4. **lib**: Shared libraries and helper files. (e.g., Utils).

5. **pages**: Individual screens of the app. (e.g., Login Page, Dashboard, Landing Page).

6. **routes**: Navigation logic and route protection.

7. **services**: Logic for API calls and data handling. (e.g., Auth Services, Firestore Services).

8. **utils**: Utility functions and setup files (e.g. Firebase config).

Interface Within Structure

Firebase SDK Integration

Firebase SDK has been integrated in both Flutter(mobile) and React(web) apps.

This provides access to core Firebase services:

- Firestore - For structured real-time sensor data storage and retrieval
- Authentication - For sign up and login

These SDK methods have been implemented in the service layer for better reusability and maintainability.

Google Cloud Storage Integration

Used in both mobile and web apps to **store and serve images** used in ML analysis.

Real-time Listeners

Both web and mobile apps use Firestore's onSnapshot():

- Automatically listens to the changes in sensor data raw_readings collections.
- According to those changes, updates UI widgets like recent activity and weather data cards.
- It reduces the polling, request latency and improves the system responsiveness.

REST API Communication

The web app calls a custom ML API via 'fetch()'.

- **How does this work?**
    - Flutter app sends a `POST` request to the hosted API endpoint, Cloud Run.
    - Image is sent as multipart/form-data or base64.
    - API processes it using YOLOv8 model and returns:

        - Processed image URLs
        - Tree count
        - Detection summary (healthy and unhealthy trees)

```
{

  "input_image_url": "...",

  "output_image_url": "...",

  "tree_count": 102,

  "detection_summary": {

    "Healthy": 95,

    "Unhealthy": 7

  }

}
```

Those results are stored in Firestore and shown in the UI components.

Google Maps API Integration

Used in both mobile and web applications for geospatial visualization of whole plantations.

Displays the estate, section and field boundaries and locations of actual readings have been taken. (via markers)

Integrated using:

- `google_maps_flutter` (for mobile)
- `@react-google-maps/api` (for web)

Coordinate and boundary data are stored and retrieved from Firestore.

Authentication Interface

Firebase authentication has been used to ensure security across both platforms.

Custom claims are used to manage role-based access.

Access to certain screens and functions are role-dependent.

Software Setup and Architecture

Tree Health Detection concept used in mobile app : Upload tree image and receive detection results.

This section provides a comprehensive overview of the software stack and setup process for SAMS, covering both Flutter-based mobile app and React-based web app. The architecture integrates Firebase, Google Cloud Services and custom APIs for scalable, responsive monitoring systems.

Technology Stack

- Flutter: Cross-platform mobile application framework.
- React (Typescript): Web application frontend framework.
- Firebase: Auth, Firestore, and Cloud Functions.
- Google Cloud Storage: For storing ML image uploads and outputs.
- Node.js: For backend admin scripts (custom claims, GeoJSON uploads).
- Python: ML model for tree detection and health analysis.
- Docker: For containerization and consistent deployment.

Module Responsibilities

Mobile App (Flutter)

1. Authentication Module

- **Responsibilities:**
  - Handles login, signup, and logout
  - Validates user credentials
  - Manages session and Firebase Auth state
- **Used Technologies:** Firebase Authentication, firebase_auth package

2. Sensor Data Visualization Module

- **Responsibilities:**
  - Fetches real-time and historical sensor data from Firebase
  - Displays data in graphs, charts, and sensor cards
  - Supports section-wise (field-wise) filtering and viewing

- **Used Technologies:** Firestore, charts_flutter, flutter_chart_kit

3. Alert and Notification Module

- **Responsibilities:**
  - Listens to threshold breaches in Firebase
  - Triggers in-app or local notifications for critical events (e.g., low moisture)
  - Displays a list of recent alerts
- **Used Technologies:** Firebase Cloud Firestore, flutter_local_notifications

4. Tree Health Detection Module

- **Responsibilities:**
  - Allows users to upload tree images for analysis
  - Sends image to REST API endpoint running YOLOv5 model
  - Displays processed image and health statistics
- **Used Technologies:** HTTP Client, Firebase Storage, Swagger API

5. Map & GPS Visualization Module

- **Responsibilities:**
  - Shows plantation fields on a map
  - Fetches GPS-based sensor/device locations
  - Enables field-specific data tracking
- **Used Technologies:** Google Maps API, flutter_map, GPS module on device

6. User Management Module

- **Responsibilities:**
  - Allows profile updates and password changes
  - Manages role-based access (Admin, Field Officer)
  - Displays user-specific information and linked devices
- **Used Technologies:** Firebase Firestore, Firebase Auth

7. Navigation & Routing Module

- **Responsibilities:**
  - Manages screen transitions via routes
  - Controls bottom navigation and page routing
- **Used Technologies:** custom route management

8. Settings and Configuration Module

- **Responsibilities:**
  - Let admin configure threshold values (e.g., min moisture level)
  - Stores control logic for each field/section
  - Supports default device group configuration

- **Used Technologies:** Firestore config documents, in-app forms

9. State Management Module

- **Responsibilities:**
  - Maintains app-wide state using Providers
  - Handles authentication state, sensor data state, and command logic

- **Used Technologies:** provider, ChangeNotifier

Web App (React + TypeScript)

1. Authentication Module

- **Responsibilities**:
  - Handles login, signup via Firebase Auth
  - Guards routes and UI elements based on user role
  - Supports persistent sessions
- **Used Technologies**: Firebase Auth, React Context, Protected Routes

2. Sensor Dashboard Module

- **Responsibilities**:
  - Displays real-time sensor readings (lux, humidity, temperature)
  - Uses cards, tables, and charts
  - Syncs with Firestore updates dynamically
- **Used Technologies**: Firestore, Chart.js, Firestore onSnapshot

3. Tree Health Detection Module

- **Responsibilities**:
  - Accept image uploads for analysis
  - Sends images to ML API and displays result
  - Shows output image, tree count, and health stats
- **Used Technologies**: fetch, REST API, Google Cloud Storage, Firestore

4. Field and Estate Management Module

- **Responsibilities**:
  - Displays estate, sections, and fields using Firestore data
  - Allows navigation and selection to view detailed sensor readings

- **Used Technologies**: Firestore, React Tree/List Views

5. Map Visualization Module

- **Responsibilities**:
    - Renders plantation boundaries and GPS data on map
    - Displays markers for sensor locations
- **Used Technologies**: Google Maps API, GeoJSON, react-google-maps/api

6. Role-Based Access Module

- **Responsibilities**:
    - Renders content based on user role
    - Prevents unauthorized access to admin panels or data
- **Used Technologies**: Firebase Auth, Custom Claims, Route Guards

7. Analysis History Module

- **Responsibilities**:
    - Fetches and displays previous ML analysis results
    - Allows users to view past detections and statistics
- **Used Technologies**: Firestore, Timestamp filtering, ListView

Data Flow Overview

From Sensors to Apps

1. ESP32 reads sensor (e.g.: soil moisture, NPK levels)
2. Sends them via MQTT to Firebase Firestore
3. Firestore stores each reading under `/states/{}/sections/{}/fields/{}/readings`
4. Mobile and web apps are used onSnapshot() to listen to the real time updates and display them in UI components.

Image Analysis Flow

Admin uploads an image via the web dashboard.
Image is sent to ML API via POST request
API processes it, returns:

- tree_count
- health_summary
- output_img_url

Web app stores result in Firestore and Firebase Storage

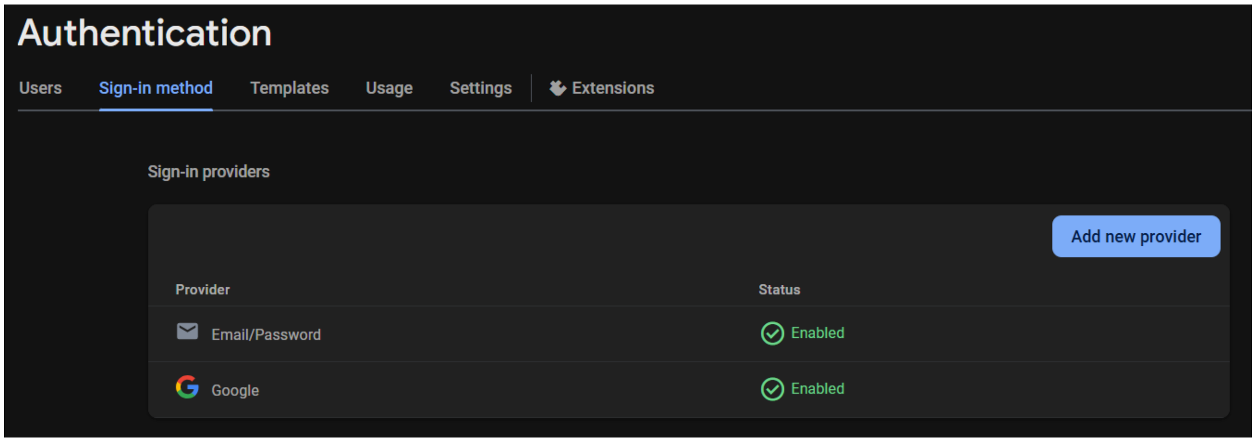
Google Maps

Google Maps overlays are rendered using boundary data fetched from Firestore under each of the fields.

