ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

Department of Information and Electronic Engineering

Diploma Thesis

"Study, Design and Development of a Hydroponic Cultivation System"

Student
Komninos Lazaros Lekkas
Registration Number 185213

Supervisor
Maria S. Papadopoulou
Assistant Professor

Student
Antonios Mavrakis
Registration Number: 185222

Date 22/01/2026

Τίτλος Δ.Ε.
«Μελέτη, σχεδίαση και ανάπτυξη συστήματος υδροπονικής καλλιέργειας»
Κωδικός Δ.Ε.
24226
Ονοματεπώνυμο φοιτητών
Κομνηνός Λάζαρος Λέκκας / Αντώνιος Μαυράκης
Ονοματεπώνυμο εισηγητή
Μαρία Παπαδοπούλου
Ημερομηνία ανάληψης Δ.Ε.
23-09-2024
Ημερομηνία περάτωσης Δ.Ε.
22-01-2026

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Acknowledgements

We would like to express our sincere gratitude to our supervising professor, Mrs. Maria Papadopoulou, for her valuable guidance, support, and advice throughout the duration of our thesis project.

We would also like to extend our special thanks to our fellow students, Georgios Poutachidis, for his assistance in the construction process and the practical help he provided, his contribution was essential in enabling us to successfully implement the project on a practical level.

And also to Vasilios Konstantinidis who lent us the Arduino MEGA and some starting tools, without him we would have experimented with the arduino and start the process of coming up with this project.

Prologue

The selection of this thesis topic stemmed from our shared interest in modern technology and its practical applications in agriculture. Our goal was to develop a smart system capable of monitoring and managing hydroponic crops through a mobile application built with React Native (Expo) [1], in combination with multiple sensors connected to an Arduino MEGA. This work aims to provide an innovative and efficient solution in the field of agriculture.

Our objective was to create a complete application that enables real-time sensor data monitoring, device management over Wi-Fi, and data analysis through interactive charts.
To achieve this, we utilized technologies such as Redux, WebSockets and Expo. Through this project, we gained valuable experience in the design and development of IoT-based mobile applications, as well as in the integration of backend and frontend systems into a unified and interactive environment.

Working on this project not only allowed us to deepen our understanding of modern application development technologies, but also made us appreciate the role of technology in improving agricultural practices and promoting sustainable development.

Περίληψη

Η υδροπονική καλλιέργεια έχει αναδειχθεί ως μια αποδοτική και βιώσιμη γεωργική μέθοδος, προφέροντας ταχύτερη ανάπτυξη φυτών και εξοικονόμηση νερού. Ωστόσο, η διατήρηση των ιδανικών υδροπονικών συνθηκών απαιτεί συνεχή παρακολούθηση βασικών περιβαλλοντικών παραμέτρων, όπως το pH, η θερμοκρασία, η υγρασία και η συγκέντρωση των θρεπτικών συστατικών. Οι παραδοσιακές μέθοδοι παρακολούθησης είναι χειροκίνητες, απαιτούν πολύ χρόνο και κόπο και είναι επιρρεπείς σε ανθρώπινα λάθη, γεγονός που μπορεί να οδηγήσει σε μειωμένη παραγωγή και σπατάλη πόρων.

Η παρούσα πτυχιακή εργασία παρουσιάζει το **Auto Harvest**, ένα σύστημα υδροπονικής παρακολούθησης βασισμένο στο Διαδίκτυο των Πραγμάτων (IoT), το οποίο επιτρέπει τη συλλογή και απεικόνιση δεδομένων σε πραγματικό χρόνο μέσω μιας κινητής εφαρμογής κατασκευασμένης με React Native (Expo) [1]. Το σύστημα έχει σχεδιαστεί με σκοπό την βελτίωση της αποδοτικότητας μιας υδροπονικής καλλιέργειας, παρέχοντας στους καλλιεργητές δεδομένα αισθητήρων σε πραγματικό χρόνο και άμεσες ειδοποιήσεις, όταν κρίσιμες παράμετροι αποκλίνουν από τα ιδανικά όρια. Επιπλέον, οι χρήστες έχουν τη δυνατότητα να ελέγχουν χειροκίνητα βασικά εξαρτήματα, όπως την αντλία νερού και την αντλία αέρα, εξασφαλίζοντας έγκαιρη παρέμβαση, όταν αυτό είναι κρίνεται απαραίτητο.

Με την ενσωμάτωση αισθητήρων IoT σε ένα σύστημα απεικόνισης και ελέγχου δεδομένων μέσω κινητής συσκευής, το έργο αυτό στοχεύει στη μείωση της ανθρώπινης προσπάθειας, ενώ ταυτόχρονα βελτιώνει τη λήψη αποφάσεων και την ακρίβεια στη διαχείριση της υδροπονικής καλλιέργειας. Τα αποτελέσματα της μελέτης δείχνουν ότι η παρακολούθηση σε πραγματικό χρόνο ενισχύει σημαντικά την αξιοπιστία του συστήματος και τη διαχείριση της παραγωγής, ανοίγοντας τον δρόμο για μελλοντικές επεκτάσεις που θα περιλαμβάνουν ανάλυση μέσω cloud, βελτιστοποιήσεις με τεχνητή νοημοσύνη και έξυπνο αυτοματισμό.

Abstract

Hydroponic farming has emerged as an efficient and sustainable agricultural method, offering superior water conservation, faster plant growth, and reduced dependency on soil conditions. However, maintaining optimal hydroponic conditions requires continuous monitoring of key environmental factors such as pH levels, temperature, humidity, and nutrient concentration. Traditional monitoring methods are manual, labor-intensive, and prone to human error, which can lead to poor crop yields and increased resource wastage.

This thesis presents **Auto Harvest**, an IoT-driven hydroponic monitoring system that enables real-time data collection and visualization through a mobile application built with React Native (Expo) [1]. The system is designed to improve hydroponic farming efficiency by providing farmers with accurate, real-time sensor data and instant alerts when critical parameters deviate from optimal ranges. Additionally, users have the ability to manually control essential components such as the water pump and air pump, ensuring timely intervention when necessary.

By integrating IoT sensors with a mobile-based data visualization and control system, this project aims to reduce human effort while improving decision-making and precision in hydroponic farming. The findings of this study demonstrate that real-time monitoring significantly enhances system reliability and crop management, paving the way for future cloud-based analytics, AI-driven optimizations, and smart automation enhancements.

Content

Section 1: Introduction

1.1 Problem Definition

Traditional agricultural methods are increasingly challenging due to the ongoing impacts of climate change and natural disasters happening all around. Farming practices typically consume huge amounts of water and rely on weather conditions for their growth, demanding extensive care from humans. Soil agriculture also has many problems such as soil degradation, pest infestations, disease outbreaks and unpredictable cycles which reduce efficiency and diminish crop yields [2]. As a solution to the problem, hydroponic systems have emerged all around us, using nutrient solutions. Even though these systems contribute to the solution, the lack of advanced technological integration to the systems present shortcomings. Manual monitoring of the systems is very time-consuming, and it requires a human to be present at all times to take a reading and analyze it, causing in some cases an inconvenience if something did not go as planned. It was not detected on time.

1.2 Goal

The goal of this project is to create a reliable, scalable, and user-friendly hydroponics monitoring system that allows users to control and observe multiple installations through a single mobile application. By combining real-time sensor feedback with remote management capabilities, **Auto Harvest** aims to simplify the daily operation of hydroponic systems, reduce the risk of unnoticed issues and support more efficient and automated growth (even for users without technical expertise).

1.3 Proposed System

To solve the challenges of manual hydroponic monitoring, we developed **Auto Harvest**, an automated system that uses IoT sensors and an application to monitor and manage hydroponic setups remotely. At the hardware level, an Arduino Mega collects real-time data from sensors measuring pH, EC, temperature, and light. This data is sent via Wi-Fi to a backend service, which processes and stores it. Users can view this information through a mobile app built with React Native and Expo, where they can register controllers, track sensor readings, and receive alerts when something goes wrong. The app supports multiple controllers and allows real-time control of devices like pumps and relays, making it scalable and flexible. The system is designed to be easy to use, even for non-experts, while also being modular and future-proof for further expansion.

This project covers the complete implementation of an automated hydroponic monitoring system, from hardware assembly to application development. It includes the design and programming of the Arduino firmware, the creation of a backend service for data handling and the development of a mobile application for the interaction of the user. The system supports multiple hydroponic controllers, real time data visualization, remote device control, and push notifications.

We recommend the adoption of our project for individuals or small-scale growers seeking a reliable, easy to use, and scalable solution for hydroponic monitoring. Its modular design, mobile application and real time alerting system make it suitable for users without technical expertise. By centralizing control and automating data collection, **Auto Harvest** reduces the risk of human error, improves crop consistency, and simplifies day to day maintenance. For anyone looking to enhance the precision and efficiency of their hydroponic system, this solution offers a practical and cost-effective approach.

Section 2: Software

Auto Harvest provides software that makes managing the system easy and logging data safely. Software is split into two repositories, the front-end and the back-end. Our front-end provides an asynchronous way to monitor and handle systems using React Native and Expo for using the app both as a web app and a mobile application. To manage the application's data and to ensure data persistence, we used React Redux and Redux Persist. The firmware acts as the intelligence layer of Auto Harvest. It is written for the Arduino ATMega 2560 and collects real-time data from sensors (temperature, pH, humidity, TDS and flow), controls pumps and actuators, and communicates securely with the backend over MQTT. It is designed in a modular and portable way, supporting persistent storage of Wi-Fi and calibration data, automatic pairing through a lightweight local webserver and safety mechanisms like pump shutdown on dry-run detection. This ensures the system can operate reliably, extend easily to future hardware like the ESP32 and continue functioning autonomously even when cloud connectivity is unstable.

The backend is built with Node.js and Express, designed to act as the central coordination point of the system. It provides REST APIs, WebSocket streaming and MQTT handlers, bridging communication between the hardware devices, the mobile/web clients and the cloud infrastructure. MongoDB is used for persistent storage of both user entities and high-volume telemetry logs, with support for time-series queries and historic record retrieval. The backend enforces strict authentication and authorization using JSON Web Token (JWT), ensuring secure access to devices and user data. It also implements notification delivery, command routing and validation logic. This allows the system to operate reliably and safely in real-time.

There are also 3rd party services in use such as self hosted containers for MongoDB, a NoSQL database optimized for performance compared to SQL databases, and ActiveMQ, a lightweight, fast and reliable MQTT message broker used by the backend and the firmware.

2.1 Backend Specifications

2.1.1 Specifications

The Auto Harvest backend acts as a central coordination point for the entire IoT platform. It bridges communication between the hardware devices, the mobile/web client and the cloud services, enabling real-time control, data logging, user management and feedback mechanisms. In this section we will present its core responsibilities.

Among the most important responsibilities of the Backend is the User Authentication and Authorization. JSON Web Tokens (JWT) are utilized for stateless and client agnostic session Management,Tokens are issued after a successful user authentication. Each client is responsible for presenting them to the backend when requesting secure endpoints. All endpoints require authorization via JWTs, except the publicly accessible Signup and Login routes. The Authentication process is handled via custom middleware triggered after a successful route match, before the controller's code is executed.

One of the main functions of the Backend is the device and metadata management of the system. Users register and manage devices called "controllers" which represent the physical hydroponic units. Each

controller maintains metadata such as owner, location, operational state. A user can have multiple controllers registered, edited, listed and deleted as permitted by his authorization role since he is authorized to use all of the relevant CRUD (Create, Read, Update, Delete) endpoints of the Backend.

Another critical function of the backend is the "Sensor Data Ingestion" and "Command Routing". The physical IoT devices periodically publish environmental readings utilizing the MQTT protocol, the reported values are: air temperature, water temperature, humidity, pH, TDS, water volume flow rate. These values are sanitized, validated, stored in MongoDB through the sensor log service and also emitted to WebSocket clients for real time data reporting. These are the inbound MQTT messages, but there are also outbound MQTT messages classified as "Commands". Commands are issued via the mobile app (for example, start-pump, set-polling-rate, stop-pump, start-air-pump, stop-air-pump) and are received over HTTP by the Backend and routed to devices via MQTT. This ensures the stateless nature of the clients and reduces the resource requirements of monitoring devices, improving overall security, since no client device is required to open and maintain an MQTT connection with the IoT device except the backend itself.

Furthermore, another important function of the Backend is the notification management. Alert conditions such as, "low/high water level", "high/low salinity", "water pump is blocked", are detected by the backend during the sensor data ingestion step, and delivered as push notifications using the expo-server-sdk library. So, provide end-user real time critical alerts to prevent crop failure, hardware deterioration, and nutrient burns and deficiencies.

Finally, the backend is also capable of real-time two-way communication via WebSocket streaming. The backend exposes a secure socket connection authorized with JWT and utilizing the Socket.io library, allowing clients to subscribe to real-time sensor logs and other in-app updates, excluding the notifications.

2.1.2 Express Server

The backend is structured as a modular Express.js application and utilizing a custom Route-Middleware-Controller-Service-Model approach, served under the /api namespace. Its core responsibilities are divided across HTTP routes, WebSocket connections, and MQTT handlers that communicate with firmware and frontend clients. Its key components include the setup of a basic express application with CORS and JSON parsing, REST API routes for HTTP communication, WebSocket server utilizing Socket.io, an MQTT connection managed in mqtt.ts and a MongoDB connection using the mongoose library. The REST API HTTP routes are presented at the next table.

Table 2.1: REST API routes

Route	Method	Description
/api/signup	POST	Create a new user
/api/login	POST	Authenticate user, produce JWT
/api/controller	POST	Register new IoT controller for a user
/api/controller	PATCH	Edit existing IoT controller

		the user owns
/api/controller	DELETE	Delete a selected controller the user owns
/api/controller	GET	List all the controller the user owns
/api/controller/{id}	GET	List a selected controller the user owns
/api/logs	GET	Retrieve logs by a controller, value type, range
/*	ANY	The last route in order, redirecting to the 404 page

Each route (excluding signup and login) utilizes the Json WebToken and the Authorization middleware. The JsonWebToken Middleware is responsible for the validation and decoding of the tokens payload, while the Authorization Middleware attaches the user session object to the request in order to be handled and parsed by controllers downstream on the Middleware chain.

2.1.3 Code Structure and Patterns

The code is structured around a service-driven layered architecture, utilizing NX generators. Each domain (users,controllers,logs) has a clear path from request to response.

Route → Middleware(s) → Controller → Service(s) → Model(s)

This approach enables a flexible and modular design prioritizing code reusability and loose coupling of the services. These are the most important constraints that are taken into account during development of the Backend:

- A request must match 1 route, including the fallback 404 route
- A route must trigger 0 or many middleware and must have 1 controller at the end of the middleware chain
- A middleware must either call the next() function to execute the next middleware in order or throw an an unhandled error in order to be handled by the error middleware (bypassing the controller entirely)
- A controller is technically a middleware but must not use the next() function. It either responds successfully or throws an unhandled exception in order to be handled uniformly at the error handling middleware, a controller can utilize 0 or many services in order to fulfil the request. The controller must not implement domain/business logic, and is only responsible for parsing the request object, calling relevant service functions with parameters derived from the request object and transforming the return value of the service functions in order to be compliant with the REST API documentation.
- A service is responsible for the implementation of the domain/business logic, it can utilize 0 or many models in order to access and 0 or many services in order to create a simplified facade for the controller to utilize. A service function could be invoked at the controller level, but also outside of it, including cronjobs, or the initialization phase of the application

- A model is responsible for mutating and querying the database as well as calling more complex aggregate methods (group by, sum, average).

The collector service handles log validation, type-based parsing, and invocation of the Log model functions in order for the logs to be inserted to MongoDB, it is also responsible for performing critical safety checks such as: Detecting no water flow while the pump is on, emitting real-time shutdown commands to prevent hardware damage, logging the sensor value anomaly and notifying the user.

The io service sets up and manages websocket rooms per controller ID, allowing subscribed clients to receive new-log events for live telemetry, push-command for backend-to-device actions (like manual pump shutdown for maintenance), info messages for diagnostics (pump shut down successfully, water is now at adequate levels, salinity is restored to expected range).

The expoPushNotification service is responsible for sending critical and non-critical push alerts to mobile users using Expo's push API, triggered during fault states (e.g pump failure), it is responsible for token validation, error handling on push failure, asynchronous delivery and notification logging (seen status of notifications, delivered, received).

Each service is stateless, reusable and separated from route/controller logic (it doesn't handle request and responses), this approach simplifies unit testing and the overall scalability of the system.

2.1.4 Utilized Libraries

The backend uses a stable and modern mix of libraries to support functionality and performance, a short description and reference of them will be provided below.

express - Web routing and server logic.
mongoose - MongoDB schema and model definitions.
cors - Setting options for Cross-origin resource sharing.
morgan - HTTP request logger
mqtt - Lightweight Publish/Subscribe protocol for IoT device messaging
socket.io - Enables full duplex real-time communication between backend and frontend
bcrypt - Modern and secure password hashing
jsonwebtoken - JWT sign/verify utilities
expo-server-sdk - Sends push notifications to registered devices
@faker-js/faker - Simulates fake input generation for testing (sensor log generation testing)
node-cron - Scheduled background jobs (future alert sweeps, cleanup)
nx, @nx/node - Monorepo tooling and generators

2.1.5 Debugging and Profiling

The Auto Harvest project is set up in a way that supports debugging and profiling during development as well as production-like environments. During development the most common way of troubleshooting bugs is performed by utilizing console.log and console.trace by printing relevant objects and looking for unexpected values. When console logging is insufficient to identify the bug, the app is launched in debug mode, with a configuration specified in the "launch.json" file that launches the app using ts-node, mapped source paths, and custom ports so the google Chrome DevTools can be attached. This approach is also utilized when the code is inspected for memory leaks, by attaching it to Chrome DevTools and using the built in memory profiler.

In order to test the HTTP endpoints behaviour, Postman is utilized in both development and production, this is especially useful in the production environment, where the server itself may be able to produce the correct response, but either the request never reaches it, or the response fails to be propagated by the ingress services (cloudflare → VPC IP routing → nginx load balancer → backend).

The MQTT behaviour is validated either by logging each received message at the IoT device level and the backend level, and messages can be replicated on demand by the MQTT's web interface exposed on a separate port. At production it is publicly inaccessible and to be accessed it must be tunneled to a localhost port using Aptakube or similar DevOps tools.

In order to rapidly develop the Backend and Frontend client, simulated sensor streams were implemented using @faker-js/faker in order to generate schema-compliant fake sensor logs triggered by a custom interval function. These mocks were proven to be very helpful in allowing our team to develop and test the Backend and Frontend independently of real hardware.

In the production environment, logs are streamed to standard output and parsed via kubectl logs in the GKE environment, tools like Aptakube provide a GUI in order to filter and query the output logs. In the future we plan to centralize the logging process via Prometheus or Grafana, GCP is ready out-of-the-box to be integrated with such tools, which are critical for production-grade observability.

2.2 Firmware

2.2.1 Firmware Specifications

The Auto harvest firmware is designed to operate as a self-contained intelligent microcontroller system, responsible for real-time sensing, actuator control, connectivity management, and communication with the cloud backend via MQTT. It is deployed on an Arduino ATMega 2560, but it was written in a fully modular and portable manner to support future migration to more powerful chips like the ESP32, since the ATMega 2560 is not viable for production-ready systems.

The firmware supports periodic polling of six key sensors, including temperature, ph, humidity, TDS and water flow, using timer-based logic loops to avoid blocking behavior and maintain performance and reliability. It provides actuator control via relays for water and air pumps. It is capable of live updates to a 16x02 LCD display, including real-time status messages and telemetry for infrastructure-independent monitoring. It supports bi-directional MQTT communication with the backend in order to handle commands and to publish telemetry data. Furthermore, a key-value API for the EEPROM module is used to persistently store Wi-Fi credentials, MQTT settings, and calibration data. It also contains a minimal implementation of a custom webserver for the initial communication of the Frontend client and the IoT device for pairing and Wi-Fi credentials provision.

Once powered on, the firmware loads the stored configuration of SSID, Wi-Fi password and calibration data from the EEPROM, it attempts to connect to Wi-Fi, and if it fails to connect or the EEPROM is empty, it falls back to AP mode enabling the pairing process to begin. Upon Wi-Fi connection, it establishes a secure MQTT session and begins normal operation. Periodically, it reads all sensor values and sends telemetry info to the backend as well as listening for incoming MQTT commands and executes them. As a safety measure, it detects sensor anomalies and auto-shuts down in order to protect itself from dry running.

The firmware is structured around a global AppContext, which initializes and holds references to the Wi-Fi Manager for AP and Station mode switch, Web Server for the pairing UI, Disk Manager for EEPROM access, Module Manager for actuator initialization, Data Collector for structured telemetry readings and the MQTT Client for backend communication. This context-driven approach avoids global variables, improves encapsulation and makes the firmware easier to extend, test and in the future migrate to a more production ready platform.

2.2.2 PlatformIO for Arduino

The firmware for the Auto Harvest system is developed using PlatformIO, a modern ecosystem and build system designed for embedded software. PlatformIO replaces the traditional Arduino IDE with a more robust, version-controlled, and scalable workflow that integrates seamlessly with Visual Studio Code, unit testing and multi-board support.

PlatformIO offers several critical advantages over the traditional Arduino workflow.

- Per-project dependency management using platformio.ini
- Intelligent IntelliSense and LSP integration for C++ code
- Faster builds and incremental compilation
- Source folder structure support, enabling a clean modular architecture (src/, lib/, include/, test/)
- built-in support for multiple environments (for example, different boards, test configs)
- Easy integration with serial monitors, OTA uploaders and debugging tools (which sadly are not supported for the Arduino ATMega 2560)

The platformio.ini file defines the board used (in our case the megaatmega2560) the framework used (arduino core), custom build flags for the C++ compiler, the library dependencies and the location of the core folders (libraries, source code, third-party file inclusions and the testing directories)

```
[platformio]
lib_dir = apps/iot/io-manager/lib
src_dir = apps/iot/io-manager/src
include_dir = apps/iot/io-manager/include
test_dir = apps/iot/io-manager/test

[env:megaatmega2560]
platform = atmelavr
board = megaatmega2560
framework = arduino
build_flags = -DSOC_SDMMC_HOST_SUPPORTED -std=c++11
lib_ldf_mode = deep
```

Image 2.1:Platform.IO ini file

2.2.3 Internet Connectivity

A reliable internet connection is needed for the system to provide remote control, telemetry and integration with cloud services. The current prototype connects to Wi-Fi using a lightweight pairing process and securely publishes data to the cloud using the MQTT protocol over TLS.

On first boot or after an EEPROM reset, the device checks for stored SSID and password entries via a custom EEPROM parser. If no credentials are found the device enters pairing mode. The pairing process follows these steps:

- User installs the Android app and taps “Pair Device”
- The app prompts the user to connect to a Wi-Fi AP called ESP8266, hosted by the ESP8266 in AP mode
- Once connected, the app displays all visible Wi-Fi networks scanned by the Arduino
- The user selects one and submits their password via a POST/pair request, sent as a FormData body

- If the credentials are valid, the ESP responds with HTTP 200, shuts down the access point and stores the network configuration in EEPROM
- The app then prompts the user to name and register the device and to add metadata like location and plant count

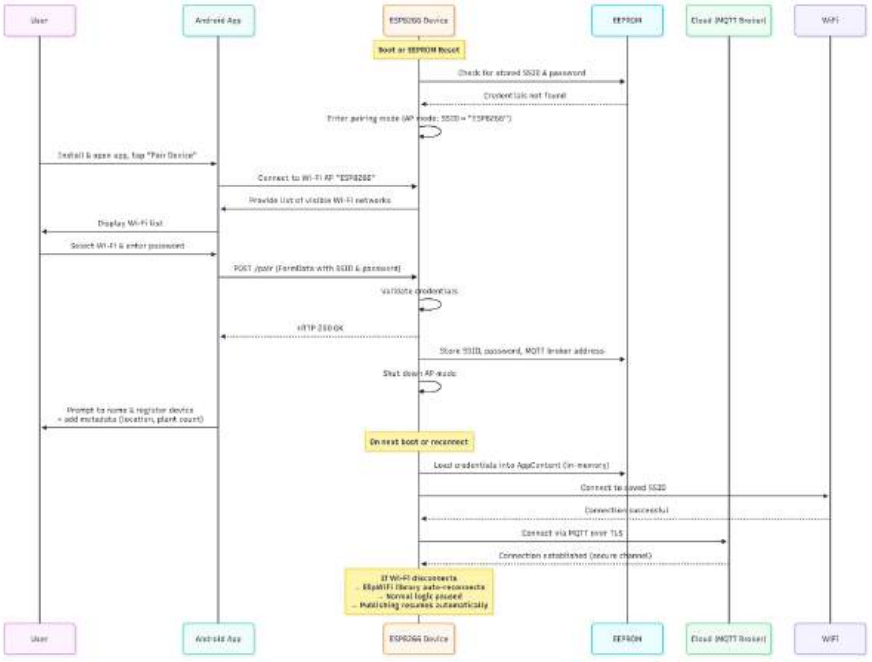

Image 2.2: Sequence Diagram of the System's initialization logic (Out of the box experience)

Once paired, the ESP-01 automatically reconnects to the saved Wi-Fi on startup. If the connection drops, the ESpWiFi library attempts to reconnect in the background. During disconnection all normal logic is paused to avoid data loss or errors, once reconnected, publishing resumes without user intervention. All credentials like SSID, password and MQTT broker address are stored in the EEPROM. On boot, they're loaded into the dedicated AppContent object to reduce EEPROM reads and enable fast in-memory access across the application

Once online, the device connects to the backend using MQTT over TLS, ensuring both performance and security.

2.2.4 Code Structure and Patterns

The firmware for Auto Harvest is built around a strictly layered and modular architecture, enforcing separation of concerns between data acquisition, connectivity, control login and system configuration. Inspired by modern embedded systems and software architecture principles, the codebase emphasizes clarity, flexibility and reusability, sacrificing performance in order to utilize Object-Oriented design patterns, such as Strategy, Context and Singleton as the resource-constrained ATMega 2560 proved it could handle it without any problem and that the overall benefits outweighed the potential drawbacks of such approach. At its heart the firmware is organized into the following component groups:

Table 2.2: Firmware component groups

Layer	Description
Abstract	Shared interfaces for all modules and sensors
Sensors	Individual drivers for pH, TDS, flow, temperature and humidity
Modules	Output devices like pumps, LCD, and Wi-Fi controller abstraction
Services	Infrastructure services like MQTT, EEPROM, Wi-Fi manager, LCD
Utility	Low-level tools like timers, string parsers, map formatters
App Context	Central orchestrator that initializes, connects and updates layers

The directory structure is organized as follows

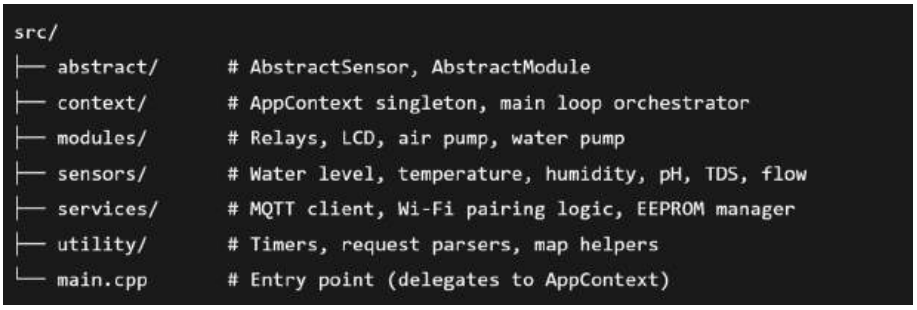

Image 2.3: Firmware directory structure

This approach ensures that each hardware feature is implemented in isolation, dependencies flow downward not across modules and minimal changes when a component swap is needed (like replacing the LCD16x02 with an OLED).

In order to focus on code reusability and clarity, the AbstractSensor abstract class and AbstractModule abstract class were created. By sharing and consolidating the functionalities of sensor initialization, polling, and status reporting, module initialization, switching, forcing the instance classes to implement methods like getName(), getType(), getStatus(), developing in isolation becomes streamlined and easier, with IntelliSense catching implementation errors before having the code built. This interface-driven approach allows for module hot-swapping (replacing one pump with another, or a ph sensor with another), parallel iteration through sensors/modules for polling or control, and shared data containers, so all sensors can report data in a uniform manner. This design effectively applies the Strategy Pattern, as each sensor implements the same abstract interface and can be iterated uniformly. The firmware treats all sensors as interchangeable strategies, invoking readData() without being coupled to the specific sensor type. It should be noted that ATMega2560 doesn't support breakpoints nor error reporting, so when an exception occurs, the microcontroller resets code execution. This means that without preemptively coding guardian assertions (like checking manually if a value is 0

before having it as a denominator in a division, instead of handling it with try-catch-finally blocks) development becomes exponentially harder as more complexity is introduced to the system. Another problem that needed to be addressed as complexity increased, where unintended side effects, as globally scoped variables were mismanaged by dependencies down the line, creating problems in completely different places. This problem was addressed by creating the AppContext, a globally available reference where all shared object instances exist in, with safeguards in place to prevent accidental re-initialization of shared variables (Singleton Pattern). Effectively, this means that the ModuleManager and the SensorManager can be accessed from other classes only by referencing the AppContext global instance, which can strictly exist only once in the scope of the firmware.

Regarding the Arduino's main loop mechanism, its functionality was offloaded and delegated to AppContext::loop() for better manageability and code clarity. Inside the AppContext::loop() function, a timer mechanism was implemented, where the Timer class instance remembers the last time it was fired, and using conditional assertions, it prevents data reads before the internal timer resets after a pre-set amount (5 seconds by default).

```
if (sensorPollTimer.expired()) {
    collector->readAllSensors();
}

if (dataSendTimer.expired()) {
    mqttClient->publishSensorData(currentData);
}

if (lcdUpdateTimer.expired()) {
    lcd->updateScreen();
}
```

Image 2.4: AppContext logic

The approach mentioned above provided several benefits. It made the codebase modular, and loosely coupled since additional sensors could be added with minimal effort, and module replacement became an isolated process without unintended side effects. Future unit testing can be implemented minimally, since each service and hardware driver can be simulated or mocked. The overall scalability of the system increased, more classes that implement the same abstract classes can be written and added with little to no effort, since code duplication is strictly avoided whenever it is possible. The AppContext acting as a single, universal source of truth simplifies state management and keeps track of all system-wide state and resources, preventing the developer from re-initializing existing modules by mistake. The performance loss and memory bloating associated with employing classes and object-oriented patterns were outweighed by all of the previously listed advantages.

2.2.5 Control Logic and Custom Behaviors

While the last section focused on the structural organization of the firmware, this section highlights its runtime behaviour and the way components are utilized in practice. Beyond basic sensor polling and device connectivity, the firmware is designed to exhibit intelligent and proactive behaviour, automatic safety decisions, responding to commands and controlling modules in a state-aware manner. These features add real value in unattended environments like hydroponic farms. Previously we explained why Timers were used, now we are going to explain how they are used. There are three primary timers that coordinate the firmware's runtime operation.

Table 2.3: Primal timers for data handling

Timer	Upon expiry	Interval
SensorPollTimer	Reads data from all sensors	2 seconds
DataSendTimer	Publishes collected data to MQTT	5 seconds
LcdUpdateTimer	Rotates and updates LCD screen contents	1 second

Each timer checks .expired() inside the AppContext::loop() and triggers its associated behavior without blocking the main thread. This ensures effective and efficient non-blocking multitasking, a necessity for sensor-heavy embedded systems.

Another aspect of intelligent behavior is IoT-side fault detection. One of the most critical custom behaviours is safety logic built into the firmware itself. A key implementation is in the DataCollector service where it performs the following checks.

If "the water pump is active" and "the flow sensor detects no pulses"
then deactivate the pump, push a warning log to MQTT and push a message to the LCD queue ("No water flow detected!")

This local safeguard prevents the pump from dry-running and burning out, and allows the system to act independently of cloud latency or outages, or wait until it gets a pump-off message from the MQTT queue, preventing hardware damage proactively. The firmware also listens to a configurable MQTT topic for incoming device commands. When received, commands are enqueued and processed in the loop according to the "First-Come, First-Served" principle. Supported commands include:

Table 2.4: Device command's behavior

Command	Behavior
pump-on/pump-off	Turns the water pump on or off
air-pump-on/air-pump-off	Deactivates the air pump
set-sensor-poll-interval	Dynamically changes the poll rate

turn-to-ap	Enables access point mode for local Wi-Fi pairing
lcd-on/lcd-off	Turns the LCD screen on or off
liveness-check	Lets the server know if the device is responsive

The LCDModule is used as a local feedback interface, displaying uptime, critical sensor readings, Wi-Fi connection status. The design also allows future extensions, such as additional system states (e.g. “Waiting for pairing”, “Wrong credentials”, “Further setup needed”). It also features a message queue, to temporarily override the screen displaying critical sensor reads (e.g. “Water Low!”), giving real-time offline feedback to users or technicians on-site as well as firmware developers for isolated debugging. As it was discussed above, all major interactions are delegated to AppContext, which has the following responsibilities:

- Tracks and updates timers
- Executes fault logic
- Reads sensor values
- Dispatches MQTT messages
- Injects debug messages to LCD
- Enforces command routing and permission logic

This allows all firmware control logic to remain centralized, ensuring minimal bugs and race conditions between modules.

2.2.6 Utilized Libraries and Drivers

The firmware relies on a set of open-source libraries that provide low-level sensor support, communication protocols, hardware interfacing and memory access. These libraries are managed via PlafromIO’s dependency manager. Each library was chosen for reliability, active maintenance and compatibility with the ATMega2560 and ESP01 boards. In the table below, a complete list by type and a short description is provided for each utilized 3rd party library.

Table 2.5: 3rd party library details

Library	Type	Purpose
PubSubClient	Communication	MQTT client for communicating with ActiveMQ.
ESP8266WiFi	Communication	Manages Wi-Fi client and AP modes.
EEPROM	Storage	Reads and writes bytes positionally to EEPROM.
OneWire	Sensor	Required for DS18B20 (water temperature sensor).