Setup Instructions

Firebase Configurations

- **Firebase Auth:** Secures the app from unauthorized access.
  - Email/Password Login
  - Google Login



Sign-in providers

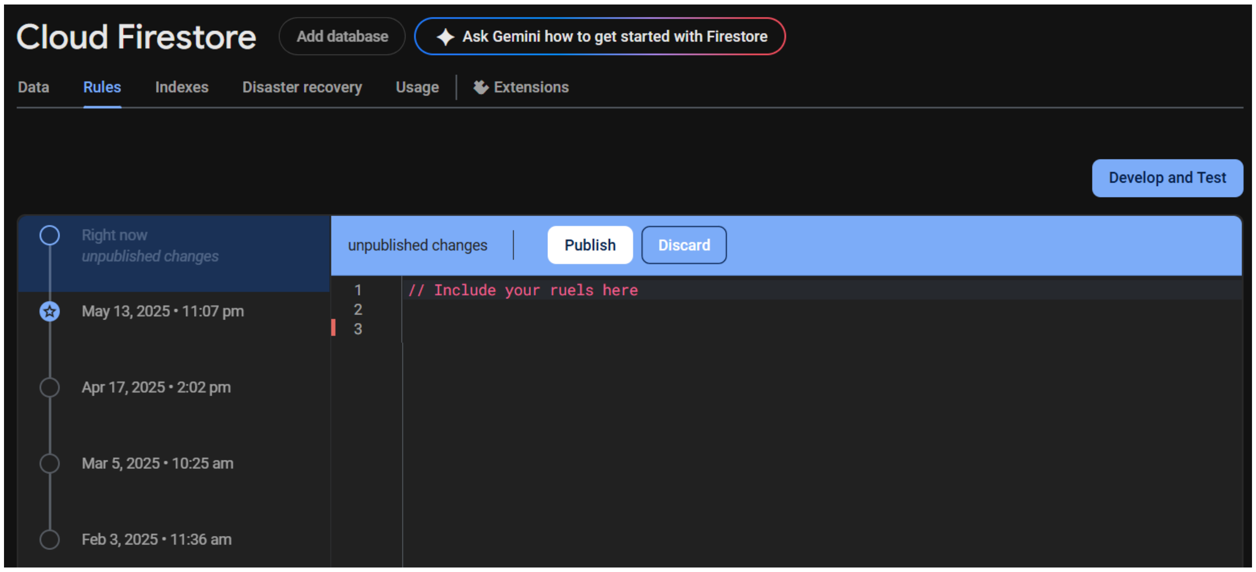
Provider	Status
Email/Password	Enabled
Google	Enabled

- **Firestore Structure:** Stores the sensor raw readings, user roles, estate structures according to the sections and fields and ML analysis results.

```
/states/{stateId}
  /sections/{sectionId}
    /fields/{fieldId}
      /readings/{timestamp}

/raw_readings/{readingId}

/raw_rain_data/{readingId}

/lux_level/{readingId}

/tree_detection/{analysisId}
```

```
/users/{email} → {role: 'admin'/'user'}

/config/{thresholds}
```

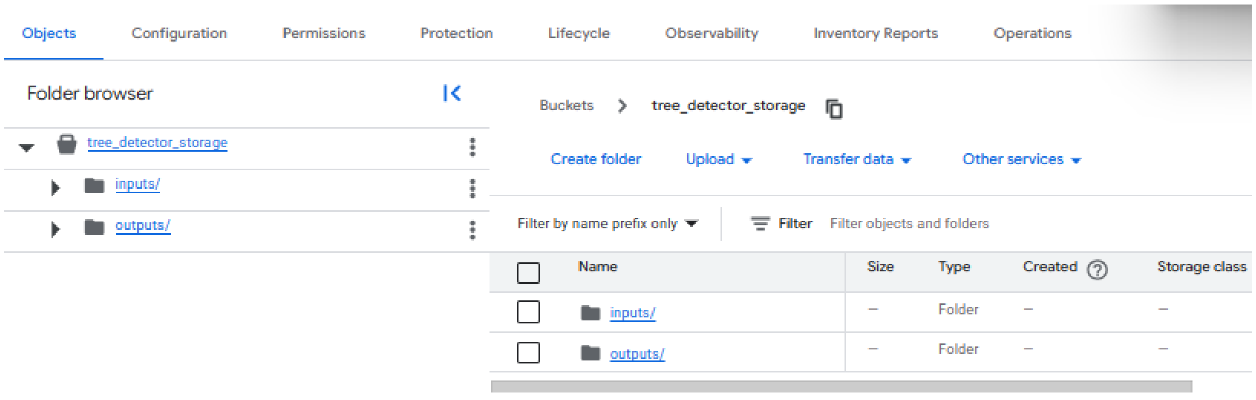
- **Firebase Security Rules:** Ensures the data security in Firestore.



```
// Include your ruels here
```

Google Cloud Storage Integration

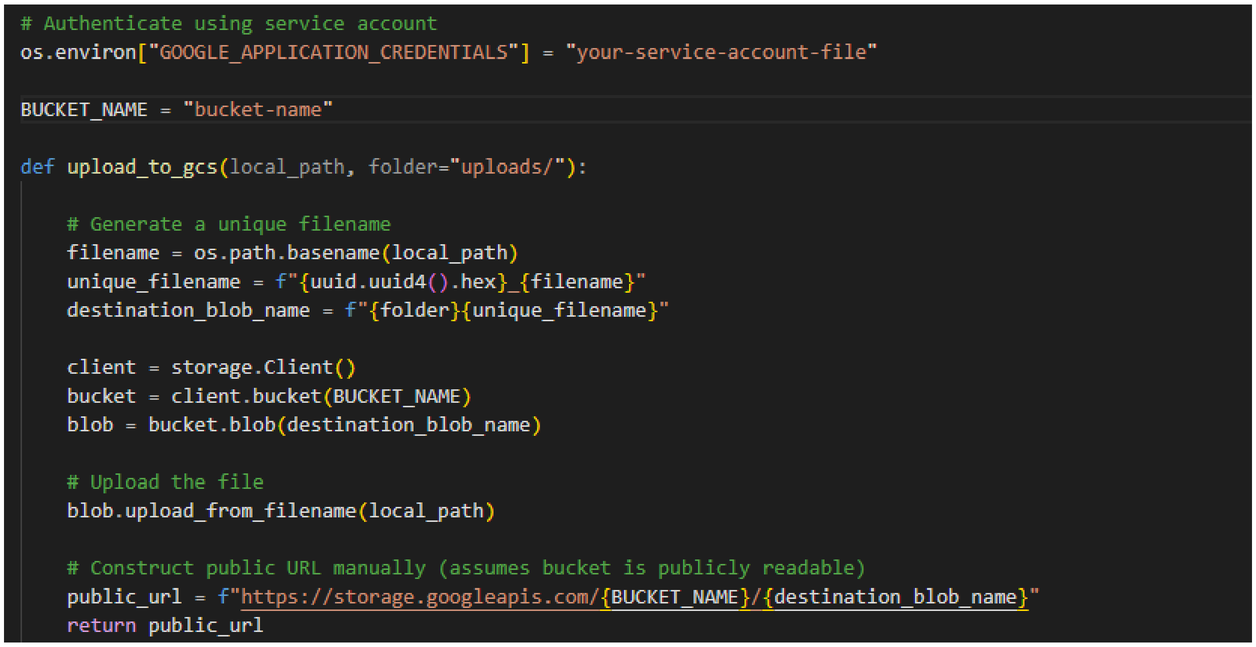
Google Cloud Storage is used for Tree Count and Tree Health detecting function. All the original images and Outputs from the tree health system are managed in there.



Name	Size	Type	Created	Storage class
inputs/	—	Folder	—	—
outputs/	—	Folder	—	—

To run the Machine learning module scripts, Firebase Admin SDK must be initialized using a service account.

Download the Firebase service account from the Firebase console.
Change below lines.



```python
# Authenticate using service account
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "your-service-account-file"

BUCKET_NAME = "bucket-name"

def upload_to_gcs(local_path, folder="uploads/"):

    # Generate a unique filename
    filename = os.path.basename(local_path)
    unique_filename = f"{uuid.uuid4().hex}_{filename}"
    destination_blob_name = f"{folder}{unique_filename}"

    client = storage.Client()
    bucket = client.bucket(BUCKET_NAME)
    blob = bucket.blob(destination_blob_name)

    # Upload the file
    blob.upload_from_filename(local_path)

    # Construct public URL manually (assumes bucket is publicly readable)
    public_url = f"https://storage.googleapis.com/{BUCKET_NAME}/{destination_blob_name}"
    return public_url
```

Mobile App Setup (Flutter)

```
// Clone repo and open project
flutter pub get



// Configure Firebase (GoogleServices.json for Android)
{
  "flutter": {
    "platforms": {
      "android": {
        "default": {
          "projectId": "project_id", // Replace with your project id
          "appId": "app_id", // Replace with your app id
          "fileOutput": "android/app/google-services.json"
        }
      },
      "dart": {
        "lib/firebase_options.dart": {
          "projectId": "project_id", // Replace with your project id
          "configurations": {
            "android": "app_id_for_android", // Replace with your app id
for Android
            "ios": "app_id_for_iOS", // Replace with your app id for iOS
          }
        }
      }
    }
  },
}


// Run the app
flutter run
```

For Web App (React)

```
// Clone repo and install packages
npm install

// Add your Firebase config in firebase.ts

import { initializeApp } from "firebase/app";
import { getAuth } from "firebase/auth";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = {
  apiKey: "AIzaSyAJsr2x9fVSTZLjk_h-yjLjJEY5YlRgJFs",
  authDomain: "auth_domain", // Replace with your auth domain
  databaseURL: "database_url", // Replace with your database URL
  projectId: "project_id", // Replace with your project id
  storageBucket: "storage_bucket_id", // Replace with your storage bucket
id
```

```
  messagingSenderId: "msg_sender_id", // Replace with your messaging
sender id
  appId: "app_id", //Replace with your app id
  measurementId: "measurement_id" //Replace with your measurement id
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const auth = getAuth(app);
const db = getFirestore(app);

export { auth, db };


// Start dev server
npm run dev
```

Firebase Deployment

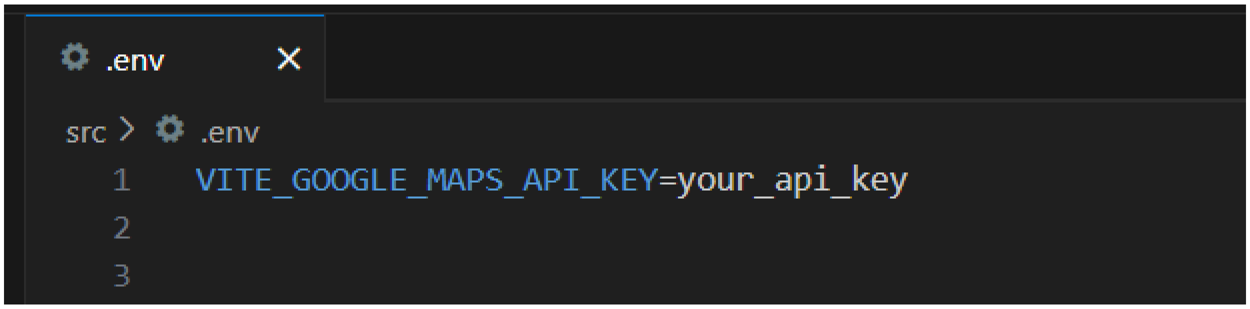
```
firebase login
firebase init
firebase deploy
```

Google Map API Integration

Google Maps API is used for geospatial visualization of estate boundaries and exact locations that readings have been taken.

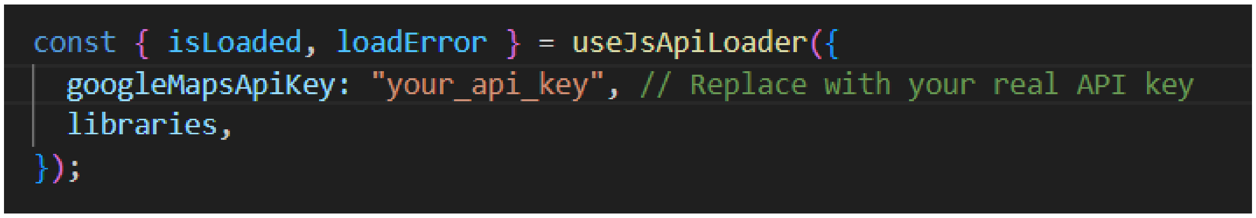
For Web App (React):

Use an .env file and reference it in your code.



```
VITE_GOOGLE_MAPS_API_KEY=your_api_key
```

Load it in your app as below (In the map.tsx file):

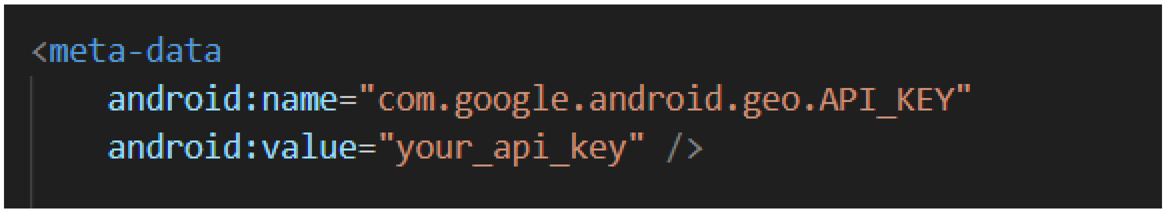


```
const { isLoaded, loadError } = useJsApiLoader({
  googleMapsApiKey: "your_api_key", // Replace with your real API key
  libraries,
});
```

For Mobile App (Flutter):

1. Android

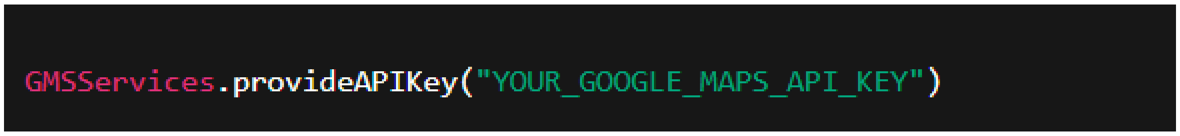
In android/app/src/main/AndroidManifest.xml , add:



```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="your_api_key" />
```

2. iOS

In `ios/Runner/AppDelegate.swift` , add:



```
GMSServices.provideAPIKey("YOUR_GOOGLE_MAPS_API_KEY")
```

Admin Scripts and Role Management

In this section, we are considering how to configure the roles and store data related to estates in Firestore at the initial stage. For that Node.js scripts and Firebase Admin SDK are used to manage backend tasks that are not handled through the frontend UI.

Firebase Admin SDK Setup

To run the backend scripts, Firebase Admin SDK must be initialized using a service account.

Download the Firebase service account from the Firebase console.
Change below lines in the related files under \flutter_application_1\functions\admin

```
const serviceAccount = require("path_service_account"); // Replace with
your service account path

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
});
```

Uploading Estate data to Firestore

Change below lines in \flutter_application_1\functions\admin\uploadEstateData

```
async function storeFieldData() {
  const fieldName = "Field NKD Main-03"; // Replace With your actual field
name

  // Example GeoJSON object (could be loaded from file or request)
  const geoJson = {
    type: "FeatureCollection",
    features: [
      {
        type: "Feature",
        properties: {},
        geometry: {
          coordinates: [
            [
            [ 80.33191365201179,6.167046263564487], // Replace with your
actual coordinates
            [ 80.3317643463239,6.166997344021411],
            [ 80.33155421239275,6.166931369946781],
            [ 80.33137172766385,6.1669148764278106],
            [ 80.33116159373401,6.1670193353769065],
            [ 80.33104546708813,6.167134789981375],
            [ 80.33191365201179,6.167046263564487]
            ],
          ],
          type: "Polygon",
        },
      },
    ],
  };

  // Convert coordinates to GeoPoints (GeoJSON uses [lng, lat])
  const polygonCoordinates =
geoJson.features[0].geometry.coordinates[0].map(
    ([lng, lat]) => new admin.firestore.GeoPoint(lat, lng));

  const fieldData = {
    fieldName,
    boundary: {
      type: "Polygon",
      coordinates: polygonCoordinates,
    },
  };

  // Store in Firestore (adjust path as needed)
  const fieldRef = firestore
    .collection("states")
    .doc("state2") // Replace With your actual estate name
    .collection("sections")
    .doc("NKD Main")  // Replace With your actual section name
    .collection("fields")
    .doc("field-03"); // Replace With your actual field id

  await fieldRef.set(fieldData);

  console.log("Field data saved successfully.");
}
```

Run the command:

```
npx ts-node uploadEstateData.ts
```

Hardware

To address different monitoring needs, the system is divided into two primary hardware units: the **Portable Device** and the **Fixed Device**.