DallasTemperature	Sensor	High-level driver for DS18B20 sensor
DHT sensor library	Sensor	Reads DHT11 sensor for humidity and air temp
ArduinoJson	Utility	Serializes/Deserializes JSON payloads
LiquidCrystal_I2C	Display	Controls the 1602LCD via the I2C protocol

Custom drivers were also developed in addition to the utilized 3rd party libraries, this need came after we discovered that the provided library for the pH sensor didn't respond as expected. The vendor-provided library consistently displayed inverted readings, such as high pH for acidic solutions and low pH for basic solutions even after a correct 2-point calibration. To resolve this, a custom driver was developed based on the Nernst Equation, which models the proportional relationship between electrode voltage and hydrogen ion concentration. The driver averages multiple ADC samples, converts the result to millivolts, and applies real-time temperature compensation by integrating with the DS18B20 sensor. This ensures stable, accurate readings across environmental conditions.

```
// Take multiple samples and average them
const int numSamples = 50;
float totalVoltage = 0;

for (int i = 0; i < numSamples; i++)
{
    totalVoltage += analogRead(pin);
    delay(10); // Small delay to allow for stable readings
}

// Convert averaged ADC value to millivolts
voltage = (totalVoltage / numSamples) / 1023.0 * 5000.0;
```

Image 2.5: ADC averaging and conversion to mV

The sensor readings take approximately 500ms, the sensor is polled 50 times at a 10ms interval. Once polling is completed an average is calculated and the value is normalized and scaled to output mVs.

```
float k = 0.33; // sensitivity factor
float tempSlope = (59.16 + 0.1984 * (temperature - 25.0)) * k; // mV adjustment for given temperature
```

Image 2.6: Temperature compensation

The theoretical Nernst slope at 25C is 59.16mV/pH. The term 0.1984(temperature - 25.0) approximates the slope change per C, and the k parameter is a sensitivity correction factor added empirically.

```
float slope = (7.0 - 4.0) / ((neutralVoltage - 1500.0) / tempSlope - (acidVoltage - 1500.0) / tempSlope);
float intercept = 7.0 - slope * (neutralVoltage - 1500.0) / (tempSlope);
```

Image 2.7: Calculation of slope and intercept for the standard line equation

This is the relevant code for the 2-point calibration, this will allow us to convert our expected output at a correct interception point, and with a matching slope. The neutralVoltage is the reading of the sensor while being submerged in distilled water, and acidVoltage is the reading of the sensor while being

submerged in an acidic calibration buffer solution. Both of those values are calculated and saved while the sensor is being calibrated. The value “1500” refers to the neutral solution expected output in mV provided by the sensor manufacturer. These parameters are then combined in a standard line equation form f(x) = mx + b in order to calculate the pH from the mV [6].

```
// Calculate pH
float rawPh = slope * ((voltage - 1500.0) / tempSlope) + intercept; // y = mx + b
```

Image 2.8: Application of standard line equation with precalculated parameters

The new driver implements the same abstract interface as the other sensors, meaning it can be polled, calibrated and reported uniformly by the system without special handling. This allows it to integrate with the DataCollector and MQTT telemetry pipeline, while still giving it full control over calibration and signal interpretation.

Another sensor that didn’t have vendor-provided drivers was the Gravity TDS Meter [7] which measures the total dissolved solids (TDS) in water by converting an analog voltage into parts per million (ppm). The firmware treats it like other sensors, implementing the shared AbstractSensor interface. The sensor reading is adjusted for water temperature using a linear compensation factor, ensuring consistent results under varying environmental conditions. The compensated voltage is then converted to TDS using a polynomial formula provided by the manufacturer. While the current implementation uses a fixed conversion formula, the architecture allows for future two-point calibration to correct sensor offset and scaling, similar to the pH electrode calibration process. Data from the TDS sensor is incorporated into the sensor polling timer, published to MQTT, and displayed on the LCD, providing real-time feedback for both monitoring and automated control decisions.

```
double GravityTDSMeter::readTDS(double temperature)
{
    // Read the analog value from the TDS sensor
    int analogValue = analogRead(pin);

    // Convert the analog value to voltage
    double voltage = analogValue * (5.0 / 1024.0);

    // Calculate the TDS value using the voltage and temperature
    double compensationCoefficient = 1.0 + 0.02 * (temperature - 25.0);
    double compensationVoltage = voltage / compensationCoefficient;
    double tdsValue = (133.42 * compensationVoltage * compensationVoltage * compensationVoltage
        - 255.86 * compensationVoltage * compensationVoltage
        + 857.39 * compensationVoltage) * 0.5; // TDS conversion factor

    return tdsValue;
}
```

Image 2.9: Firmware code suggested by DFRobots for sensor handling

First of all, an analog-to-voltage conversion is performed to map the 10-bit ADC reading to 0-5V. Then the temperature compensation coefficient is calculated utilizing the water temperature sensor value. Then a polynomial conversion is applied, which converts the compensated voltage into TDS, based on the coefficients derived by the manufacturer. Finally, a conversion from ppm to EC is performed by multiplying TDS with a k conversion factor[8] that usually ranges between 0.5 and 0.8. We measured the output of the TDS sensor and the output of a store-bought EC meter and found the value to be 0.5.

The flow meter sensor also required custom code in order to function. This sensor needed a module supporting hardware level interrupt supporting pins, and luckily our ATMega2560 comes with a couple of them. Initially during the sensor initialization phase an interrupt is attached to the sensor pin, triggering the flowSensorISR function, while the input is rising.

```
void WaterFlowSensor::initialize()
{
    pinMode(pin, INPUT);
    attachInterrupt(digitalPinToInterrupt(pin), flowSensorISR, RISING);
    Serial.println("Flow sensor initialized");
}
```

Image 2.10: Attaching interrupt to input pin

When the flowSensorISR is triggered this is the executed code

```
void WaterFlowSensor::flowSensorISR()
{
    unsigned long currentTime = millis();

    // Calculate time between pulses
    unsigned long elapsedTime = currentTime - instance->lastPulseTime;

    if (elapsedTime > 0)
    {
        instance->flowRate = 1000.0 / elapsedTime; // Hz (pulses per second)
    }

    instance->pulses++;
    instance->lastPulseTime = currentTime;
}
```

Image 2.11: Code executed during an interrupt

This code measures the time elapsed between 2 pulses in order to calculate the pulses/sec (Hz), it increments the total pulse count and updates the lastPulseTime in order to correctly calculate the values during the readData() call.

```
std::map<std::string, double> WaterFlowSensor::readData()
{
    noInterrupts(); // Ensure consistent data access
    float currentFlowRate = flowRate;
    uint16_t currentPulses = pulses;
    unsigned long timeSinceLastPulse = millis() - lastPulseTime;
    interrupts();

    // Reset flow rate if no pulse detected in the last 1 second
    if (timeSinceLastPulse > 1000)
    {
        currentFlowRate = 0;
    }

    // Calculate total liters
    float liters = currentPulses / (7.5 * 60.0);

    std::map<std::string, double> data;
    data["flow-rate-hz"] = currentFlowRate;
    data["pulses"] = currentPulses;
    data["flow-rate-liters"] = liters;
    data["liters-per-minute"] = currentFlowRate * 60.0;
    return data;
}
```

Image 2.12: Code executed during polling

In order to prevent side effects and safely read the flowRate, pulses and lastPulseTime, interrupts are deactivated, if the timeSinceLastPulse is more than a second, we can assume that the flow rate is 0 and no water is passing. The liters are calculated by a vendor-provided coefficient, for every pulse

1/(7.5*60) liters pass through the sensor. These values are then inserted to a std::map data structure to be handled uniformly by the collector class.

The last sensor with custom drivers was the water level sensor. These sensors are simple to interface with, reading a pin value to detect whether the water level is above or below the sensor. By combining two sensors, one placed at a high point in the tank and another at a low point, the system can determine whether the water level is low, adequate, or high. Using this combination, four distinct states are possible, including an easter egg state called “Gravity Reversed” triggered when the high sensor reports high but the low sensor reports low, which is an impossible condition unless the tank is inverted. The numeric values a returned by the driver encode these states: -1 for low, 0 for adequate, 1 for high and 101010 for “GravityReversed”.

```cpp
bool WaterLevelSensor::isWaterLevelLow()
{
    return digitalRead(lowPin) == LOW;
}

bool WaterLevelSensor::isWaterLevelAdequate()
{
    return digitalRead(lowPin) == HIGH && digitalRead(highPin) == LOW;
}

bool WaterLevelSensor::isWaterLevelHigh()
{
    return digitalRead(highPin) == HIGH;
}

bool WaterLevelSensor::isGravityReversed()
{
    return digitalRead(lowPin) == LOW && digitalRead(highPin) == HIGH;
}

double WaterLevelSensor::waterLevelIs()
{
    if (isWaterLevelLow())
    {
        return -1.0;
    }
    else if (isWaterLevelAdequate())
    {
        return 0.0;
    }
    else if (isWaterLevelHigh())
    {
        return 1.0;
    }
    else if (isGravityReversed())
    {
        return 101010.0;
    }
    return 101010.0;
}
```

Image 2.13: Float Sensor custom code for handling

2.3 Services

2.3.1 3rd Party Dependencies

The Auto Harvest system software is composed of both custom-built components (firmware, backend, frontend) and several essential third-party services that provide runtime infrastructure. These services are managed using Docker Compose for local development and provisioned as managed resources in Google Cloud Platform (GCP) for production. They serve key roles such as messaging, persistence

and container networking, allowing modular and distributed behavior across microcontroller devices and cloud-hosted APIs.

2.3.2 MongoDB Service

MongoDB is a fast NoSQL document-based storage solution and is used for log and entity storage purposes. Standard collections are used to express the User, Controller, Notification, Historic collections, and Time-series collections are used to store sensor logs and other telemetry logs (such as error messages). Time-series collections allow efficient queries over timestamped telemetry records. Locally, the service is utilized using this Docker Compose configuration.

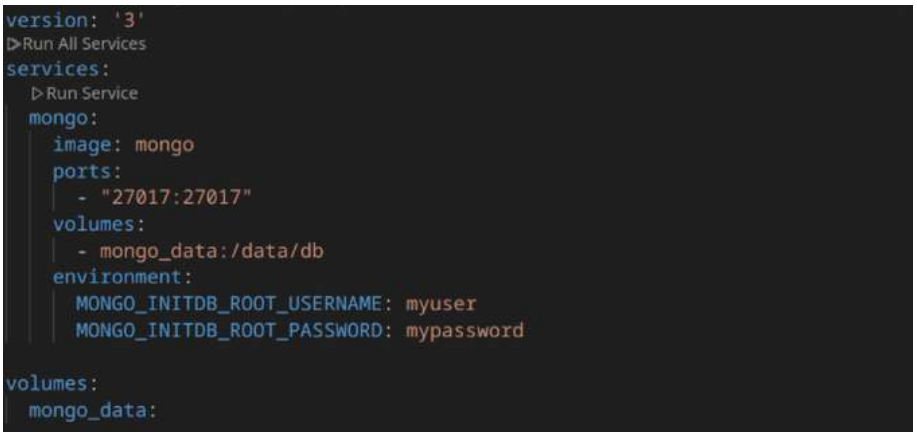

Image 2.14: Docker compose file for the MongoDB service

This configuration supports data persistence (mongo_data volume), dev credentials and port mapping. The official mongo image is utilized.

In the production environment, the same configuration is used, but the service is, of course, not exposed to the public and the address can only be resolved in a “local context”. The service runs in a self-hosted container within the GKE cluster rather than as a managed GCP service. This approach allows us to reduce costs and have full control over resource allocation. While this setup is not fully production-ready, it was chosen to allow rapid development within limited funding constraints.

2.3.3 ActiveMQ Service

ActiveMQ is the messaging hub of the Auto Harvest system. It enables asynchronous, real-time communication between devices and the backend using MQTT, STOMP, and OpenWire protocols. The ActiveMQ service is set up using the following Docker Compose configuration file

Image 2.15: Docker Compose file for the ActiveMQ service

The configuration exposes a port to communicate directly to the broker using the OpenWire protocol on port 3009, a web interface on port 3010 and the MQTT port at 3011 where both firmware and backend clients subscribe/publish to.

In production, ActiveMQ is deployed inside the GKE cluster, exposed via a dedicated LoadBalancer service on port 1883 mapped to the 3011 port, secured by Cloudflare and being constantly monitored by internal health and liveness checks. Only authorized developers can access the monitoring console by creating short-lived web-tunnels (e.g., internal.gcp.com/mq:3010→localhost:3000) Future improvements include adding TLS-based firewall rules to restrict connections exclusively to firmware clients, further strengthening the resilience and reliability of the self-hosted service.

2.3.4 Service Orchestration

In both local and cloud environments, these services are networked together via docker compose or Kubernetes services. They are accessed via environment variables in the case of the frontend and the backend, or are injected during runtime (in the case of the firmware). The local development stack simulates production topology to ensure identical behavior, while in GCP, load balancing, auto-scaling and firewalling are managed by Kubernetes and Cloudflare.

2.4 UI

2.4.1 Repo Overview

The user interface of **Auto Harvest** was designed to offer a responsive experience for managing hydroponic systems. It is developed using React Native and the Expo framework, which allows the application to run seamlessly on both mobile and the web. Navigation within the app is handled using Expo Router, which simplifies screen transitions and URL based routing while supporting deep linking. The interface supports multiple controllers, allowing users to register different systems and monitor them independently. Each screen is designed with clarity and usability in mind. The dashboard provides realtime sensor readings such as pH, temperature, TDS, and water flow, while chart components build with react-native-chart-kit visualize historical data trends. These visualizations help users better understand changes over time and make informed decisions. Interaction with system components (like relays) is done through control screens, where users can toggle elements with a single tap. The app also delivers alerts via notifications, ensuring the user is immediately informed when something abnormal is detected. State is managed using Redux Toolkit, with Redux Persist ensuring that data such as user preferences and session info remain stored between app launches. The entire UI is written in TypeScript, which improves development speed and helps catch UI level logic bugs early.

Overall, the Auto Harvest UI focuses on providing accessibility, efficiency, and visibility making it possible for users to interact with their hydroponic system in real time, without needing technical expertise.

2.4.2 Navigation Flow

The navigation structure of the Auto Harvest application is built using Expo Router, which helps assure a file based routing system. The Expo router method was inspired by web frameworks like Next.js. This approach helps us a lot by making the structure scalable. The main navigation consists of a bottom tab bar that gives quick access to key screens of the application. Those screens are the Controller's **Dashboard** which shows all the metrics at one place, the **Alerts** screen that has the purpose of collecting and searching the alerts received for this specific controller and the **Settings** screen in which you can control some options for the controller. Each screen corresponds to a physical

file inside the app/ directory, and nested folders are used for sub routes, such as the specific graph views.

Image 2.16: Bottom Navigation Bar

Expo router has some features for naming files so that you can use them dynamically or even categorize and manage static files. In detail, expo router uses _layout files in which the developers define shared UI elements such as headers, tab bars etc. so that they persist between different routes [1]. In a project there could exist multiple _layout files, one of them is the root layout file in which developers inject global providers, themes, styles, delay splash screen rendering or defining the app's root navigation structure. Expo also provides a way to group navigation files together using (groupName) as a folder name, this gives the option that the group could remain hidden in the URL and you can give another _layout file with other specifications for that group. Another expo feature is the Not Found Route that navigates to a file with the name +not-found.tsx in the root app directory, that redirects to that page when a file is not found. Finally, one of the best features that Expo Router provides, in our opinion, is the use of dynamic routes. A dynamic segment of a route is created by wrapping a file's name in square brackets. For example, [id].tsx. A dynamic route allows matching one or multiple paths based on a dynamic segment embedded in the URL. This segment is in the form of a variable, such as a unique identifier.

Now that we understood the functionalities of Expo Router lets see the implementation we did using the features. As we can see from the image below (*Image 2.17*), we used every feature that Expo Router provided there is a base layout file that configures some basic options for our files and other layout files in each group of files. We can see that a [metric].tsx file is used to navigate dynamically to the graphs of each metric. By clicking on the metric of choice the app redirects dynamically to that graph.

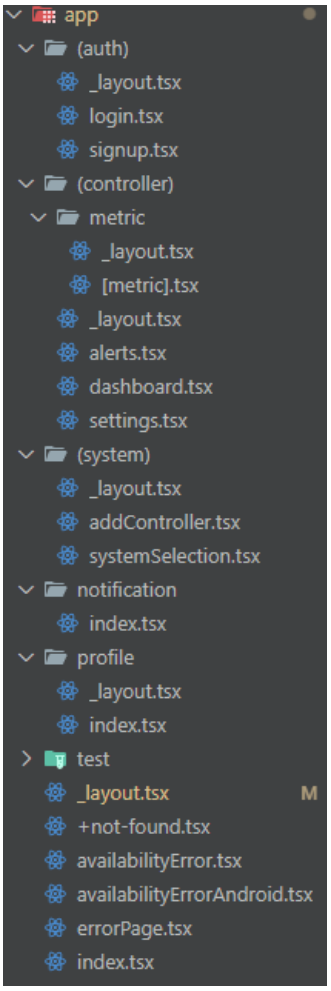

Image 2.17: File structure of the navigation

To achieve all that that were mentioned earlier the app uses userRouter() and useLocalSearchParams() hooks provided by Expo Router to navigate and retrieve route parameters, making dynamic views easy to maintain. Error screens, like unavailable functionality pages or not found pages, are also handled through specific routes. The navigation is scalable without requiring major changes.

2.4.3 Main Screens

The project has a variety of screens to help monitor and manage the systems. Each screen has a specific purpose in the app. The **Dashboard** screen is the cockpit of the user. It displays real time data from the connected controller and interacts with that controller if needed to turn on and off some sensors. Each controller is shown in a card-like format, in which each card can be pressed to navigate you to a graph of that metric so that you can view the historic data of the metric. If any sensor reports an abnormal reading, a visual warning is triggered on the card, notifying the user to take action.

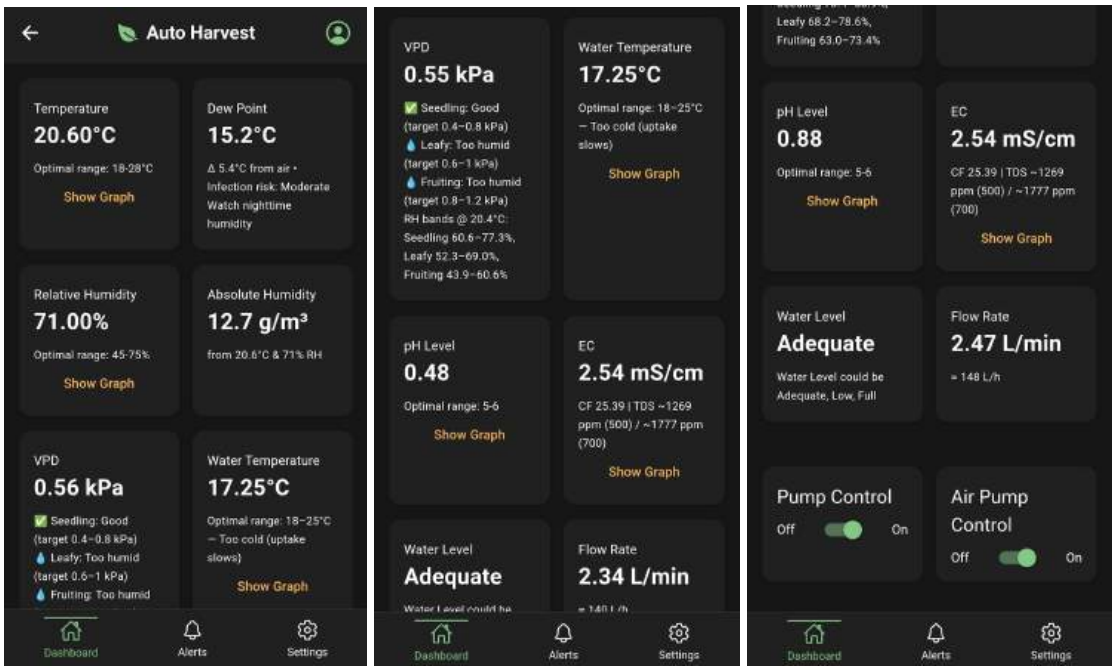

Image 2.18: Dashboard Page

Graph screens are what is expected, a screen that shows a graph of a metric. The screen provides historical data visualization for each metric. The graphs were made by using react-native-chart-lit, it allows users to view trends in different groups (hour, day, week, month). This helps with identifying if any anomalies occurred at some point so that the user can further investigate what happened to the system.