Portable Device

The Portable Device is a mobile, battery-powered unit used by field workers or farmers to manually gather sensor data at specific points in a plantation. It integrates multiple sensors, including:

- **NPK Soil Sensor Module** for measuring nitrogen, phosphorus, and potassium.

- **Capacitive Soil Moisture Sensor** for measuring soil water content.

- **GPS Module (Neo-6M)** to record the geographic location of readings.
- **Real-Time Clock (RTC)** for time-stamping data when Wi-Fi is not available.
- **SD Card Module** for offline data logging.
- **LCD Module (SPI-based)** for local display of readings.

The device is built around an ESP32 board and the user can later upload the collected data to the cloud once internet connectivity is available.

Fixed Device

The Fixed Device is a stationary setup designed to continuously monitor environmental parameters in real-time. It remains connected to power and Wi-Fi and periodically sends sensor data to the cloud database (Firestore) for long-term tracking of the field.

Key components include:

- **Tipping Bucket** for measuring rain gauge.
- **Sunlight Intensity Sensor (BH1750)**
- **Temperature and Humidity Sensor (DHT22)**

By separating the system into portable and fixed units, the design addresses both point-based spot checking and long-term area-based monitoring, enhancing the flexibility and efficiency of agricultural data collection.

Main Microcontroller

Portable Device

The **ESP32 DEV KIT V1** microcontroller has been chosen for portable device, because of following main features,

- Built-in Wi-Fi & Bluetooth for connectivity.
- Low power consumption.
- Sufficient processing power & memory.
- Low latency
- High throughput

Fixed Device

For the fixed device **ESP32 8266 Node MCU** has been chosen.

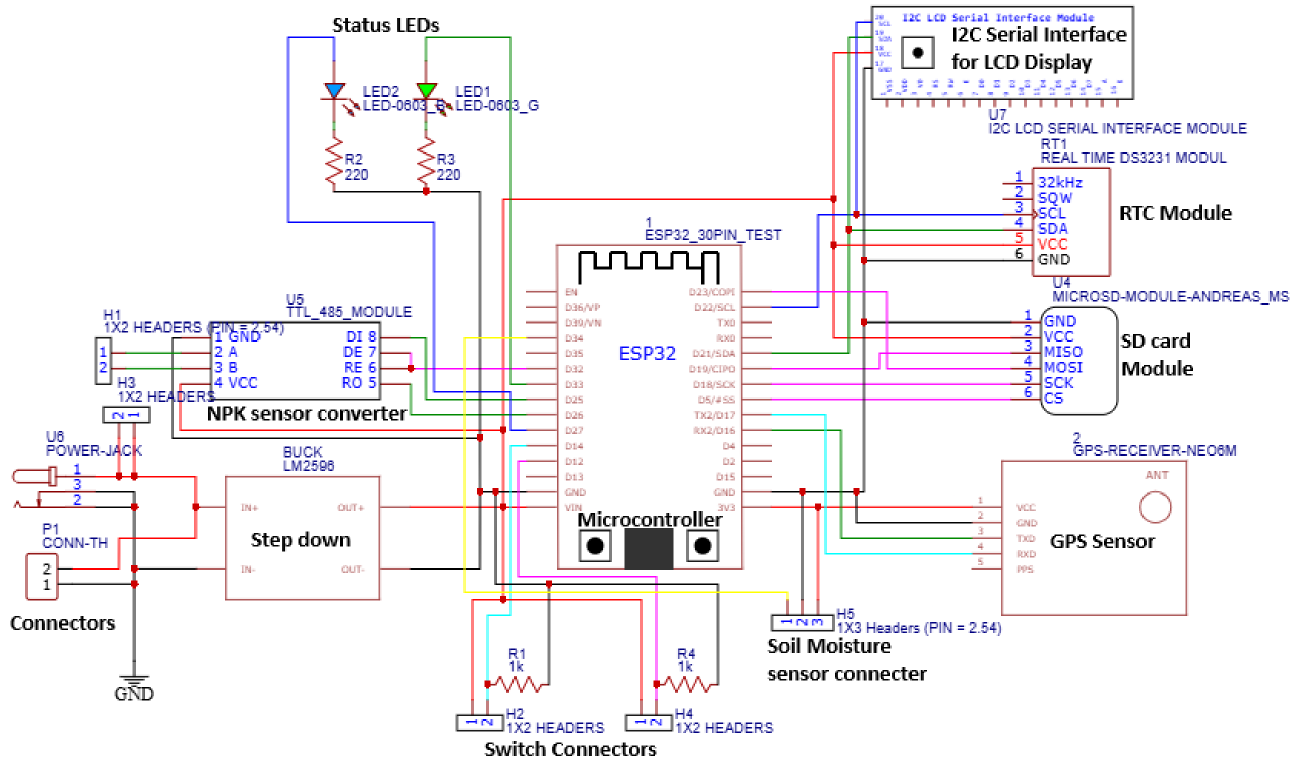
- Built-in Wi-Fi for wireless communication and IoT integration
- Low power consumption, especially with deep sleep mode
- Adequate processing power & memory for lightweight embedded applications
- Moderate latency, suitable for real-time device control
- Decent throughput for small data packets and sensor updates

ESP8266 doesn’t have Bluetooth, so it uses slightly less power.

Hardware Setup and Architecture

(How to integrate these modules)

Portable device



The above diagram shows the hardware setup of the Portable device.

Power Supply Configuration

The portable device is powered by **four** rechargeable lithium-ion batteries, each rated at **3.7V**, connected in series to provide a combined voltage of approximately 14.8V. This configuration is essential to meet the operational requirements of the **NPK sensor**, which requires a minimum input voltage of **12V** for proper functionality.

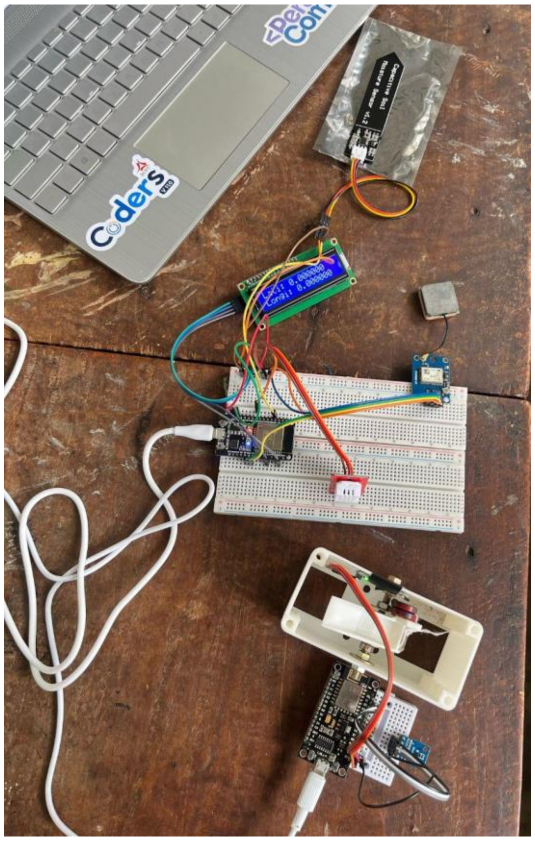
To supply appropriate voltage levels to other components:

- A Step-Down Voltage Regulator (LM2596) is employed to reduce the 14.8V input to a stable 5V output, which powers the ESP32 DEV KIT V1 microcontroller.
- The GPS module and soil moisture sensor are powered via the **3.3V** output provided by the ESP32 itself, ensuring compatibility with their operating voltage requirements.