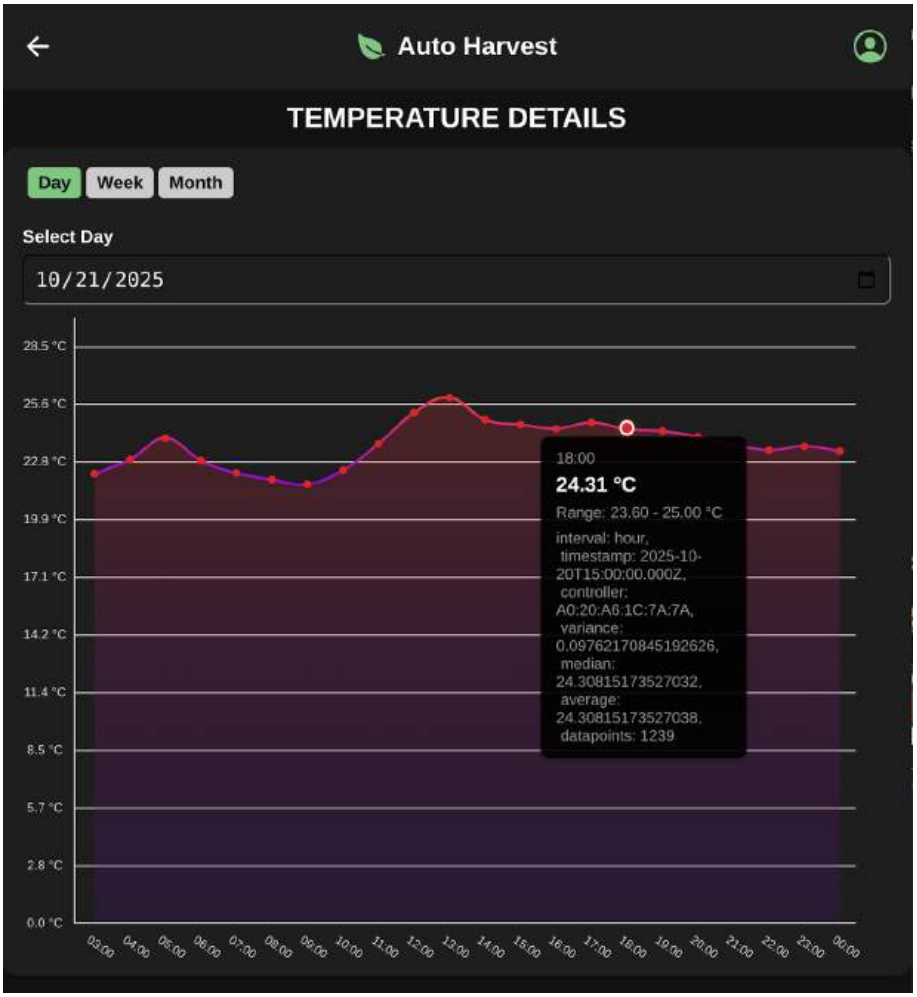

Image 2.19: Air Temperature Graph of a day

Alerts screen gathers all the warnings and alerts (connection issues, overflow, abnormal PH level) and displays them here. Each alert is displayed with a category tag, the context of the alert and a timestamp. For “Warings”, notifications are also pushed to the device if a critical change occurred while the app is closed.

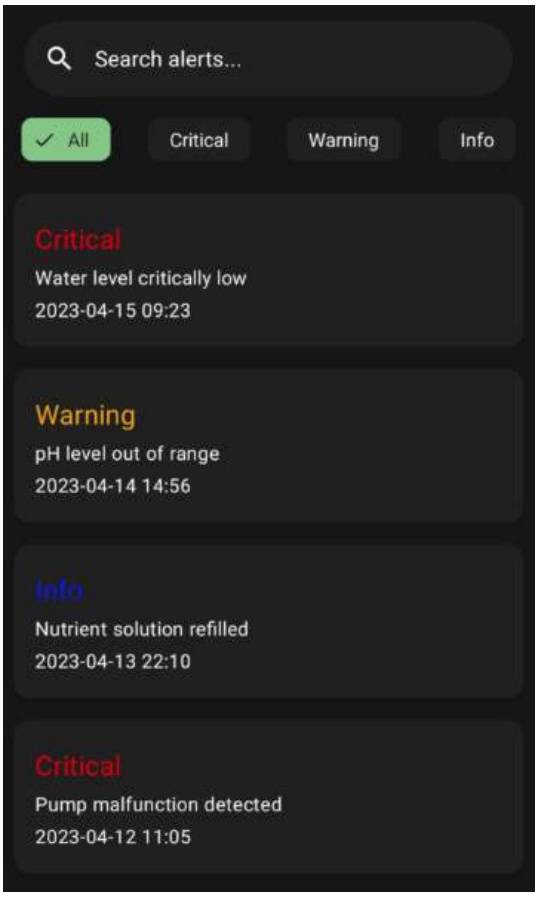

Image 2.20: Alert screen

Controller Registration screen allows users to add a new controller to their profile. After some steps of ensuring that the controller is into “pairing” mode, allowing location services, and connecting it to the local Wi-Fi, the user adds some details for the given controller. The app links with the device to the user’s account and begins pulling data from it. Its a simple process, even for non-technical users.

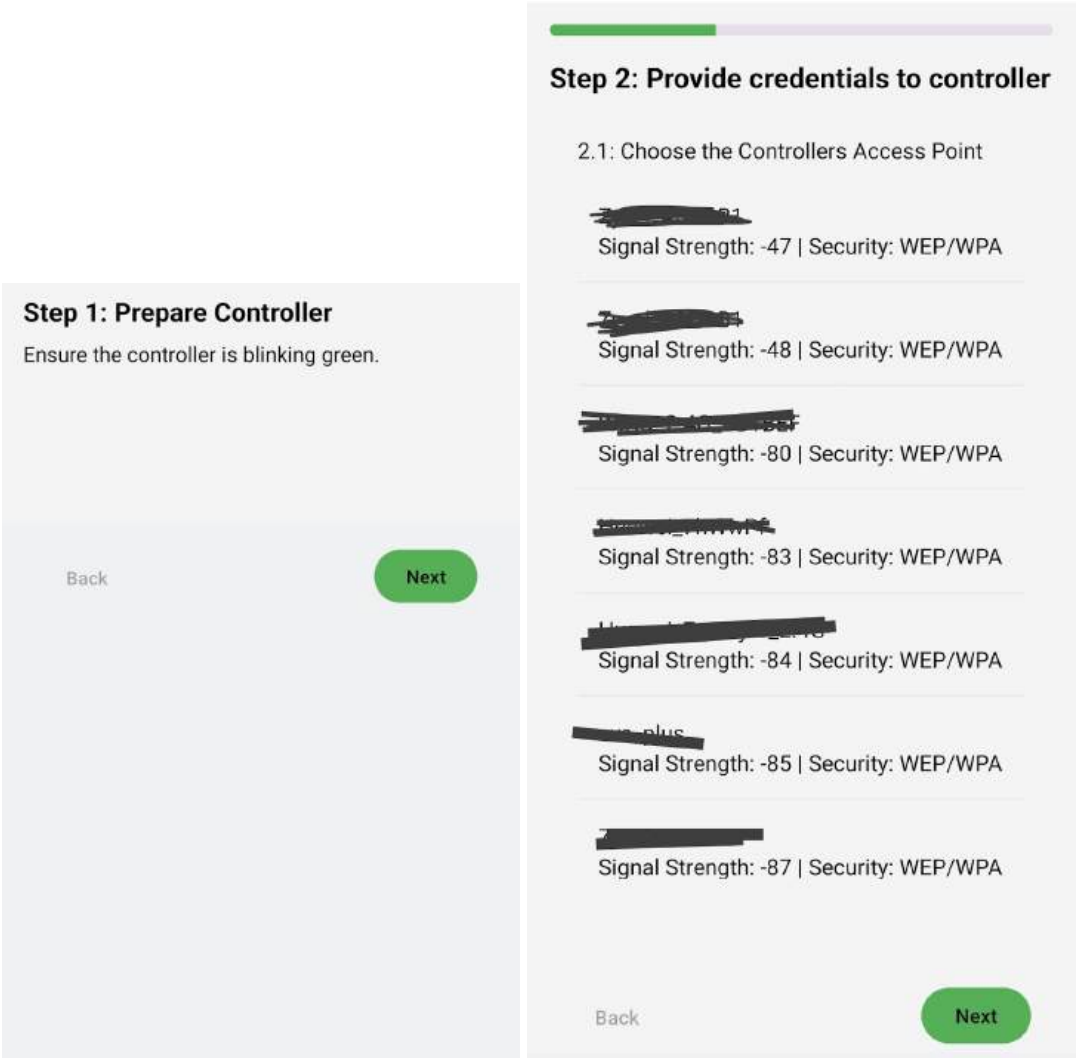

Image 2.21: Controller registration steps

2.4.4 State Management

State management in the Auto Harvest application is handled using Redux toolkit, which provides a scalable way to manage global application state across screens and components. Redux is essential for maintaining consistency in real time sensor data, user session management and controller interactions. We use redux toolkit to define slices of state for controllers, sensors, authentication, and UI preferences. Each slice contains its own actions and reducers, making the code modular and easy to maintain. For example, the controllerSlice manages the list of active controllers and their sensor readings, while userSlice tracks login status and user related settings.

Sensor data and backend communications are handled asynchronous using createAsyncThunk, which allows us to perform network requests and update the state only after responses are received. This ensures the UI always reflects the most recent system state without blocking user interaction. To ensure important data is retained between app session, we use Redux Persist. It automatically saves the Redux state to device storage and rehydrates it when the app restarts. This is especially important for maintaining controller associations and session tokens when the user closes the app or switches between devices.

All UI components use useSelector to access current state values and useDispatch to trigger actions. This keeps data flow centralized and avoids redundant state declarations across components. Updates to state are reactive, meaning any screen that depends on the state will automatically rerender when the data changes. This architecture provides a robust foundation for managing real time, multi controller systems and ensures the app remains responsive and consistent, even in complex user flows.

2.4.5 Styling & Responsiveness

The design of the Auto Harvest user interface was approached with the intention of maintaining visual consistency across platforms, while ensuring that the layout could adapt to different screen sizes and devices. To achieve this, a restrained design system was adopted that defines a clear set of colors, typographic scales, and component behaviors. These together help the application appear uniform regardless of whether it runs on a mobile device or through the web. The color palette follows a neutral base, while status information such as warnings or errors is highlighted through accent tones, ensuring that important conditions stand out to the user without overloading the interface. The application also automatically adapts to the device's light and dark mode, allowing it to remain legible and visually coherent in different environments.

The layout system is built on React Native's flex box model, which avoids the use of hard coded pixel values and instead uses proportional sizing to allow the interface to scale across devices with varying resolutions and aspect ratios. Runtime dimensions are taken into account so that charts and data grids can expand or contract depending on the available space, and safe-area handling ensures that content does not overlap with modern device features such as notches or status bars. This flexibility allows the application to maintain its usability both in portrait and landscape orientations.

Typography was chosen to be simple and readable, using a base font size that is comfortable for mobile reading and scaling headings and body text is a consistent ration to create hierarchy. Icons are applied sparingly, only in cases where they provide an immediate recognition of status or actions, and they are always accompanied by text labels to avoid relying on visual symbols alone. Accessibility was considered by ensuring that text contrast remains adequate, that interactive components provide sufficient spacing and touch area, and that error or empty states are explicitly communicated to the user with guidance for recovery rather than leaving screens blank.

Data visualization is a critical aspect of the interface, since sensor information forms the backbone of the system. Charts are designed to resize dynamically with the screen width, ensuring they remain clear and legible across devices, while axis labels and data density adjust to avoid overlaps. Important thresholds, such as acceptable pH or EC ranges, can be represented with subtle visual guides, and outlier are emphasized so that the user can identify issues quickly.

Finally, the design also accounts for the differences between mobile and web usage. On mobile devices, the interface prioritizes touch based interactions with larger input fields and native pickers, while the web version makes use of hover states and keyboard navigation to enhance efficiency. Performance considerations are present across both platforms, with lists and dashboards optimized to render only what is visible on screen to ensure smooth operation even on lower end devices. By following these design principles, the Auto Harvest user interface succeeds in offering an experience that is no only functional and responsive but also accessible and user friendly across different platforms and usage scenarios.

In conclusion, the design of the Auto Harvest user interface brings together principles of usability, accessibility, and technical efficiency to create an application that is both practical and adaptable. By relying on modern frameworks such as React Native and Expo, combined with robust state management through Redux, the system achieves consistency in performance while maintaining clarity in its presentation of critical sensor data. The navigation flow ensures that users can easily move between different screens, the main views focus on delivering essential information with

minimal effort, and the styling choices allow the interface to scale across a wide range of devices and environments. Ultimately, the UI serves as the primary point of interaction between the user and the hydroponic system, and its careful design ensures that the monitoring and management of complex processes are presented in a simple and accessible way.

Section 3: Hardware

3.1 Components

The system is developed around a modular set of electronic components designed to monitor environmental and chemical parameters while automating water and air control processes within a hydroponic setup. The current build uses a development prototype using an Arduino Mega 2560 with a Sensor Shield V2.4, for development testing and fast reconfiguration. The current hardware configuration is intended only for development and testing. In a production-grade deployment, the system would transition to a more compact, cheap and energy efficient design, most likely using the ESP32 microcontroller, that offers Wi-Fi, bluetooth, higher processing capabilities and Over-The-Air (OTA) updates. A custom PCB would also be designed to integrate the necessary sensors and connectors, so the complete board can be ordered prebuilt.

The Arduino Mega 2560 acts as the main microcontroller in this prototype, handling sensor readings, relay control and serial communication, while being expanded by the DFRobots Sensor Shield V2.4 which expands the I/O and powering capabilities, as well as offering support for MicroSD log dumping and support for ZigBee devices.

All the sensors used are listed in the table below:

Table 3.1: List of sensors used

Sensor	Function	Type	Interface	Voltage
DHT11	Measures ambient humidity and temperature	Digital	D16	5V
DS18B20	Measures water temperature	Digital	D17	5V
SEN0161	Measures the pH of aqueous solutions	Analog	A6	5V
SEN0244	Measures the TDS in aqueous solutions	Analog	D40	5V
Water Flow Sensor	Measures water circulation rate	Digital	D40	5V
Water Level Switches	Detect low/high reservoir levels	Digital	D14/D15	5V

These sensors are used to monitor core hydroponic parameters in real time. The analog sensors require calibration and conditioning to ensure data accuracy, which is handled in software, while the digital sensors are ready to be used out of the box.

The actuation and control modules are listed below:

Table 3.2: Control modules

Module	Function	Controlled by	Interface	Notes
4-Channel Relay (DSP MSP430)	Switches pumps and AC loads	Digital	D24-D27	Isolates 220V AC control
Water Pump Modules	Circulates nutrient solution	Relay	D24	Corresponding interface is controlled via Relay
Air Pump Module	Oxygenates solution	Relay	D25	Corresponding interface is controlled via Relay

These modules provide automated actuation for vital hydroponic mechanisms, while also providing the ability to control high voltage electrical appliances by software.

For communication and output the following modules were used:

Table 3.3: Communication modules

Module	Function	Interface	Voltage	Pins
ESP8266 (connected via ESP-01)	Wi-Fi communicatio n	Serial	5V	COM1
Waveshare 1602 LCD	Local display of readings and status	I2C	5V	SDA: D20, SCL: D21

The ESP8266 allows the board to transmit data wirelessly to a cloud backend. The Waveshare 1602 LCD serves as a user feedback device for debugging and local offline display. The architecture is designed with future hardware abstraction in mind, allowing us an easy migration to ESP32-based boards and a custom daughterboard layout for sensor integration and housing.

3.2 Power, Connections and Wiring

The current connectivity and wiring of the system is designed for modular prototyping, easy open public demonstration and accessibility. While the current prototype serves for display purposes only, the power cables were carefully spliced and wrapped with shrinkwrap, and custom Dupont connectors were used to give the modules a plug-and-play feel, which helped us in rapid prototyping. All component connections follow the pin mapping shown. The MEGA Sensor Shield V2.4 provides labeled breakout headers for analog, digital and I2C pins, paired with 5V and GND breakouts, eliminating the need for jumper wires and breadboards.