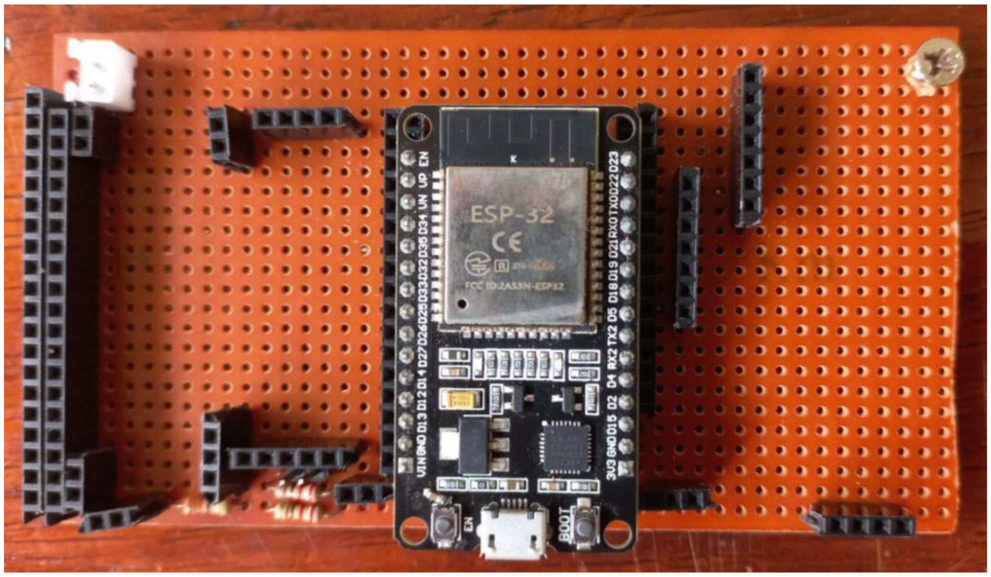
Module	Voltage Requirement	Power Source
NPK Sensor	≥ 12V	Directly from 4 × 3.7V batteries (≈14.8V)
ESP32 DEV KIT V1	5V	Output from LM2596 Step-Down Regulator
I2C Serial Interface	5V	Output from LM2596 Step-Down Regulator
SD card Module	5V	Output from LM2596 Step-Down Regulator
RTC Module	5V	Output from LM2596 Step-Down Regulator
NPK Sensor Converter	5V	Output from LM2596 Step-Down Regulator
GPS Sensor	3.3V	3.3V output pin on ESP32
Soil Moisture Sensor	3.3V	3.3V output pin on ESP32

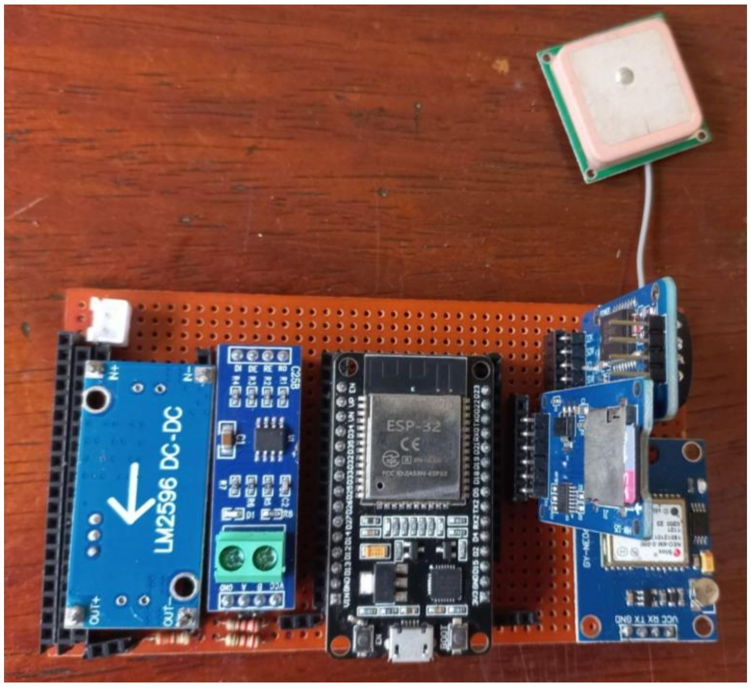
Initial Implementation

Using a bread board first we can implement the prototype version.



Then implement in the dot board.

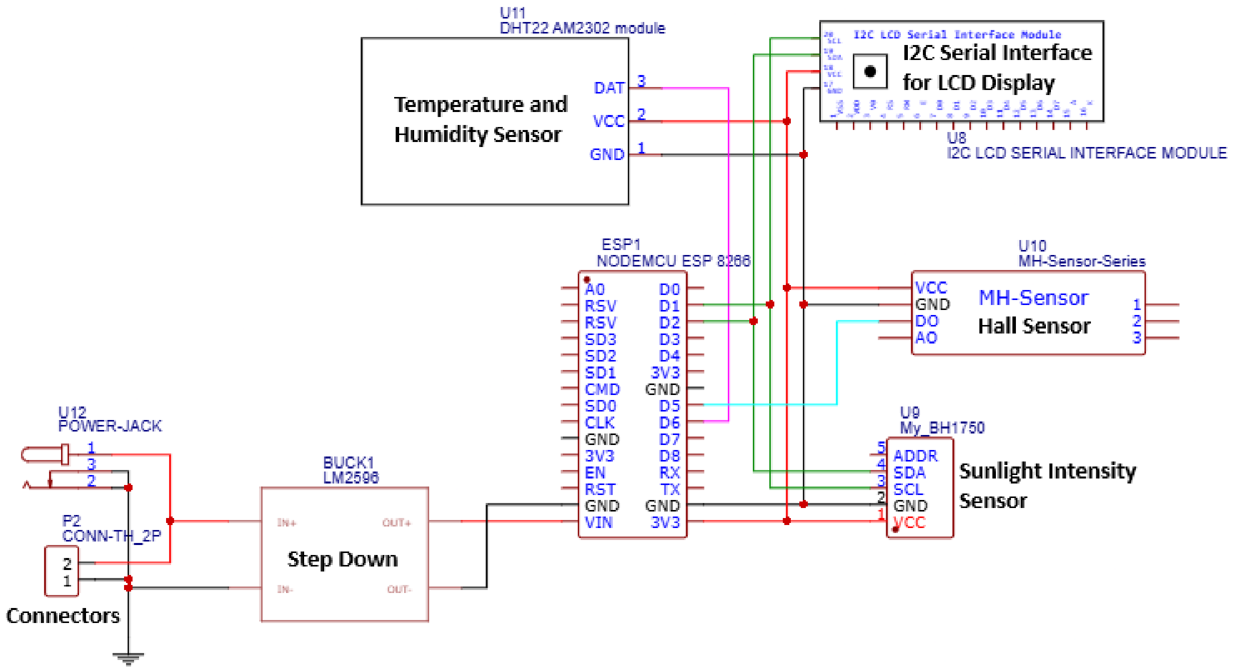




Final Implementation

If dot board also working fine, we can go for a PCB design.

Fixed device


U11
DHT22 AM2302 module
Temperature and Humidity Sensor
DAT 3
VCC 2
GND 1
I2C LCD Serial Interface Module
I2C Serial Interface for LCD Display
U8
I2C LCD SERIAL INTERFACE MODULE
ESP1
NODEMCU ESP 8266
A0 RSV RSV SD3 SD2 SD1 CMD SD0 CLK GND 3V3 EN RST GND VIN
D0 D1 D2 D3 D4 3V3 GND D5 D6 D7 D8 RX TX GND 3V3
U10
MH-Sensor-Series
VCC GND DO AO
MH-Sensor Hall Sensor
U9
My_BH1750
ADDR SDA SCL GND VCC
Sunlight Intensity Sensor
U12
POWER-JACK
P2
CONN-TH_2P
Connectors
BUCK1
LM2596
IN+ OUT+ IN- OUT-
Step Down

The above diagram shows the hardware setup of the Fixed device.

Power Supply Configuration

The fixed device is powered either by **two** rechargeable lithium-ion batteries, each rated at 3.7V, connected in series to supply approximately **7.4V**, or by an external regulated power source exceeding 5V. This voltage is fed into a Step-Down Voltage Regulator (LM2596) to deliver stable power outputs required by the various components of the system.

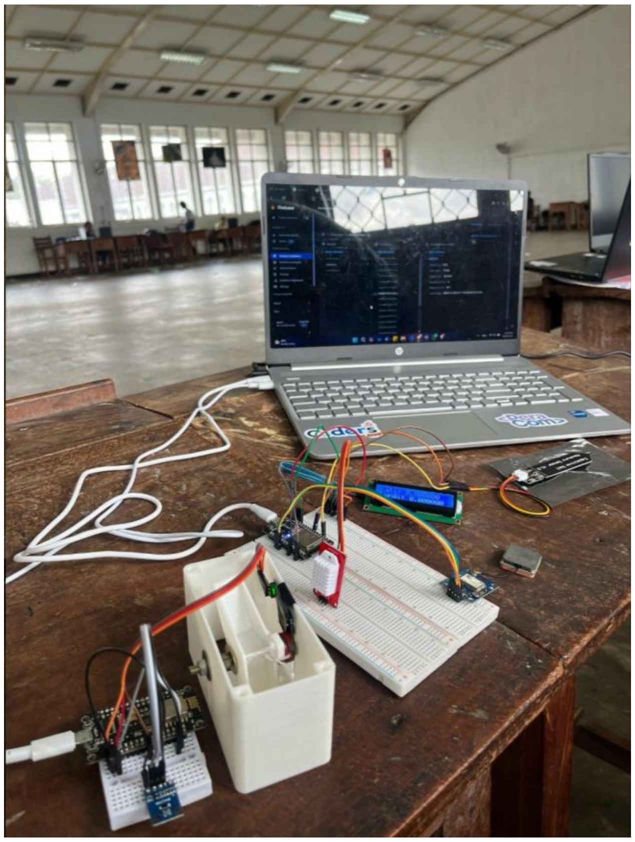
- The **NodeMCU ESP8266** microcontroller receives 5V via the regulator's output, ensuring reliable system operation.
- Peripheral modules such as the **Hall effect sensor, Sunlight sensor (BH1750), and Temperature and humidity sensor (DHT22)** are powered through the 3.3V pin of the NodeMCU, matching their voltage specifications and protecting them from overvoltage.
- The LCD module, which requires higher voltage, is powered directly from the regulated 5V output of the LM2596.