Sensors/Modules	Serial	Code	Voltage	Type	Pin/Powered By
Water level low switch	N/A	wl	5V	Digital	D14
Water level high switch	N/A	wh	5V	Digital	D15
Wifi module	ESP8266 /w ESP-01	wifi	5V	I2C	COM1
Water pump module	N/A	wp	12V	Analog	r_1 (D24)
Air pump module	N/A	ap	12V	Analog	r_2 (D25)
Humidity/Air temperature sensor	DHT11	h	5V	Digital	D16
Water temperature sensor	DS18B20	wt	5V	Digital	D17
Ph sensor	SEN0161	ph	5V	Analog	A6
TDS sensor	SEN0244	tdp	5V	Analog	A7
4 channel relay module	DSP MSP430	r_1, r_2, r_3, r_4	5V	Digital	(r_1-r_4) -> (D24-D27)
Water flow sensor	N/A	lpm	5V	Analog	D2
LCD module	Waveshare 1602		5V	I2C	SCL: D21, SDA: D20

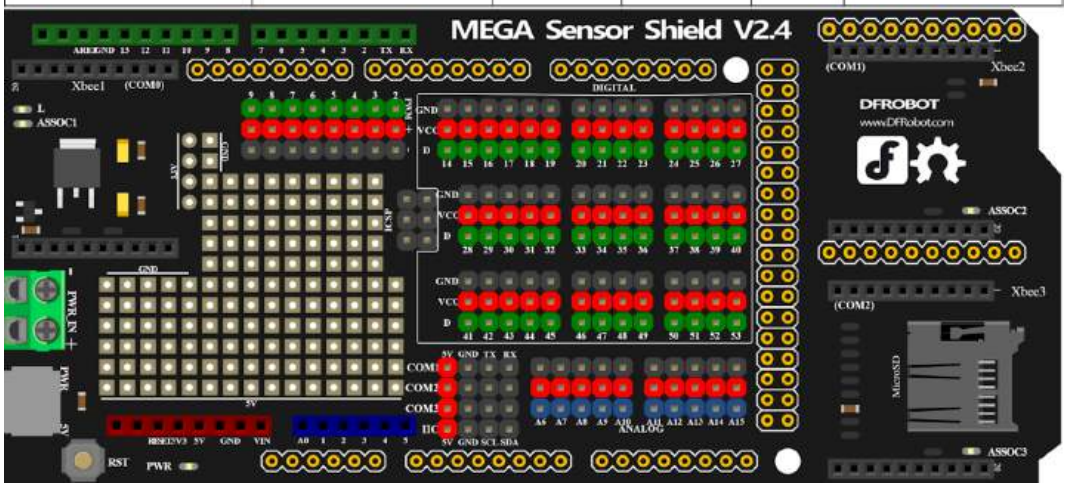

Image 3.1: Wiring reference table with a picture of the Shield used for prototyping

The system is powered through a 12V 2A DC rotary switch power supply, which feeds into the Sensor Shield and the relay-connected modules, it is split into three branches to simultaneously power the arduino mega, the water pump and the air pump. All power connections are done via direct lines and securely fitted Dupont connectors. To ensure future compatibility with 3.3 V-based microcontrollers like the ESP32, a step-down voltage regulator is integrated into the layout. This allows the system to switch to lower voltage while keeping the actuators supply stable.

All three main communication interfaces (analog, digital, I2C) are used simultaneously, but during early integration a few challenges were encountered. The flow sensor that uses hall effect, required an interrupt-capable pin to measure pulses accurately. Since not all digital pins on the Mega support interrupts, and the manual didn't note which pins supported hardware interrupts, the pins were selected by trial and error. Also the float sensors experienced perioding signal instability, which was resolved by enabling internal pull-up resistors, stabilizing the voltage levels.

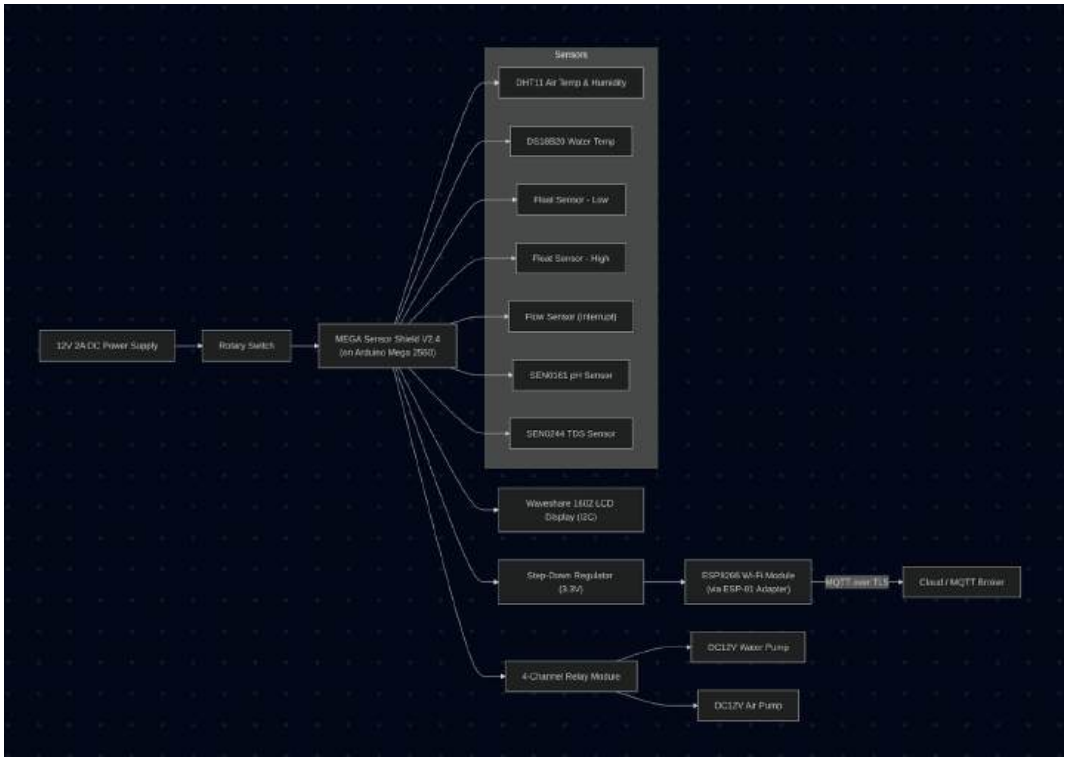

Image 3.2: Connection diagram of major hardware components

The prototype hardware is mounted on a hard insulating display board, arranged to expose each component clearly for demonstration and educational purposes. The wires are routed and fasted using heat shrink tubing, to reduce clutter and tangling and nail anchors to secure the modules and shields in place.

3.3 Sensors

The system incorporates sensors that enable real-time environmental, chemical and fluidic monitoring. These sensors provide essential data for maintaining healthy crop conditions in a hydroponic setup, such as pH levels, water temperature, humidity and flow rate. Each sensor was selected for its simplicity, availability and relevance to the project's goals, though calibration and custom signal handling required experimentation and tuning beyond the typical datasheet expectations. A categorised overview of the sensors will be listed in the section below.

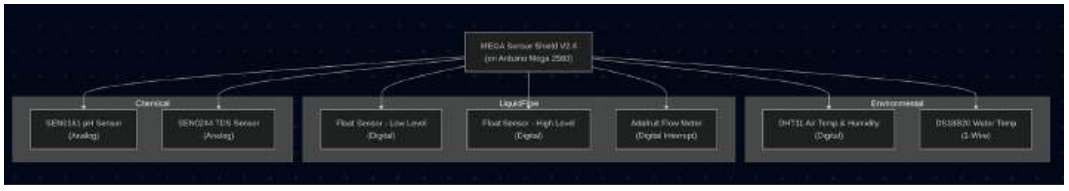

Image 3.3: Connection diagram of the sensors

Environmental monitoring sensors:

- DHT11: Air Temperature and Humidity [9]
  A simple digital sensor used to track the ambient conditions around the grow area, its

positioned above the water line and provides two values, relative humidity (in %) and temperature (in °C). The values are filtered in software to remove jitter from occasional faulty reads.

- DS18B20: Water Temperature [10]
  Waterproof digital sensor using 1-Wire protocol. It is submerged in the nutrient solution reservoir and provides high-resolution temperature data with low noise and minimal drift. Its manually calibrated against a lab thermometer during testing.

Liquid and Flow monitoring:

- Electronic Spices P43: Liquid Level Float Sensors (x2) [11]
  Simple reed switches that close when the buoyant part of the sensor that encapsulates a magnet floats upwards and closes the reed switch located at the top part of the sensor. They are installed in low and high positions inside the tank. Initial testing showed signal noise and oscillation which was resolved by enabling internal pull-up resistors in code [13].
- Adafruit Liquid Flow Meter: Liquid volume sensor [12]
  Measures water circulation rate through a pulse count from a hall sensor. It's connected to a pin supporting hardware interrupts to avoid missed pulses. Timing accuracy is very important for the volume/time to be measured correctly, all code execution is paused except the interrupt logic when it is triggered. The output was validated by passing 1L of water through the sensor and calculating the total volume passed.

Chemical monitoring:

- SEN0161: pH Sensor [14]
  Analog sensor used to determine the acidity/alkalinity of a solution. The provided library was found unreliable as the intended behavior was inverted (acidic solutions read high pH values). A custom calibration approach was adopted, it used a two-point calibration method (distilled water at 7.0 pH and acidic buffer solution at 4.03 pH), its details will be explained further in the firmware section.
- SEN0244: TDS Sensor [15]
  Analog sensor that measures the electroconductivity of liquids. It too required temperature compensation for accurate readings. Its outputs were compared with a commercial EC meter in a calibration solution.

3.4 Modules

The system isn’t just monitoring environmental values, it also controls key hydroponic components (such as the water pump and the air pump through a 4 channel relay module), displays informative and critical messages through the 16x2 LCD display and connects to the internet using the ESP8266 Wi-Fi module. Like in the sensors’ section, a categorised overview of the utilized modules will be presented below.

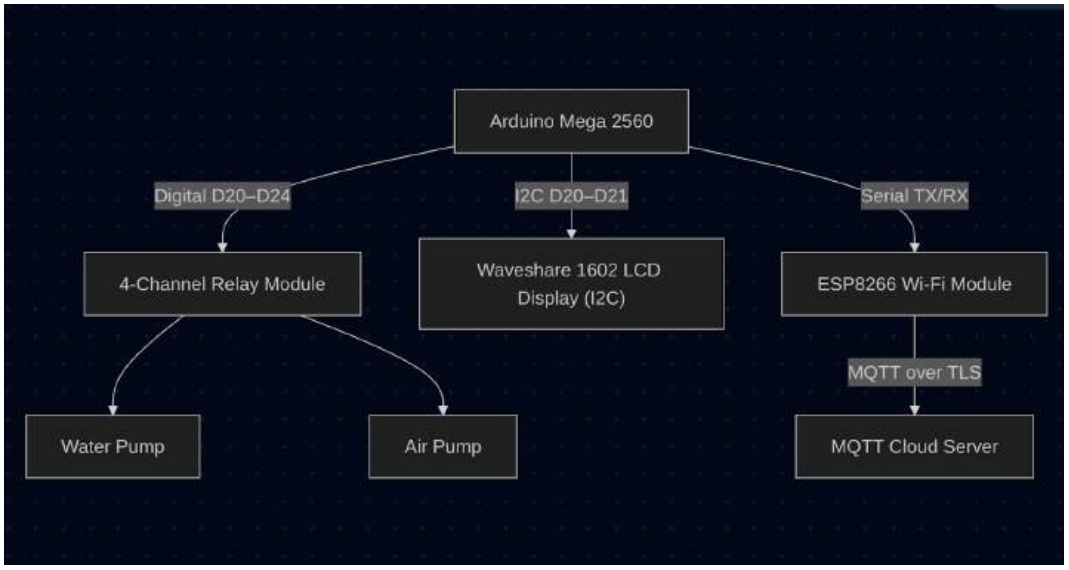

Image 3.4: Connection diagram of the actuators and communication devices

Actuator modules:

- **DC12V AD20P-1230A**: Water Pump [16]
  A brushless 12V DC water pump capable of pushing 240L/h up to 3 meters vertically. According to the manufacturer it is resistant to weak acids (>5pH) and weak bases (<10pH) making it ideal for small hydroponic setups. It is energy efficient, consuming only 3.7 Watts. It is connected to the first channel of the 4-channel relay module.
- **Mini Air Pump 12V DC** [17]
  A compact and energy efficient air pump capable of pumping 2 liters of air per minute. It is relatively silent (at 45dB) and doesn't vibrate much. Its cost effectiveness and adequate gas flow made it suitable for our needs
- **Relay Module - 4 Channel 5V** [18]
  A 4 channel relay module that controls the other actuators. It is connected to the Arduino at digital mode pins D20-D24. Each relay supports 10A draw and 250V which makes it suitable for controlling heavier appliances like water/air heating elements and artificial lights.

Display modules:

- **Waveshare 1602**: LCD Display [19]
  A minimal 16x2 used to display informative, critical and debugging information. It uses I2C for communication with the Serial Data pin connected at D20 and Serial Clock pin connected at D21

Communication modules:

- **ESP8266**: Wi-Fi module [20]
  The ESP8266 is a low-cost Wi-Fi microchip, working at 3.3V, with built-in TCP/IP networking software allowing the Arduino ATMega2560 to use it in order to connect to the local Wi-Fi or act as an access point to initiate the pairing process.
- **ESP-01**: Wi-Fi module adapter[21]
  To simplify the connection of the ESP8266 module to the ATMega2560 microcontroller, the ESP-01 adapter was used, since it can connect with the 8pins of the ESP8266 and provide just 4 pins at 3.3V-5V (VCC,GND,Tx,Rx), making it much easier to operate the Wi-Fi chip.

3.5 Calibration

Chemical sensors like the pH and TDS sensors are susceptible to drift and environmental noise. Calibrating them was important in order to have accurate readings, since the plants' roots do not tolerate significant variation from the optimal pH and TDS values at certain and all stages of growth.

The SEN0161 pH sensor after it initially exhibited reversed as discussed in the firmware section 2.2.6 and custom drivers were developed for it, it was calibrated using distilled water (pH 7) and an acidic reference solution (pH 4.03) verified with pH testing strips. The calibration parameters are stored persistently in the EEPROM, so they survive power cycles and don't require re-entry after restarts [14]. Sensor readings are smoothed using averaging to reduce fluctuations caused by analog noise

The SEN0244 TDS sensor measures electroconductivity to measure nutrient strength in ppm, calibration was performed by checking the values of the sensor when submerged in distilled water (0ppm) and EC meter[24] calibration solution (883ppm).

Digital sensors used for air and water temperature as well as humidity (DHT11, DS18B20) were indirectly verified using consumer-grade tools. The DS18B20 water temperature sensor was submerged in the same water solution as a food thermometer and their values were observed to be within 1 degree of deviation. The DHT11 ambient humidity and temperature sensor was tested in the same room with a commercial weather station, all values were within acceptable error margins.

No user accessible calibration menu exists yet, but a future update will allow app-guided calibration steps, by having the mobile app to prompt the user to apply known buffer solutions and enter their values for calibration. This way the sensors are going to be utilized until their readings are too inaccurate to be calibrated, making the system maintainable over time. Since user-side calibration is a software issue it makes it easy for us to include it in the same hardware configuration as a future firmware update.

Below we provide a table where we compare the measured values of our sensors to aftermarket equivalent sensors to test the variation and precision of our readings and to perform calibrations as needed, the only missing sensor from the table is the pH sensor that was damaged while trying to gather its sensor values for comparison and as a result it became unusable. The only sensor that required calibration was the TDS meter, which needed its slope corrected as it displayed correct results for low TDS values (distilled water) but drifted considerably on salinity concentrations equivalent to late vegetative growth medium. As seen in the values of the table below, the TDS meter was more sensitive the higher the salinity was, after the calibration was performed, the sensor values were acceptable both in low CF values and high CF values.

Table 3.4: Sensor modules compared to aftermarket sensors.

Sensor Type	Condition / Sample	System Reading (A)	Reference Reading (B)	Deviation (A − B)	Result
TDS (Before Calibration)	Distilled Water	0 CF	0 CF	0	Acceptable
	Mid-range	6 CF	5 CF	+1	Minor deviation
	Higher concentration	13 CF	9 CF	+4	Needs calibration
TDS (After Calibration)	Distilled Water	0 CF	0 CF	0	Acceptable
	Mid-range	5 CF	5 CF	0	Accurate
	Higher concentration	10 CF	9 CF	+1	Acceptable
Ambient Temperature	Sample 1	25.4 °C	25.7 °C	−0.3 °C	Acceptable
	Sample 2	23.6 °C	23.8 °C	−0.2 °C	Acceptable
Ambient Humidity	Sample 1	70%	68%	+2%	Acceptable
	Sample 2	68%	67%	+1%	Acceptable
Water Temperature	Single measurement	36.55 °C	36.63 °C	−0.08 °C	Acceptable

3.6 Maintenance and Replacements

Even though the system is designed for autonomous operation, regular maintenance is necessary to ensure long-term sensor accuracy, mechanical reliability and safe operation.

The pH probe has a limited lifespan and is the most sensitive sensor in the system. If not kept hydrated or stored correctly, it may degrade irreversibly. Replacement is recommended once accuracy drifts beyond +/- 1pH level, even after recalibration. The SEN0244 TDS probe and DS18B20 temperature probe are susceptible to corrosion if left exposed to air with residual nutrient solutions. If calibration no longer yields stable readings within +/- 50ppm for the TDS sensor or +/- 0.5C for the DS18B20 sensor, replacement is recommended. That said, all probes are socketed and hot-swappable. If unplugged the system identifies them via out-of-bounds readings and automatically ignores the associated input (like in the case of the pH sensor). In the final enclosure design, all sensors and actuators will be connected via mini Molex connectors, allowing quick disconnection and replacement without the need for soldering or disassembly

If the following cleaning protocols are followed correctly, the sensors lifespan will not be diminished:

- pH probe: The pH probe should be cleaned only with distilled water to prevent damage. After cleaning, it should be stored in a 4M KCl ion-rich solution in its cap to maintain membrane saturation and accuracy[14]. Scrubbing or abrasive cleaning is strongly discouraged. The glass ball in the center of the probe should be touched or scratched because every change to the shape of it is going to affect the readings and require recalibration, if not damaged severely enough.
- TDS, Temperature, Float and Flow Sensors: These sensors should be cleaned with food-safe agents such as diluted hydrogen peroxide[25], which is effective for removing organic buildup and algae while also acting as a disinfectant. A warm water rinse is recommended afterward to remove any residual peroxide or larger debris

The sensors should never be cleaned with ionic surfactants (dish soap, hand soap etc.) as any residue mixed with the nutrient mix is going to damage the roots of the plants. Since residues from synthetic fertilizers are water soluble, and peroxide is good at cleaning organic residues[25], these options are considered the best for maintaining all hardware and material components of the system.