This arrangement ensures efficient distribution of power while maintaining voltage integrity across all connected components. The use of a step-down regulator simplifies voltage management and enhances safety and system stability.

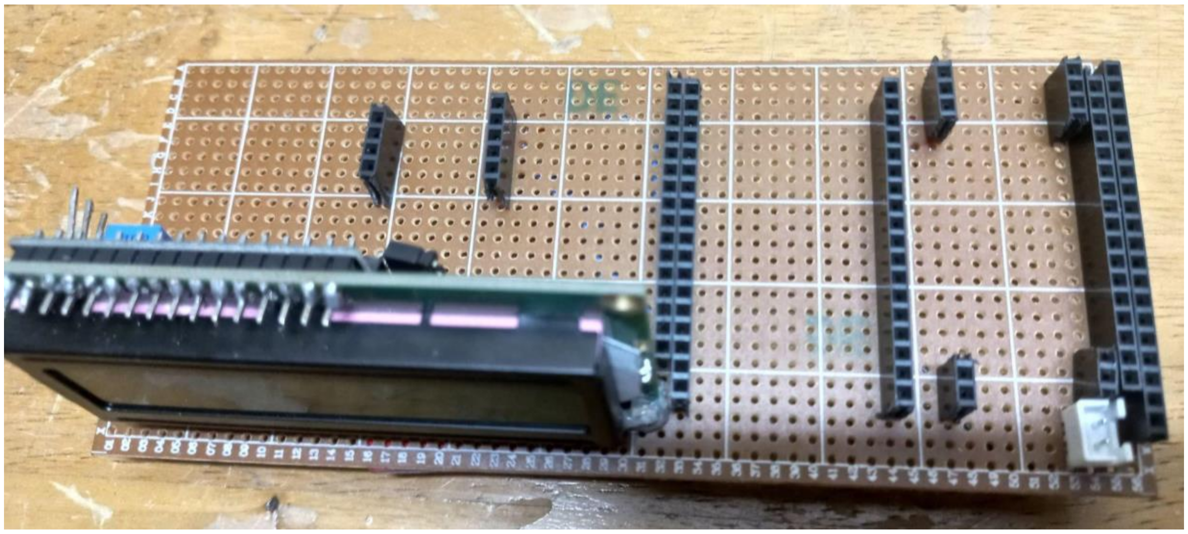
Module	Voltage Requirement	Power Source
NodeMCU ESP8266	5V	Output from LM2596 Step-Down Regulator
I2C Serial Interface	5V	Output from LM2596 Step-Down Regulator
Hall Sensor	3.3V	3.3V output pin on NodeMCU
Temperature and Humidity Sensor	3.3V	3.3V output pin on NodeMCU
Sunlight Intensity Sensor	3.3V	3.3V output pin on NodeMCU

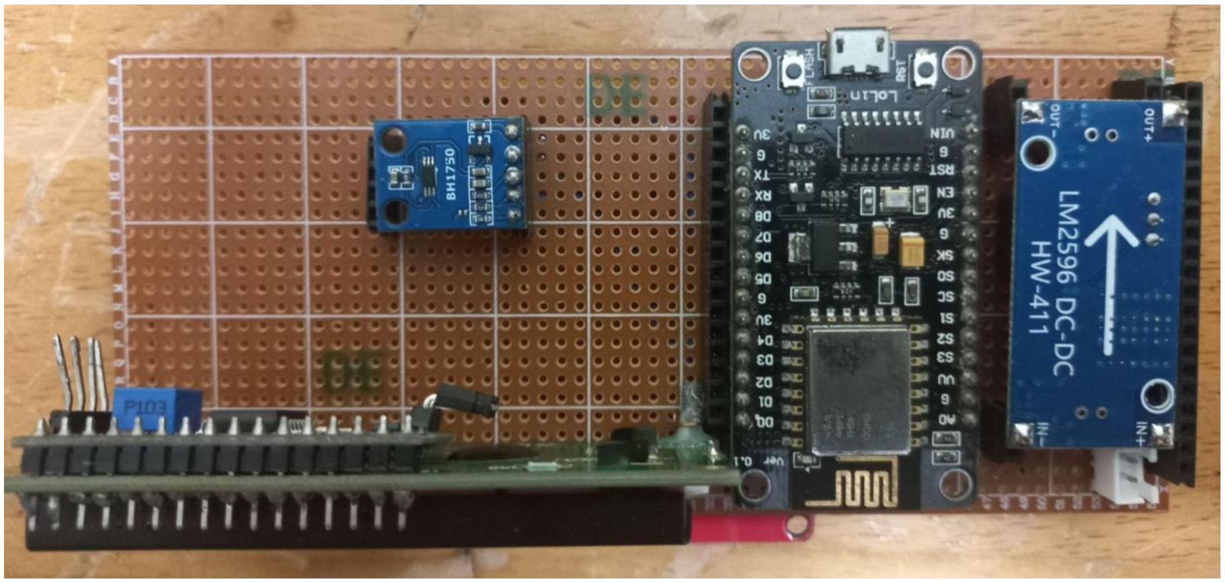
Initial Implementation

Using a bread board first we can implement the prototype version.



Then implement in the dot board.



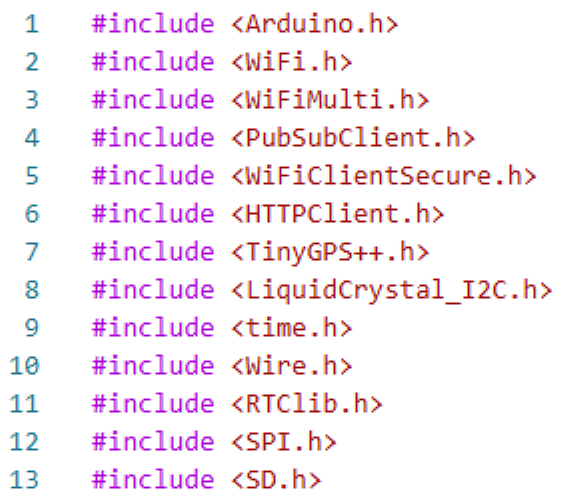


Final Implementation

If dot board also working fine, we can go for a PCB design.

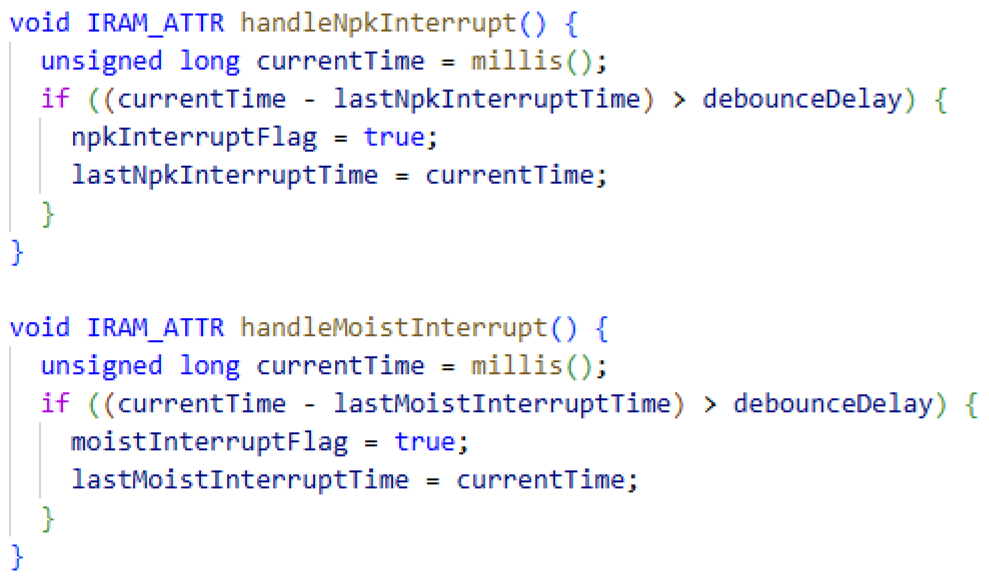
Code Implementation

The firmware for the device was developed using **PlatformIO**, leveraging the **Arduino framework** for simplified integration and cross-platform compatibility. This setup enables efficient development, debugging, and deployment of embedded applications. A suite of external libraries was utilized to interface with hardware modules and cloud services, ensuring robust functionality and minimizing development time.