Users are expected to rinse or clean the sensors monthly, or at least once per crop cycle. Recalibrate chemical sensors after cleaning or when values drift, replace sensors that cannot be recalibrated within expected error margins and periodically inspect tubing and flow systems for clogs or buildup. Instructional support for cleaning and maintenance will be delivered via the mobile app in future releases as well as instructional videos when the production system is ready.

Section 4: Equipment

The construction of the Nutrient Film Technique (NFT) hydroponics system required a selection of materials and tools to ensure stability, functionality and maintainability. The goal of our project was not only to build a working prototype but also to demonstrate its potential with modern monitoring systems. We kept in mind that the physical components that were going to be added could be replaced and also easy to connect with sensors and software. The materials had to have a strong withstanding with continuous exposure to water and nutrient solutions. Since this was a student project everything had to be cost-effective. In this section, it is described the components and tools we used, the process of building the construction, the maintenance tasks that should be followed and the interoperability features that allow it to expand and connect with the Auto Harvest software.

4.1 Components and Tools

For constructing the NFT (Nutrient Film Technique) hydroponic system, we gathered both the structural materials and the necessary equipment for assembly. The main construction of the system consists of a PVC pipe serving as a growing channel, in which the water flows through, with drilled holes on top of it to hold the net pots for the plants. As the base of the setup is a nutrient solution reservoir equipped with a recirculation pump and sensors. All components are connected with durable tubes and adaptors, which are supported by a frame that provides the proper slope for the nutrient film. Various tools were used to build and ensure the safe operation of the system. Below is an overview of the main components and tools used:

System Components:

- **PVC Grow Channel**: The plant growing channels are made from PVC pipes with eight holes that have been made by us using a drill and a hole saw bit to accommodate the net pots. These pipes form the core structure where plant roots are exposed to the flowing nutrient solution. The PVC material was chosen because it is light weight, durable, resistant to algae making it durable using nutrient solutions. The channels are arranged with a slight slope of 1:30, so that the nutrient solution flows slowly and does not overflow the pipe. That gentle slope also ensures that the roots of the plants always receive a constant supply of nutrients and oxygen as the solution passes by.
- **Reservoir**: A plastic reservoir of about 10-12 liters capacity (high durability black plastic storage box) is used to hold the nutrient rich water solution. This container was selected for its toughness, meaning it will not react with or leach substances into the nutrient solution. The reservoir is placed at the lowest point of the system so that the unused nutrient solution can drain back into it by gravity. The float sensors are installed in the reservoir to monitor the fluid level. These sensors provide an indication or trigger a cutoff when the nutrient solution is low, preventing the pump from running dry.
- **Water Pump**: Inside the reservoir, a submersible water pump (12V 1A DC) is installed to recirculate the nutrient solution through the system. The pump (a small fountain/aqurium-type pump) is powerful enough to continuously push the solution up from the reservoir to the high end of the inclined channels. It was chosen to have an adequate flow rate to maintain a constant thin film of nutrients along the channels without overwhelming the plant's roots. The pump's specifications were matched to the system's size, to ensure reliable and continuous circulation.
- **Air Pump**: In the reservoir there is also an air pump connected to an air stone inside the reservoir. This pump ensures that the nutrient solution remains oxygenated, which is critical

for root respiration and overall plant health. By constantly making air bubbles into the water, the air pump created a better environment for nutrient uptake.

- **Tubing and Fittings**: Flexible plastic tubing is used to carry the nutrient solution from the pump to the channels and back to the reservoir. A 12.5mm inner diameter garden hose serves as the main supply line, which then connects to narrower 9mm tubing, in some parts, for distributing the flow into the grow channels and for the return line. These hoses and tubes are secured and connected with adapter connectors to ensure a leak proof and continuous circuit. Quick connect fittings and end caps were used where needed to allow easy disassembly and maintenance of the system.
- **Net Pots**: Plastic net cups are fitted into the holes on the PVC channel to hold the plants in place. The net pots used are small (around 5cm in diameter, sized to fit the pipe openings) and allow the plant roots to dangle freely into the flowing nutrient film. Each net pot is filled with clay pebbles, which support the seedling's base and retain moisture without soil. The net cup and clay pebble setup keeps the plant stable and exposes the majority of its roots to the nutrient solution, enabling efficient uptake of water and nutrients.
- **Support Frame (Base)**: To achieve the required incline for the NFT channels, a custom support structure was built. We constructed an iron frame that holds the PVC pipes at the correct angle and height. This frame is strong enough to support the weights of a few pipes, plants and circulating water. It was assembled to provide space for 2~4 channels connected together. The metal stand was measured and bolted together to ensure stability and alignment of all channels.
- **Sensors and Monitoring**: In addition to the main hydroponic hardware, the system includes several sensors that allow us to monitor both the environment that the plants are growing into and also the solution. Notably, two float sensors were installed inside the reservoir to continuously monitor the nutrient solution level and are connected to a control system that alerts us if the solution level drops too low, prompting a refill. Beyond that, there is also a humidity and air temperature sensor placed outside the system on the support frame to record the ambient conditions that affect the plant growth. Inside the reservoir we can also find a water temperature sensor to always know if the water temperature is within the acceptance criteria. To have an understanding of the chemical solution, a ph sensor and a tds sensor are used to measure acidity changes and nutrient concentration through electrical conductivity. In addition, a water flow sensor was positioned on the location that the water enters the channels from the solution to verify that there is always a stream of water entering the grow system. We have also used an LCD screen that displays a basic overview of the systems stats that were gathered from the sensors mentioned before.

Tools used for assembly

- **Drilling Equipment**: A power drill with hole saw attachments was used to cut out the openings for the net pots and also the intake and exhaust holes on the PVC pipe and the solution container. Also some holes were also drilled on the base that was built.
- **Cutting and Assembly Tools**: Pliers and wrenches were used to grip and secure connectors,valves and bolts during assembly.
- **Safety Gear and Supplies**: Through the build process, safety equipment was used to protect us. Protective glasses and gloves were worn when drilling, cutting metal and handling the stone wool insulation and chemical nutrients. Other supplies included clamps and zip ties for organizing hoses, sealant tape for any connection and cleaning wipes to remove plastic and metal debris after construction.

All of these components and tools together allowed us to construct a functional NFT hydroponics system. The parts that were used ensure that the structure is easy to maintain and everything can be replaced easily.

4.2 Build Process

The building process for the structure started with heavy duty work, by preparing the PVC pipe that would serve as the main growing channel. Using a hole saw bit, we drilled eight 5 cm holes along the top of the pipe where the net cups would sit. We also added two small holes, one at the top for the intake hose that takes water from the solution, and one at the bottom on the opposite side for the outlet. To close the open ends of the pipe, we used end camps, which not only sealed it but also made it easier to take apart for cleaning and maintenance.

Image 4.1: PVC Pipe with intake and exhaust hoses

To serve as cost-effective net pots within the NFT system, we utilized standard plastic cups. Using a pyrograph, we created multiple openings along their surfaces to allow adequate water flow and root aeration, which are essential for proper nutrient uptake. After perforation, each cup was carefully trimmed and shaped so that it could be securely inserted into the NFT tube without obstructing the nutrient film or compromising the structural stability of the channel. *(Check Image 13.2 in the Appendix section)*

Once the channel was ready, we started looking into the hosing of the structure. After a bit of research, we decided to go with a basic 12.5mm garden hose, because it was affordable, easy to find and replace and also helped us with the connection of the whole system due to the standards of the garden adapters. Using the connections with the adapters made the whole system more maintainable and also gave us flexibility to move it whenever it was required.

With all the basic connections ready, we turned our focus to the reservoir. We searched for many “budget” reservoirs that we could use, ones that we had in storage, or others that were way cheap but most of them did not work for us. The ones that we had in storage were transparent and that would create a problem with algae. After a trip to a warehouse, we bought a 12 liter black plastic container and drilled openings for both the intake and exhaust lines. Inside it we placed the submersible water pump that circulates the nutrient solution, along with two float sensors that help us indicate the water level of the reservoir. The float sensors were drilled into a hard tube to stabilize them into place and then we tighten up the tube inside the reservoir to hold it into place.

At the point where the intake hose connects to the PVC channel, we mounted a flow sensor to make sure that the water was delivered as expected. Inside the reservoir we installed a ph sensor, a tds sensor and a thermometer on a small pillar, so that we could constantly take readings from them. For the environment outside we added a combined humidity and air temperature sensor so that we always have an idea of the growing conditions.

Before wiring everything together, we had to choose a microcontroller. We were lucky that a friend of ours was kind enough to lend us an Arduino Mega that he had. To make the wiring easier and more organized, we also used a shield (MEGA Sensor Shield V2.4 by DFROBOT) so that we could plug the sensors in without soldering directly onto the board. In the beginning we ordered a lot of sensors that we ended up using just a few, but trying a bunch of them helped us figure out what worked best for the project. After trial and error we were finally ready to connect everything to the hydroponic system.

The last step was building the base. We used five metal rods, bolted together, to create a steady frame that could support 1 to 3 PVC channels. With the frame in place, the sensors calibrated, and the pump running, the system was complete and ready for planting. *(Check Image 13.3 in the Appendix section)*

4.3 Maintenance

One of the things that troubled us while we builded the system, was that it had to be easy to maintain, not only because it was a student project, but also to have a better understanding of it if it meant to be used for commercial purposes in the future. For that reason, most of the components were chosen and placed in a way that makes the cleaning, replacing and upgrading very simple.

Lets start with the tubing, we chose to use hardened hoses. If one gets damaged or worn out, it can be replaced cheaply and quickly by using another garden hose of the same diameter. Because we used universal hose adapters for all the connections, the system works in “plug and play” setup. Disconnecting the broken hose and attaching the new one is very easy, also the adapters themself are very cheap and can't be easily replaced. This flexibility also makes everything easy to expand and rearrange the structure if needed.

The PVC channel was made with maintainability in mind. Both ends are sealed with screw caps, so they can easily be removed to flush and scrub the inside.

The sensors were installed in accessible locations so that they can be swapped out without effort. They can be replaced with the same sensors of the same make and model.

On top of hardware maintenance, there are a few routine checks that should be done to keep the system running smoothly. The nutrient solution needs to be checked regularly, both for its pH and its concentration, since these values can change. We also found out that it could be good for the plants to change the solution every week or two to avoid salt buildup. Cleaning the pump filter is also important, as bits of debris can clog it and reduce water flow.

Thankfully, the system is very low maintenance. Because we chose common parts, had everything be easy to dismantle and making sure everything is in reach, we managed to create a setup where maintenance is simple flow of steps, to ensure that the plants are healthy and the system flows well.

4.4 Interoperability

While designing the system we had in mind how to easily connect with other tools and more systems. That is why also the rack was made with adding more pipes to it easily, the use of universal adapters also helps with the whole process of it. Everything can be done without the need of rebuilding everything.

On the electronics side, the sensors we included can all be linked to an Arduino or similar microcontroller. This means the system can expand in ways that everything can be handled by the Auto Harvest software and make all the checks and also the nutrient additions (In the future).

Concluding this section, the materials that we chose and the way that we handled them and connected them all together created a system that is easy to maintain, flexible to adapt and able to support continuous plant growth and expandability. By making it modular and open to upgrades, the setup

serves as a working prototype for future integrations with even more advanced monitoring and automation.

Section 5: Infrastructure

5.1 Cloud Provider

5.1.1 Google Cloud Provider and Services

The deployment of the system relies on Google Cloud Platform (GCP) to deploy and manage its core infrastructure. GCP is a centralized, scalable and cost-efficient solution that provides the backbone for our backend hosting, frontend distribution and the orchestration of utilized services.

In this project we actively use a subset of GCP's core services for managing our infrastructure. For hosting the Express backend and the MongoDB and ActiveMQ services, the Kubernetes Engine was used, utilizing an Autopilot Cluster, as our demands were not complicated. Frontend assets are hosted on Cloud Storage and distributed via CDN. To store our container images that were used for Kubernetes deployments, the Artifact Registry was utilized. A special service account was set up with the IAM tools in order for GitHub to be able to run needed CICD actions with GitHub Actions. The bucket was configured with public read scopes so that everyone can request its asset files. The deployment process is handled by a custom script that automates container builds and applies Kubernetes manifests. The GitHub Actions workflow also leverages a GCP service account to securely deploy new builds to the GKE cluster. A production Kubernetes environment is currently live on GCP. A separate staging environment is run locally using a lightweight Kubernetes distribution. This separation allows rapid prototyping and safe deployment practices before releasing to production.

Administrative access to GCP resources is granted through individual user accounts. The GitHub Actions CI/CD workflow is granted access via a dedicated service account with appropriate scope permissions

Billing was managed through the free $300 GCP credit, with monitoring provided by budget alerts. The most significant cost (>90%) came from the Autopilot cluster

5.1.2 Identity and Access Management

To ensure secure and controlled access to resources within the project's infrastructure, we employed Google Cloud's Identity and Access Management (IAM) system. This section outlines the role-based access strategies used for both automation and manual operations in our production environment.

A dedicated custom IAM role was created specifically for our CI/CD pipeline automation, named GitHub Actions Runner. This role is assigned to a service account used exclusively by GitHub Actions, and was crafted to provide just enough permissions to:

- Manage Kubernetes workloads
- Handle container image operations
- Perform frontend cache invalidation
- Read/write/delete objects in Cloud Storage

Instead of assigning multiple predefined roles, we inherited permissions from the standard Kubernetes Engine Admin role and manually extended capabilities required for interacting with Artifact Registry, Cloud Storage and Google Cloud Engine's URL maps (used for cache invalidation). This approach avoids over-permissioning and keeps service accounts scoped to task-specific actions.

For Development and Operations (DevOps), we used our personal Google Accounts. Admin actions, such as testing the runner's permissions or manually triggering deployment steps, were executed via service account impersonation, using gcloud CLI. This provided a secure way to simulate the GitHub runner locally and verify pipeline behavior in a controlled environment.

In general, IAM setup in GCP was generally smoother compared to other providers (e.g., AWS, Azure) However some permissions were not so obvious, particularly those related to URL map invalidation, such as compute.urlMaps.invalidateCache and compute.urlMaps.list. These were necessary for enabling cache refreshes post-deployment and required manual discovery during testing.

5.1.3 Artifact Registry

The Auto Harvest deployment pipelines uses Google Artifact Registry as the storage for Docker images used in the Kubernetes cluster. Artifact Registry enables seamless and secure integration with GitHub Actions and Google KubernetesEngine (GKE) while supporting fine-grained access control and lifecycle management. Compared to its competition, Google's Artifact Registry was chosen due to it being already integrated in the GCP environment, and having the cost of image maintenance be covered by the free credit provided during registration.

For our project's needs, a regional Docker repository was used within Artifact Registry, co-located with our GKE cluster to minimize latency and optimize cost. Every deployment builds and pushes a new image of the backend service to this registry. Only one container image is required per deployment, simplifying version tracking and resource usage. Each image pushed to Artifact Registry is tagged using a dual-tagging approach. The "latest" tag is used for local development and testing in the staging Kubernetes cluster, and the "sha-<HEAD>" tag as a unique identifier based on the commit hash at the time of the build for production deployments. This approach ensures immutability, traceability as well as rapid testing and developing.

Images are built automatically through GitHub Actions during each commit to the main or staging branches. The CI/CD pipeline authenticates via the previously defined GitHub Actions Runner service account, which includes permissions for pushing and tagging Docker images. Once built, the image is uploaded to the Docker registry in Artifact Registry using gcloud CLI commands and Docker-native tooling via "gcloud auth configure-docker".

To keep storage costs predictable and the registry clean, a manual pruning policy is implemented where only the 10 most recent images are retained. This is enforced during every deployment.

Regarding access control, only the CI/CD pipeline (via service account) and system administrators (via impersonation) have write access to Artifact Registry. All other access is read-only or restricted based on IAM policy, and a "public user" is not granted any access.

5.1.4 Google Kubernetes Engine

The production services for Auto Harvest are deployed on a Google Kubernetes Engine (GKE) cluster using the Autopilot mode, which simplifies node provisioning and integrates well with Google Cloud's billing model[27]. This configuration was chosen to minimize operational cost and to maintain full Kubernetes capabilities.

Our current production cluster is configured with three nodes (Abstractions of physical machines), under the constraints of GCP's free tier. This setup provides sufficient resources for prototyping and real-world testing while minimizing costs. Autoscaling is enabled within these limits allowing the cluster to scale workloads up or down depending on resource demands.

The GKE cluster hosts several core services such as:

- MongoDB is a self-hosted instance running with Persistent Volume Claims (PVCs) for durable storage.

- ActiveMQ is a message broker service that acts as the communication bridge between firmware devices and backend.
- Report Server (Express API) is the main backend responsible for processing sensor data, serving analytics and providing API endpoints to the frontend.

The services requiring external access are exposed via LoadBalancer-type services. The API server is publicly reachable for communication with the frontend and mobile clients. The ActiveMQ is exposed with a dedicated LoadBalancer to restrict access so that only firmware clients can connect to it and push messages.

All Kubernetes resources are defined and managed using Kustomize, which allows environment-specific overlays and promotes reusable configuration files. This prevents duplication while maintaining clean separation between deployment layers.

These are the Resource Definitions that required manual configuration.

- Deployments: Resources that contain the executable code of an application [29].
- Services: Resources that provide physical IPs and handle connectivity for Deployments either for internal or external usage. The LoadBalancer type Service provides extra-cluster connectivity and a ClusterIP type Service provides intra-cluster connectivity for Deployments.
- Horizontal Pod Autoscaler (HPAs): Resources that are responsible for the dynamic scaling of Deployments. According to the configuration of the HPA, the targeted deployment will scale automatically [30].
- Persistent Volume Claims (PVCs): Resources that represent persistent volume space, used by Deployments that require data persistence (e.g., The MongoDB service) [32].
- ConfigMap: An object for non-confidential data storage in key-value pairs [28].

The deployment process is automated via GitHub Actions, where a custom deployment script applies Kubernetes manifests post-build. While the detailed CI/CD pipeline is described in a later section, the integration ensures that each new image pushed to Artifact Registry is automatically deployed to the cluster.

Application configuration and secrets are managed through a combination of Kubernetes ConfigMaps, used to import environment variables from .env files and Kubernetes Secrets, used for sensitive values (e.g., database URIs, service credentials, JWT secrets).

While alternatives such as Cloud Run or Compute Engine were considered, they were not chosen, Cloud Run, although cost-efficient, is limited to stateless services and would not support our MongoDB and ActiveMQ workloads. Compute Engine would require complex manual provisioning and would not provide the Kubernetes environment necessary for our preferred deployment workflows.

5.1.5 Static Files and Google Buckets

The frontend of the Auto Harvest platform, which is built using React Native for Web, is deployed as static files to Google Cloud Storage (GCS). GCS integrates seamlessly to our CI/CD pipeline and Cloudflare and provides fast, scalable static content delivery globally.

We maintain two separate GCS buckets for static content, auto-harvest-ui-prod and auto-harvest-ui-stage. This allows isolated deployment and testing workflows so that changes can be staged before promotion to production. The Frontend files (HTML,CSS,JS,assets) are built and uploaded via a custom deployment script, which handles syncing the output of the frontend build (npx nx <project>:build) to the appropriate one. This script can be triggered either automatically through GitHub Actions during CI/CD pipelines, or manually (locally) by developers for rapid debugging or patch testing.

Although the buckets are publicly accessible, their direct URLs are obfuscated and never exposed to users. Instead, access is routed through a Cloudflare-managed subdomain, which applies URL rewrites to support React Router SPA navigation and serves files via Cloudflare's edge caching.

Due to the nature of single-page applications, cache consistency is very important. After each deployment, we trigger a full cache invalidation using compute.urlMaps.invalidateCache, ensuring updated JavaScript and CSS files are delivered in their version. Since the bucket contains only these critical assets, full invalidation is performant and predictable.

Regarding security and permissions, the bucket grants all users (allUsers) the minimal permission of retrieving objects (storage.objects.get), allowing anyone to fetch static files directly if they know the URL. However, listing, editing, or deleting contents is fully restricted from public user access. Only collaborators (via Google accounts and impersonation) and the "GitHub Actions Runner" service account have mutation privileges (write,delete,update).

5.1.6 Traffic Engineering

The traffic engineering approach for Auto Harvest is focused on simplicity, security, reliability and scalable routing for HTTP(S) and MQTT traffic, the system uses a hybrid model of Google Cloud Load Balancers and Cloudflare Edge Routing to manage ingress[37], optimize performance and enhance security.

We use domain-based routing to separate services and support future extensibility. The base domain "autoharvest.solutions" serves the static frontend files of the Web build with SPA support. The primary API entrypoint for the React native client and dashboard interface is accessible via the "api.autoharvest.solutions" url. Finally a url of this style "mqtt-<hash>.autoharvest.solutions is the dedicated entrypoint for the firmware to connect with MQTT over TCP. This domain and subdomains are managed via Cloudflare DNS.

Incoming traffic first hits Cloudflare, which acts as a DNS resolver, Reverse Proxy[39], TLS terminator[33] (with Full SSL support) and then passes through the Firewall and WAF rules layer[34]. Requests are then routed to Google Cloud Load Balancers, which forward traffic to the GKE cluster via LoadBalancer type Kubernetes service. This Hybrid approach was preferred because Cloudflare's free tier offers functionalities in a much more centralized way and personally we believe the dashboard UI of Cloudflare is very intuitive and offers the same, if not more functionalities (especially considering its a free tier) than their respective GCP counterparts.

Strict SSL mode is enforced by Cloudflare, which requires a valid certificate on the origin (GCP). GCP certificates are automatically supplied and managed by the HTTP(S) Load Balancer[36], making sure connections are end-to-end encrypted between Client→Cloudflare→GCP. With Strict SSL mode, GCP will reject all direct connections (bypass attempts) that are not coming from the "autoharvest.solutions" managed by Cloudflare.

While current deployments expose one service per endpoint, future routing may introduce URL map routing through GCP's backend service (e.g., api.autoharvest.solutions/payments, api.autoharvest.solutions/oauth). At present, each service is individually routed and exposed via its own LoadBalancer and subdomain. Stage and production environments are also isolated via different buckets and CI workflows, but share similar traffic routing logic to ensure isomorphism between the environments.

Several abuse mitigation tools are leveraged in order to prevent malicious actors from attacking, exploiting and ransoming our infrastructure. Strict CORS headers to prevent hotlinking, data exfiltration, and unauthorized cross site requests. Automatic rate limiting set by cloudflare mitigates abuse from repeated request floods (e.g., DoS attack). Automated attacks trying to leverage known exploits are handled by Cloudflare's bot detection security challenges, that block suspicious request patterns (e.g., scans for .env, wp-config.php, php.ini) and force the client to solve challenges that only "real browsers" are able to solve. Finally the "I'm Under Attack" mode[35] 38], which as the name suggests, locks down our Cloudflare managed domains, and forces everyone to solve a CAPTCHA to discern real users from bots. All similar settings from GCP have been deactivated and delegated fully to Cloudflare, to further reduce compute costs and load on the backend services.

One noteworthy limitation was encountered while attaching multiple backends (paths) to a single Load Balancer (essentially a proxy/reverse proxy configuration with Nginx). GCP's Ingress controller often failed to provision a static IP address properly under the free tier, which occasionally blocked deployment. This problem was resolved by switching to standalone Load Balancer deployments per service.

5.1.7 Load Balancer and Google CDN

The project's infrastructure integrates a Google Cloud HTTPS Load Balancer to manage ingress traffic to both API services and static frontend assets. This layer acts as the critical bridge between Cloudflare's global edge network and the internal workloads running in the GKE cluster served via Cloud Storage buckets.

Due to provisioning issues with the Kubernetes Ingress resources, we opted to manually configure an external HTTPS Load Balancer through the GCP console. This approach allowed us to reliably provision static IP addresses, a feature that often failed under the automatic Ingress workflow. Each backend service (e.g., API or Cloud Storage bucket) is attached to the load balancer via backend service bindings, exposing the services securely to the public.[40]

Google's internal Cloud CDN is enabled on the load balancer's backend services for static file hosting buckets and cache-friendly API endpoints. This approach reduces networking (traffic volume) and computational costs and further reduces the latency of the system for repeated cacheable API endpoint and static file requests. Buckets used for frontend deployments are attached to the load balancer using backend buckets (via the Bucket Connector tool of Cloudflare), allowing Google's CDN layer to serve static content. Caching behavior is controlled at both the Cloudflare and GCP levels, and we have implemented a custom invalidation strategy for manual and automatic invalidations.

- A full cache invalidation is triggered through GCP's UI when necessary.
- On-demand bypass is available via a special request header named "no-cache-baby". This header is recognized by both Cloudflare and GCP and it instructs edge nodes to bypass cache for rapid frontend testing or debugging (e.g., apps in the staging environment should always apply this header to requests). An entry with a toggle button has been created in Cloudflare's header rewrite tools[42], allowing developers to toggle it on and bootstrap all incoming requests with this header (useful when applying a hotfix)

Average latency of static assets and API endpoints was observed to be under 100ms from different physical locations using Cloudflare's Trace tool[41], which is considered acceptable for our use case. As such, no further optimizations in Caching and Networking were deemed necessary.

5.2 Cloudflare

5.2.1 Cloudflare Services

Cloudflare plays a central role in the Auto Harvest infrastructure by acting as a security perimeter, performance accelerator and edge traffic router. It is positioned as the first point of contact for all external requests, sitting in front of both the API services and frontend hosting buckets. The integration of Cloudflare enables global traffic distribution via their CDN[43], reduces the overall attack surface (with WAF, bot management and more), enables smart routing, features a header rewrite engine, and simplifies SSL and TLS management. It also offers performance monitoring and optimization, tracing, traffic analysis and fast reverse proxying making it one of the most useful services in our stack.

The following Cloudflare features are actively used in our deployment:

- DNS Management[46]: Cloudflare manages all DNS records for autoharvest.solutions.
- Proxy & CDN Layer: All A and CNAME records for public services are proxied through Cloudflare, which masks GCP IPs and leverages Cloudflare's global CDN to cache and serve content faster, especially for static files
- Full SSL with Origin Verification: TLS termination occurs at Cloudflare using Full SSL (Strict) mode, where Cloudflare validates the origin server using a custom certificate installed in GCP. This ensures encrypted communication end-to-end and mitigates man-in-the-middle or spoofing attacks
- WAF (Web Application Firewall)[44]: Rules are defined to filter traffic based on patterns (e.g., known malicious user agents, suspicious paths like wp-config.php, .env, etc.).
- Reverse Proxy and Rewrites: Custom routing logic and header rewrites support single-page application (SPA) behaviour, enabling the functionalities of React Router.
- Rate limiting and Threat mitigation: Cloudflare mitigates abusive traffic through challenge modes and request caps. The "I'm Under Attack" mode can be toggled to enable site-wide CAPTCHA protection during suspected DDoS activity.
- Caching and performance optimization: Caching controls are used to reduce bucket load.

5.2.2 DNS Record management

All domain resolution and DNS control for the ecosystem are managed through Cloudflare's DNS platform. Cloudflare's dashboard for DNS management provides intuitive record setup tools that are user-friendly, while also offering complex configuration options. Similar competitor tools were evaluated by us, namely Papaki's, and GoDaddy's and while they offer similar functionalities, Papaki's user interface is not user friendly and both Papaki and GoDaddy offer limited configuration options compared to Cloudflare for DNS management (This conclusion is supported by the official documentation of each provider, where Cloudflare offers more numerous, detailed and exhaustive feature documentation [46] compared to Papaki [47] and GoDaddy [48].).

Our root domain, autoharvest.solutions, serves as the primary namespace for all application resources. We use a combination of DNS record types to support different services.

- A Records are used to point subdomains like api.autoharvest.solutions and mqtt-<hash>.autoharvest.solutions to static IP addresses associated with Google Cloud Load Balancers
- CNAME Records are utilized for redirecting www.autoharvest.solutions to autoharvest.solutions and for bucket-based hosting references (e.g., storage.googleapis.com aliases for frontend delivery)

- MX Records are used to handle email routing for admin or notification use cases via external mail services
- TXT Records are utilized for SPF/DKIM/DMARC email verification [50]. They are also used for domain ownership validation and workarounds to bypass Google Cloud's bucket URL access restrictions and ensure SEO/indexing compliance.

Most functional records are proxied through Cloudflare allowing us to mask backend infrastructure (GCP IP addresses), apply Cloudflare WAF and firewall filtering and gain performance benefits via edge caching. DNS-only records are used only where required, such as for mail delivery or third-party validation services.

At this stage of the project, DNS records are managed manually through the Cloudflare Dashboard. This approach is sufficient for the current deployment scale and allows for quick updates during development. As the system matures, automation via Cloudflare's API[51] or Terraform[52] may be adopted for consistency and CI/CD integration.

5.2.3 WAF Rules

To protect the system's infrastructure from common web threats, automated scanners and abusive behavior, we utilize Cloudflare's Web Application Firewall (WAF). Cloudflare's WAF offers an advanced and easy-to-manage interface that lets us apply global protection rules at the DNS edge, before requests even reach our Google Cloud infrastructure.

We rely on Cloudflare's prebuilt managed rulesets, which are continuously updated to cover the most common and dangerous exploit patterns, including prototype injection attempts (Node is susceptible to such attacks), Cross-site scripting (XSS) payloads, exploits targeting CMS platforms (included by default) and malformed user-agents as well as uncommon HTTP methods. These rulesets are enabled globally with enforcement for all proxied subdomains. Cloudflare by default also explicitly blocks access to known exploit probes and penetration test paths, including paths like: /.env, /admin, /phpmyadmin, /php.ini, /.htaccess and in general any request path or query string that resembles automated vulnerability scanning attempts. This helps prevent automated enumeration and vulnerability scanning tools from probing our services.[53]

Cloudflare's Bot Management engine is enabled to distinguish between trusted crawlers and malicious or unknown bots. This setup allows search engine bots like Googlebot to index our public frontend (which is important for SEO), Block or JS-challenge unknown or suspicious bots and to give us visibility into bot traffic trends and behavior.[54]

To prevent abuse and brute-force style attacks, a rate limit is enforced via WAF rules that detects if a single IP requests the same resource more than 20 times per minute in order to temporarily throttle or challenge the connection. This is very useful for high-risk endpoints such as "/api/login" and "/api/register". WAF event logs and request analytics are monitored in frequent intervals through the CLoudflare dashboard, enabling us to rapidly detect anomalies, blocked requests and emerging threats as well as error responses. [55]

5.2.4 Reverse Proxy Provider

Cloudflare functions as the reverse proxy for all public-facing services in the Auto Harvest infrastructure. This role allows Cloudflare to act as an intermediary between clients and backend servers, handling DNS resolution, SSL termination, caching and security inspection before forwarding requests to Google Cloud services.

All application traffic follows this general route as explained previously.

Client→Cloudflare (Edge Node)→Google Load Balancer→GKE Cluster / Bucket

With this design we can ensure backend IP addresses are always hidden, TLS traffic is terminated and inspected at the Cloudflare edge and that requests are filtered, cached or rewritten before hitting our origin infrastructure.

Cloudflare applies a set of routing rules and header rewrites to support subdomain-based routing (e.g., api., mqtt-*. etc.), Single-page application (SPA) compatibility, by rewriting unknown paths to index.html for the frontend and development shortcuts, such as bypassing cache using custom headers like the “no-cache-baby” custom header. Without rewrites, React’s Router wouldn’t be able to behave correctly and a static asset bucket as the frontend’s host wouldn’t be a viable option.

By using Cloudflare as a reverse proxy, all SSL/TLS connections are fully encrypted end-to-end using Strict SSL mode, requests must carry Cloudflare’s trusted certificate to reach GCP, preventing direct origin access and firewall rules as well as bot detection are enforced before any resource consumption on Google’s side, drastically cutting costs as Cloudflare is used with a free tier plan. Cloudflare’s reverse proxy also enables fast toggling of features like “I’m Under Attack” mode, offers real-time analytics on response codes, bot traffic and cache performance and offers edge-side customization without needing to redeploy code or touch backend infrastructure (for example changing the SSL certificate), giving us total control in a centralized dashboard regarding how all external traffic is routed, protected and optimized before reaching the core system.

5.2.5 SEO and Performance Optimization

Beyond security and routing, Cloudflare plays an important role in ensuring that Auto Harvest is fast, accessible and discoverable on the web. Performance tuning and adequate SEO are the key in improving user experience and ensuring proper indexing and ranking by search engines.

As discussed above, static files served from Google Cloud Storage buckets are cached at both Cloudflare’s global edge CDN and Google CLoud’s regional CDN layer. This caching approach reduces load times, and improves core web vitals[58] for SEO ranking. Furthermore, to ensure proper indexing and discoverability, Googlebot and other trusted crawlers are explicitly allowed via Cloudflare Bot Management page, a robots.txt file is served from the bucket, specifying crawlable paths, proper canonical tags are injected in HTML templates to prevent duplicate indexing and Meta information (title, description, OpenGraph) is pre-configured for each major route in the frontend. Cloudflare caching does not interfere with SEO crawlers since it respects the HTTP Vary headers[56] and since we have zero reliance on dynamic Server-Side Rendering (SSR) which could delay and block indexing.

We monitor SEO and frontend performance via the Google Search Console[60] (for index coverage and crawling errors), Cloudflare analytics[59] (for traffic and latency problems) and perform Lighthouse audits[57] (in order to optimize asset sizes, reduce unused files, and boost performance scores)

5.3 Billing and Budgeting

Cloud infrastructure costs can scale quickly without proper monitoring and controls set in place. In this project, careful budgeting and cost visibility were essential due to the limited nature of academic funding. Google Cloud Platform (GCP) provided a flexible billing system and resource configuration that allowed us to develop and deploy this project. The project was initially deployed using the $300 GCP free credits, which provided enough resources for:

- Creating and maintaining a regional Autopilot Kubernetes cluster
- Hosting static files in Cloud Storage

- Using Load Balancers for API and broker exposure
- Using Cloud Monitoring and Artifact Repository

The Autopilot mode was chosen for Kubernetes because it allowed us to only pay for used resources, avoiding costs associated with idle node pools. To prevent overspending, we configured budget alerts in the Google Cloud Console[61] and monitored the cost breakdown tab weekly to identify and reduce unnecessary resource consumption.

Important architectural decisions were made with cost in mind, such as separating buckets for stage and production, which enabled us to reduce duplication and testing overhead, also using Cloudflare's free tier for DNS, proxying and security to offload bandwidth and minimize egress costs. Avoiding GCP's DBaaS (Database as a service, such as Cloud SQL or managed Atlas) in favor of a self-hosted MongoDB instance further cut down costs as well as the minimal use of premium services unless justified by performance or functionality (Like buying a dedicated IPV4 to use as the egress's address. To further reduce cloud resource usage during development, staging deployments were often run on a local Kubernetes cluster and only stable builds were promoted to the production GKE cluster, minimizing costs due to less pod terminations and initializations.

Another significant cost was the usage of a web-tunneling tool called Ngrok, used to map local ports to live subdomains (e.g., http://localhost:3000→https://autoharvest.ngrok.app), its usage will be explained at a later section, but its cost was $10 monthly for the basic tier[64]. A paid basic CoPilot license was also utilized in order to improve the developers' experience (DevEx) that cost $20 monthly[65].