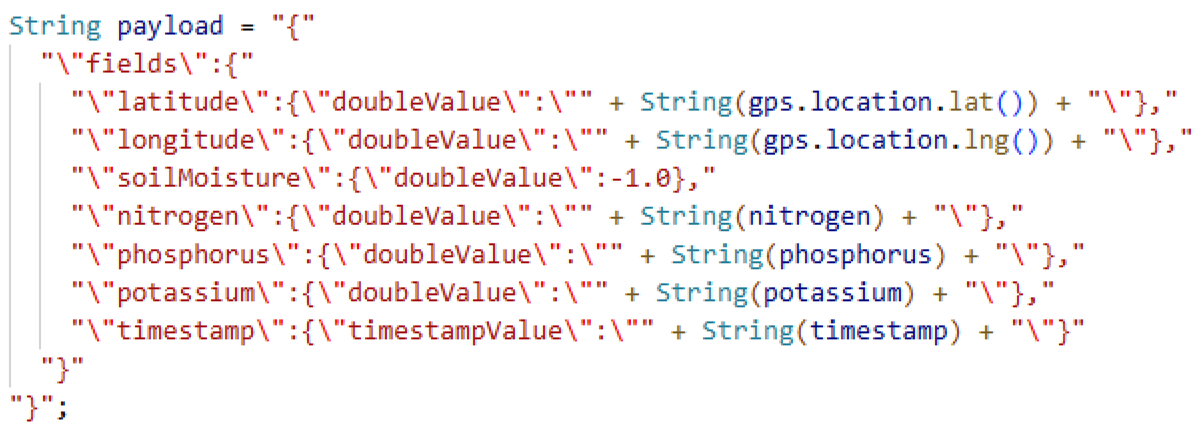
```
#include <Arduino.h>
#include <WiFi.h>
#include <WiFiMulti.h>
#include <PubSubClient.h>
#include <WiFiClientSecure.h>
#include <HTTPClient.h>
#include <TinyGPS++.h>
#include <LiquidCrystal_I2C.h>
#include <time.h>
#include <Wire.h>
#include <RTClib.h>
#include <SPI.h>
#include <SD.h>
```

To enhance responsiveness and provide a smooth user interface, **hardware interrupts** were implemented for button interactions. Unlike traditional polling methods, which consume processing time and can introduce latency, interrupt-driven input handling ensures **instantaneous recognition of user actions**. This design choice significantly improves the overall usability and real-time performance of the system, contributing to a more intuitive and user-friendly experience.



```
void IRAM_ATTR handleNpkInterrupt() {
  unsigned long currentTime = millis();
  if ((currentTime - lastNpkInterruptTime) > debounceDelay) {
    npkInterruptFlag = true;
    lastNpkInterruptTime = currentTime;
  }
}

void IRAM_ATTR handleMoistInterrupt() {
  unsigned long currentTime = millis();
  if ((currentTime - lastMoistInterruptTime) > debounceDelay) {
    moistInterruptFlag = true;
    lastMoistInterruptTime = currentTime;
  }
}
```

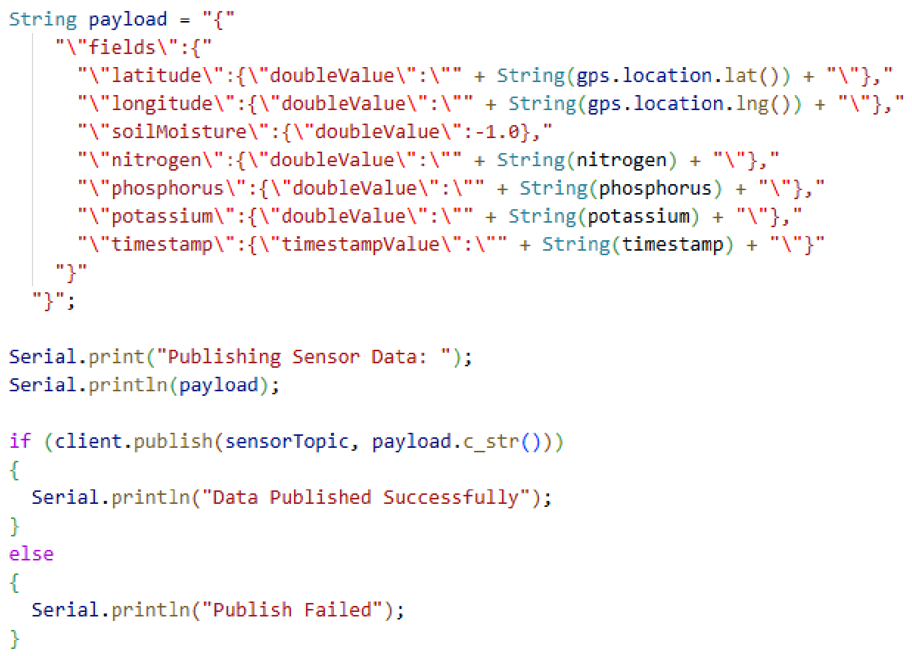
The payload structure is well-organized, modular, and scalable, making it ideal for handling multi-sensor data in a consistent format. It uses clearly defined fields with proper data typing (doubleValue and timestampValue), which aligns perfectly with cloud database formats like Firestore. Overall, it provides a reliable foundation for secure, structured, and extensible data exchange between hardware and cloud services.



```
String payload = "{"
  "\"fields\":{"
    "\"latitude\":{\"doubleValue\":\"" + String(gps.location.lat()) + "\"},"
    "\"longitude\":{\"doubleValue\":\"" + String(gps.location.lng()) + "\"},"
    "\"soilMoisture\":{\"doubleValue\":-1.0},"
    "\"nitrogen\":{\"doubleValue\":\"" + String(nitrogen) + "\"},"
    "\"phosphorus\":{\"doubleValue\":\"" + String(phosphorus) + "\"},"
    "\"potassium\":{\"doubleValue\":\"" + String(potassium) + "\"},"
    "\"timestamp\":{\"timestampValue\":\"" + String(timestamp) + "\"}"
  "}"
"}";
```

Data Binding to Backend

The firmware leverages the MQTT protocol to enable lightweight, efficient communication between the ESP32 microcontroller and a cloud-based broker. The **PubSubClient** library is used in conjunction with **WiFiClientSecure** to establish a secure connection over **TLS/SSL** on port 8883, ensuring encrypted data transmission.



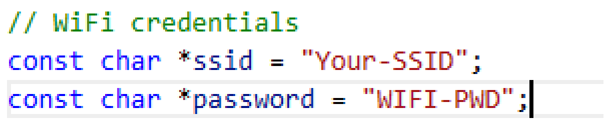
```
String payload = "{"
  "\"fields\":{"
    "\"latitude\":{\"doubleValue\":\"" + String(gps.location.lat()) + "\"},"
    "\"longitude\":{\"doubleValue\":\"" + String(gps.location.lng()) + "\"},"
    "\"soilMoisture\":{\"doubleValue\":-1.0},"
    "\"nitrogen\":{\"doubleValue\":\"" + String(nitrogen) + "\"},"
    "\"phosphorus\":{\"doubleValue\":\"" + String(phosphorus) + "\"},"
    "\"potassium\":{\"doubleValue\":\"" + String(potassium) + "\"},"
    "\"timestamp\":{\"timestampValue\":\"" + String(timestamp) + "\"}"
  "}"
"}";

Serial.print("Publishing Sensor Data: ");
Serial.println(payload);

if (client.publish(sensorTopic, payload.c_str()))
{
  Serial.println("Data Published Successfully");
}
else
{
  Serial.println("Publish Failed");
}
```

For MQTT you can start from setup in:

- Sign up for HiveMQ Cloud or EMQX Cloud
- Create a free cluster
- Set up credentials and topic permissions

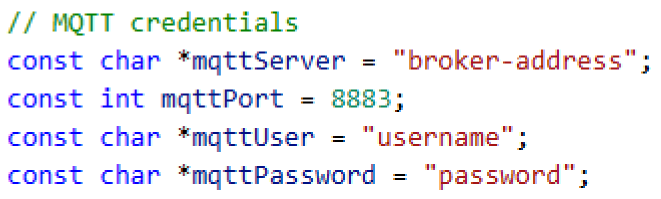
In the code implementation,

1. Update Wi-Fi Credentials (ssid and password)



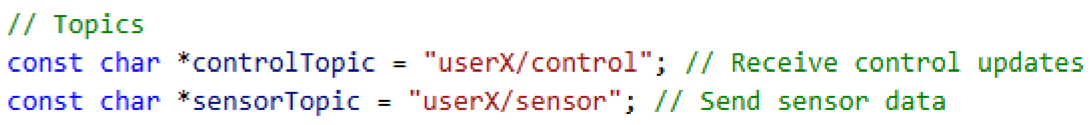
```
// WiFi credentials
const char *ssid = "Your-SSID";
const char *password = "WIFI-PWD";
```

2. Change MQTT Broker Information (mqttServer, mqttPort, mqttUser, and mqttPassword) to use their own HiveMQ Cloud instance or broker service.



```
// MQTT credentials
const char *mqttServer = "broker-address";
const int mqttPort = 8883;
const char *mqttUser = "username";
const char *mqttPassword = "password";
```

3. Edit Topic Names (controlTopic and sensorTopic) for unique topics prevent conflicts with other clients.



```
// Topics
const char *controlTopic = "userX/control"; // Receive control updates
const char *sensorTopic = "userX/sensor"; // Send sensor data
```

By now you have configured MQTT on your own.

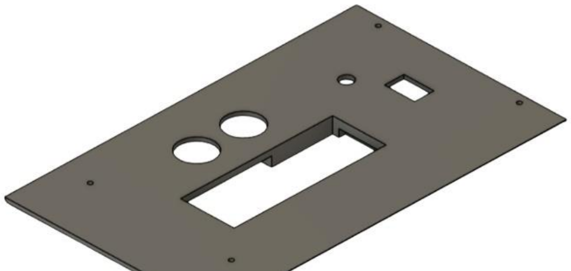
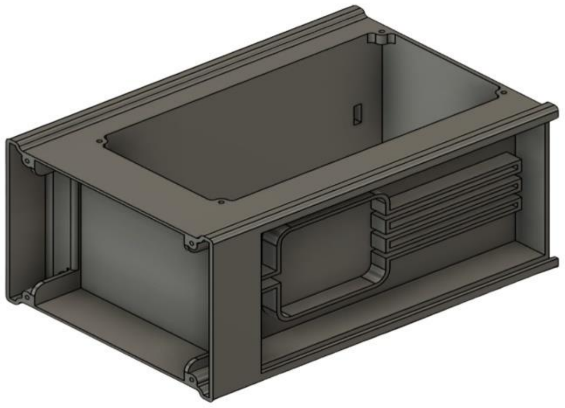
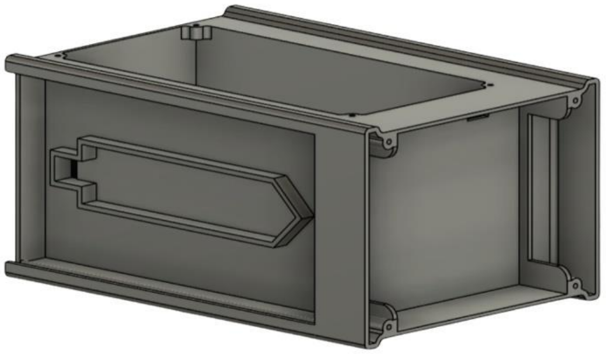
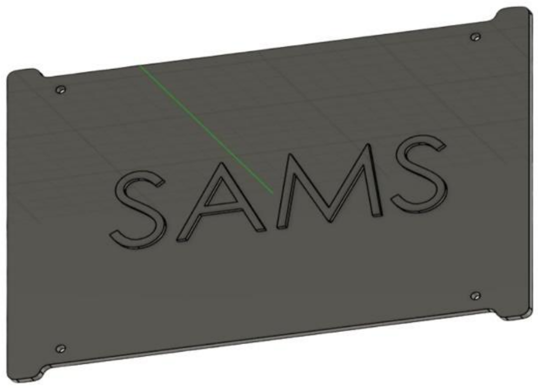
Hardware side Security

To protect user data and prevent unauthorized access, the code employs several important security measures:

Security Feature	**Description**
SSL/ TLS Encryption	WiFiClientSecure enables encrypted communication over port 8883.
Authentication	MQTT credentials (mqttUser, mqttPassword) are used for client identity.
Broker Privacy	Hosted on HiveMQ Cloud, which isolates each client's messaging stream.
Fail-Safe Design	Includes LED indicators and reconnection logic to mitigate network issues.

3D design

The hardware design includes portable sensor units equipped with soil condition sensors and microcontrollers, enabling real-time data collection from various field locations.

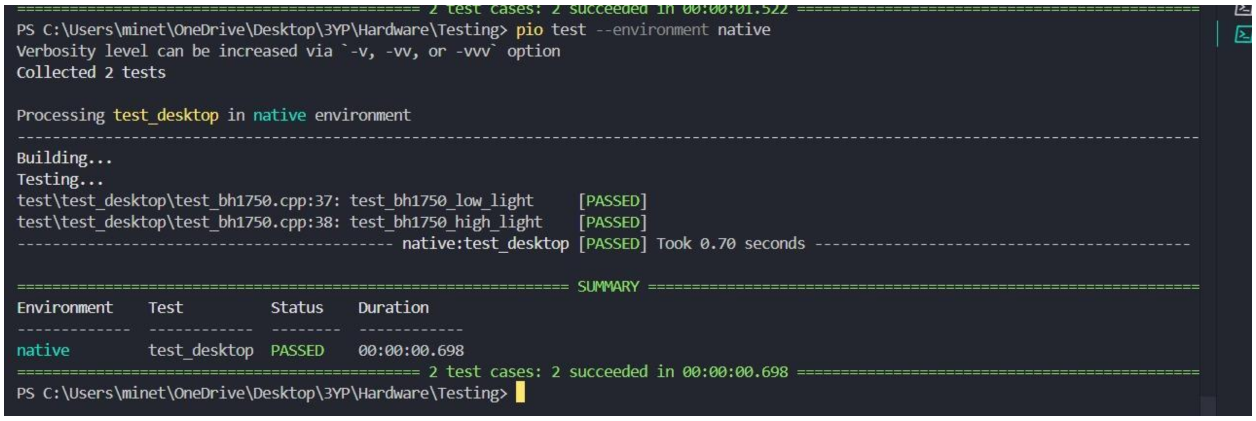





Testing Strategies

Hardware

Unit tests

Using **Unity Framework** we can test the modules one by one.



```
============================================ 2 test cases: 2 succeeded in 00:00:01.522 ============================================
PS C:\Users\minet\OneDrive\Desktop\3YP\Hardware\Testing> pio test --environment native
Verbosity level can be increased via `-v, -vv, or -vvv` option
Collected 2 tests

Processing test_desktop in native environment
------------------------------------------------------------------------------------------------------------------------------------
Building...
Testing...
test\test_desktop\test_bh1750.cpp:37: test_bh1750_low_light     [PASSED]
test\test_desktop\test_bh1750.cpp:38: test_bh1750_high_light    [PASSED]
------------------------------------------- native:test_desktop [PASSED] Took 0.70 seconds -------------------------------------------

=============================================================== SUMMARY ===============================================================
Environment    Test          Status    Duration
-------------  ------------  --------  ------------
native         test_desktop  PASSED    00:00:00.698
============================================ 2 test cases: 2 succeeded in 00:00:00.698 ============================================
PS C:\Users\minet\OneDrive\Desktop\3YP\Hardware\Testing>
```

Afterwards step by step Integration tests and End to end test can be done.

Software

Unit Testing

Unit testing has been done to ensure the correctness of individual components in isolation.

Mobile (Flutter):

Run unit tests to ensure that the UI elements and widgets are working properly.

Tooling: `flutter_test`

Run unit tests to ensure that the backend communication (to the tree health detection server) works as expected before connecting it to the UI.

Tooling: `flutter_test`, `mockito`, `http`

Integration Testing

These tests ensure smooth interaction between key modules.

Test Scenarios:

Simulated complete data flow from:

- Sensor input → Firestore → Real-time UI update (charts, tables, cards)

Test scripts have been implemented to write the sensor inputs to the Firestore raw_readings collection and verified through the UI components.

- Image upload → ML API → Firestore → UI display (analysis result card).

Images have been uploaded from the UI and called ML API (via fetch in React). API responses saved to Firestore (Includes tree count, health summary, and processed image URL).

Manual Testing

These tests have been done to validate the end-user experience across platforms.

Both **Flutter mobile app** and **React web app** tested on multiple devices and browsers.

Mainly focused on:

- Navigation flow
- Data consistency across views
- UI responsiveness
- Admin vs. user role behavior

Load Testing

To test how the system handles high-frequency data updates and concurrent users.

Tools Used:

- **Artillery** - Simulated bursts of real-time sensor data writes to Firestore.
- Results verified that real-time UI update speed under load.

Security Testing

To ensure data privacy and correct access control.

- Tested Firebase Firestore security rules using test accounts with different roles.
- Custom claims (admin, user) validated via token inspection in frontend.