After a month of having the Auto Harvest system online, we noticed that the Autopilot cluster consumes $0.93 - $0.98 (averaged at $0.95) daily[62]. Other costs such as traffic, registry image storage, bucket storage and IP provisioning cost less than $0.04 daily[63] (the system was mostly idle and under no stress, so this would explain the low daily costs). With those costs in mind, we assume that the production operating expenses of the system when it is idle is around $30 monthly, and that the development operating costs are $30 monthly. These expenses summed up to $60 as the total monthly operating costs of the system. Another non-significant cost was the acquisition of the domain name, which was sold to us for $2 annually (with a 90% discount for the first 4 years).

The most significant capital expenditure was the cost of the actual research and development (R&D). The total cost of the materials used was near $50, the hardware including the sensors, the actuators and the arduino board along with cables and the power supply surpassed $200, with the summed cost for R&D being $250 total. The product is not considered to be production ready, but merely a minimum viable product (MVP), so more R&D costs are expected in the future if we want a final working prototype ready.

Section 6: Developer Experience

Since the Auto Harvest project was a collaborative effort that involved hardware, backend, and mobile UI, we had to establish clear practices to keep development manageable. In section 6 we will describe the tools, workflows that we followed as developers, starting with version control and continuing with repository organization, local development and finally collaboration.

6.1 Version Control

To ensure a structured and efficient development lifecycle, we adopted a trunk based Git workflow with two long lived branches: dev and main. Trunk based development is a widely recommended approach in modern software engineering because it avoids long lived feature branches and reduces merge conflicts [66], [67].

New features and bug fixes were first developed in short lived branches, named using the conventions feature/issue_name or hotfix/issue_name. Once implemented, these branches were merged into the dev branch, which acted as the main integration space. When dev reached a relatively stable point, it went through validation and testing in a live Kubernetes cluster. This approach may have not been the best practice, we should have used another long lived branch named staging which would live between dev and main branches for us to do the validating and testing. Using the Kubernetes cluster allowed us to detect configuration errors and integration problems in a realistic environment before final deployment. Once the stage deployment was validated, a pull request was opened to merge the changes into main, finalizing the release pipeline [68].

Table 6.1: Example of branch usage

Branch	Purpose	Actions
dev	Integration branch where new features and fixes are merged and tested in development environments	Feature branches (feature/*) and hotfixes (hotfix/*) are merged here after review. Acts as the staging area before production.
main	Production branch that reflects the most stable and release ready version of the project	Pull requests from dev are merged here after validation, triggering deployments and release builds.

Code reviews were also part of our workflow for every pull request, especially when making changes from dev to main. This collaboration caught some potential mistakes early, and gave us an opening to talk about things that we may need and don't need in our project. Although no strict commit message format was used, we kept commit messages short and descriptive ex. *"Added an unavailability screen to use when a functionality is not working on a web app."* In some cases where branches with too many small commits were used, squashed and rebased merges were used to keep the main branch history clean.

For the mobile frontend, we used Expo Application Services (EAS) to generate internal distribution builds when native libraries were introduced and to produce production ready APKs [69].

Although unit and end-to-end testing were constructed with NX, these were not actively used during this phase of development. Similarly, tasks were tracked with GitHub Issues, but commits and pull requests were not consistently linked to them. Even with these limitations, the workflow kept development organized, reduced the risk of merge conflicts, and helped us manage a multi-repository project more effectively.

6.2 Repository Organization

Auto Harvest is split into two main codebases to keep things clean and manageable. Having a clear separation between the frontend and the backend allowed us to work on each part without stepping on each other's toes, and also made testing and deployment much easier. One repository is dedicated to the mobile/web app (frontend), while the other is a monorepo that contains both the backend services and the firmware for the hardware.

Frontend Repository
The mobile app is built with react Native using the Expo Framework[71] and the code is written in TypeScript. To avoid clutter, the project is structured into folders for components, screens, redux slices, utility functions, and assets. For navigation we used expo-router [72], which automatically creates routes based on the folder structure, so the navigation logic stays simple and does not require extra boilerplate.

For state management, the app uses Redux Toolkit [73] together with redux-persist, which lets us keep important data locally even if the app is restarted. Several native libraries are also included, like react-native-wifi-reborn for handling Wi-Fi connections, and Expo modules like expo-location and expo-haptics to access android device features. To keep the code consistent and reliable, we used ESLint for linting. This setup gave us a codebase that was both scalable and approachable, so adding new screens or features did not get messy over time.

Backend and Firmware Monorepo
The second repo is organized as an NX Monorepo [70], this gave us the option to manage multiple projects under one workspace. Inside the apps/ directory of the repo, we have all the backend services, testing utilities, and firmware projects.

Here are all them:

- api: The main backend service built with Express.js, connected to MongoDB by Mongoose, and integrated with ActiveMQ message queue, to handle all the traffic that passes from the backend. This service is responsible for receiving data from IoT devices, storing it, edit it and expose it through REST endpoints.
- api-e2e: End-to-end testing project that was created by NX's testing tools. It was used very little by us, but gave us a way to run automated test to check our API endpoints.
- front/auto-harvest: A lightweight app used internally for development and debugging. We tested backend features in a browser environment by it.
- iot/io-manager: A PlatformIO project [74] used for firmware development, mainly for the ATMEGA2560 microcontroller. All the configuration and calibrations for our sensors happened in this "app". With this, we could upload code, monitor serial output, and manage libraries directly from VS Code

Having all of these inside the same workspace made development smoother. Even though the backend services and firmware don't share code directly, the close proximity made it easier to test the

interaction between microcontrollers sending sensor data and the backend processing it. I also reduced context switching, we could code for multiple services without juggling multiple environments.

Table 6.2: Repository Comparison

Repository	Main Technologies	Purpose
Auto-harvest-ui	React Native, Expo, Typescript, Redux Toolkit, expo-router	Mobile/Web interface for monitoring and controlling the system
Auto-harvest	NX, Express.js, MongoDB, ActiveMQ, PlatformIO	Backend services and firmware for microcontrollers

This two-repo setup gave us the flexibility to move between mobile, backend, and firmware development without much hassle. It also made the system easier to maintain and expand since each part had a clear “home”.

6.3 Local Development

When we started designing the project, we aimed for something that could easily run and test all parts of the frontend, backend and firmware locally without a complex setup. As we builded the local environment, we could see that it helped us a lot because we could test ideas on the spot and see if we could use them before even deploying them.

Frontend
The mobile app was launched locally with a single command, npx expo start, which started the Expo development server [75]. Most of the time we used the Expo Go app on our phones to test changes in real time, since it supports hot reloading and makes debugging almost instant. We had a problem with some capabilities that Expo had and we needed to use native libraries in our application, when they were added, we switched to internal distribution builds, which made the testing and debugging little difficult and everything had to be close to production ready to install.

For debugging, we used tools like Redux DevTools, which let us track how the application state was changing in response to actions. Expo itself provided real time logs and quick reloads, so the workflow stayed lightweight and easy to manage without needing complicated scripts.

Backend
The backend lived in the NX monorepo and was usually started with npx nx serve api [76] or using the NX vscode plugin. We mainly worked on it inside Visual Studio Code, which provided good integration with NX Plugins. The backend was built with Express.js, but it also depended on external services like MongoDB and ActiveMQ. Instead of installing these directly, we ran them locally inside Docker containers [77]. This made it simple to set up the same environment across different machines.

Sensitive configuration files were never committed to HitHub. We shared them privately, so that everyone's local setup stayed up to date without exposing secrets.

Firmware
Firmware development was handled in PlatformIO [78], where we wrote small C++ programs for the

ATMEGA2560 microcontroller and the configuration for all sensors that needed calibration. The code was built and uploaded directly to the hardware. For debugging, we relied on two “monitors”.

- The serial monitor, which gave us live logs and could see that everything returned data and also it was the correct data after calibrations.
- The LCD Display, which was used as the monitor of reading values without having to open the repo on the laptop. It was mostly our way to check that everything that was configured was showing live in the monitor and also changed whenever an alteration happened.

Besides all that, the ActiveMQ dashboard was life saving, it helped us confirm whether messages were actually published and pushed to the correct endpoint.

All that combined meant that all the parts of the project, the two repos, could be developed and tested independently, but also connected we had a full end-to-end testing. Developing the project was very manageable with these methods, in spite that there were many different technologies used for it.

6.4 Collaboration and Development Tools

Throughout our development of the project, we used a mix of collaboration, coding and deployment tools to keep the project moving, but also to get more used to many of those tools. Since the project involved multiple parts, having the right tools made organization easier.

- GitHub was our main platform for version control and collaboration [1]. Both repos, **auto-harvest-ui** and **auto-harvest** were hosted there, and we used PRs and reviews to keep the workflow structured.
- GitHub Projects gave us Kanban-style boards, which made tracking issues and tasks more transparent. This helped us prioritize work and gave a clear view of progress, even when we weren’t together to split the work.
- Docker was used for containerizing the project [2]. Running MongoDB and ActiveMQ in containers meant everyone on the team could spin up the same environment with minimal setup. It also guaranteed that what worked locally would behave the same in testing and deployment.
- Google Cloud Platform (GCP) hosted our backend and databases [3]. By using GCP, we got a reliable, scalable and secure environment where the mobile app could pull live sensor data without interruptions.
- Expo development tools were essential on the frontend [4]. With expo start and Expo GO, we could quickly test features on real devices, and when we needed more realistic builds, we used EAS internal distributions.
- Visual Studio Code was our main IDE[5]. It handled both TypeScript for the frontend and C++ for the firmware through PlatformIO. Most of the coding, debugging and testing happened in the same environment.

Together these tools made collaboration much easier. They supported agile development, continuous integration and delivery, and they helped us maintain a consistent workflow across all parts of our project.

Table 6.3: Collaboration and Development Tools

Tool	Purpose
GitHub	Version control, PRS, code reviews
GitHub Projects	Kanban-style boards for task tracking and issues connected to the repo
Docker	Containerized backend dependencies
Google Cloud Platform	Hosted backend, database and scalable infrastructure
Expo Tools	Local Testing, fast reloads, build production ready apps
Visual Studio Code	Main IDE for all the coding lifespan

6.5 Continuous Integration and Deployment

Continuous Integration and Continuous Deployment (CI/CD) pipelines are integral during the rapid development process that occurs at the beginning of a project, and at later stages where continuous maintenance and feature addition is performed. Proper implementation of CI/CD pipelines ensures that every new version of the consisting applications of a project are built and deployed in the exact same way, limiting repetitive and error prone steps required by the developer to deploy a new version of the application.

6.5.1 GitHub Actions

The Auto Harvest backend uses GitHub Actions as its continuous integration provider. The CI pipeline is responsible for verifying code integrity, building Docker images and triggering deployments via Google Cloud Platform (GCP). A single workflow (at ci.yml) handles backend services. It is triggered on push events to the main branch, which is protected by pull request requirements and code reviews. The workflow performs the following steps:

- Authenticates to GCP using a service account secret
- Prepares the environment with GKE credentials, kubectl, Docker CLI and node
- Attempts to retrieve previous node_modules cache with a hash key from the runner
- If cache didn't exist, it runs npm i to build the node_modules directory
- It performs linting to ensure code standards are enforced and TypeScript is valid
- Uses Nx to detected affected applications (changed compared to last commit, HEAD compared to BASE, only changed applications are built, reduces runner costs) and conditionally triggers Docker builds and Kubernetes pushes
- If cache didn't exist, it saves the new node_modules version
- Performs post actions cleanup to successfully wrap up the actions run

At this stage, the workflow is production-only, but in the future it will be extended to include a staging branch in the near future (as of now, the dev branch acts as the staging branch, but this is not considered the best practice)

The workflow build for the Frontend is currently limited only to building the Web client, the steps are the same as the previous flow, with 2 differences: It does not authenticate to GKE, kubectl or Docker CLI, it only logins to Google Cloud Storage and using the same service account. Then it builds and uploads static files to GCS. invalidates the previous cache on GCS by performing a full invalidation.

Github Secrets are used to store sensitive credentials, including a GCP service account key (GCP_SECRET), these secrets are mounted into the workflow environment using google-github-actions/auth, additional runtime secrets (e.g., broker URIs) are mounted in Kubernetes via encrypted ConfigMaps.

With this CI/CD configuration, the project automatically and correctly, builds and deploys every time code with valid syntax and build output is committed to the main branch (always via a Pull Request). Collaborators are never worried about messing up with the actual production server, the process is automated and abstracted, allowing us to save time from performing all the steps mentioned manually.

6.5.2 Deployment Pipeline Flow

The Auto Harvest project follows a reliable, scalable and fast (from PR to code deployment in less than 5 minutes) streamlined deployment process. While the CI/CD workflow (Section 6.5) handles build automation and validation, the deployment strategy defines how new code moves safely from development to production and how environments are managed over time.

The repository follows a trunk-based development model supported by three branches:

- dev branch: Used for development testing. Docker images are built with a dev tag and validated within a local Kubernetes cluster.
- stage branch: Reserved for future use with a dedicated staging namespace and cluster. Currently inactive.
- main branch: The production branch. All merges to main must pass a pull request review and trigger a full deployment pipeline to Google Kubernetes Engine (GKE)

This structure enforces code quality gates [90] such as reviews, CI checks, unit tests, integration tests, health checks [92], before any production change is accepted, ensuring the production stability of the system.

Merges to main initiate an automated release that builds and pushes Docker images for affected backend services and updates Kubernetes Deployments with the new image tag. This way, production is always updated in a consistent and reproducible manner. Frontend releases are currently managed through a similar approach, the only change is the helper script that invokes GCS instead of GKE commands.

At present the project operates in a single production environment. However the kubernetes cluster is configured to support namespaced deployments [89], providing a path to scale into a shared staging name space with the existing cluster or a separate staging cluster for strict isolation once required.

In the event of a faulty release, rollbacks [91] are performed manually using the “kubectl set image deployment/<service-name> <container-name>=<previous-image-digest>” command. This mechanism uses immutable image digests stored in Artifact Registry, enabling precise reversions of individual services. Future enhancements will include an admin GUI tool for easier selection of the version to be restored so the process would not be error prone.

Deployment health is currently monitored through Github Actions logs (per PR), Google Cloud Console dashboards for GKE and Artifact Registry and using Aptakube, a GUI for the kubernetes CLI.

6.5.3 Helper Scripts

To support and extend the CI/CD pipeline, the project includes a set of custom shell and PowerShell scripts that enable environment-specific deployments, local development configuration and manual debugging or rollback workflows. These scripts are located in the "ci/scripts/" directory of the project repository and are used both within CI workflows and standalone by developers.

The "build.sh" script automates the Docker build and deployment process for backend services. It is invoked automatically by the CI pipeline through the Nx task runner (docker-push) but it can also be run manually for debugging or rapid iteration. Steps include:

- Retrieving the latest commit SHA (HEAD)
- Building the service with yarn nx run <service>:docker-build
- Tagging the image for Artifact Registry (:latest and :<sha-digest>)
- Pushing to GCP's private registry
- Applying the new image to the Kubernetes deployment via kubectl set image

Image history can be referenced from GitHub commits, allowing manual rollbacks using SHA digests when needed.

The "actions-front.sh" script handles the deployment of the web frontend to a Google Cloud Storage Bucket following these steps:

- Runs a targeted build using "yarn nx run <project>:build-<env>"
- Deletes the existing project folder in the bucket
- Uploads the new static assets
- Invalidates the Cloud CDN cache using "gcloud compute url-maps invalidate-cdn-cache"

The script detects the deployment mode (dev/stage/prod) based on the current Git branch and is triggered either manually or through Nx (bucket-push) as part of CI.

For local Kubernetes development, a PowerShell script is included to add a custom DNS entry (127.0.0.1 to local.net) to the Windows hosts file allowing the local cluster's load balancer to be resolved by name. This script requires administrator privileges to run, automatically backs up the hosts file and avoids duplicate entries by checking for existing matching rules. It is intended for developer use and is stored in the repository alongside other dev tools to support a shared and predictable local environment

These helper scripts ensure that developers can work independently, CI can remain clean and minimal and infrastructure can adapt to both cloud and local workflows over time, these scripts may be wrapped in CLi tools or replaced with Terraform/Helm for broader scale automation.

Section 7: Results

7.1 Test Run

To ensure that Auto Harvest meets both functional and practical expectations [93], [94], [95], [96], [97], a series of test runs was conducted, each building on the results of the previous. The first test run focused on the electronics and software, verifying that sensors, actuators, and network communication operated reliably and as intended.

With the system proven technically sound, the second test run evaluated its biological performance, assessing whether the hydroponic setup, including net pots, pumps, tubing and nutrient delivery, could sustain healthy plant growth over multiple weeks. This provided tangible evidence that the system could translate its electronics controls into real-world outcomes.

Finally, the third test run will focus on data validation at scale, aggregating millions of sensor readings to examine trends, distributions and correlations. Through time-series charts, heatmaps and statistical analyses, this test will confirm that the collected data is consistent, reliable and suitable for monitoring and decision-making in a real hydroponic environment.

Together, these test runs provide an extensive evaluation of the system, from hardware and firmware functionality, to biological efficacy and large-scale data integrity.

7.2 End-to-End Operational Test

Before we could consider Auto Harvest complete, we needed to validate that the system worked end-to-end, from sensor readings, to data transfer, to physical actuation. To keep things structured, we defined clear test objectives and followed a checklist approach.

Objectives
The main questions we wanted to answer were simple but essential:

- Can the firmware connect to Wi-Fi, initialize all modules and send sensor readings?
- Can we issue a command from the mobile app or MQTT broker and see a physical response, like a pup turning on/off?
- Are the readings visible in real time, either on the LCD, in the backend logs or through MQTT explorer?
- Does the fault logic behave as expected if a sensor reports an error?

This gave us a practical definition of “working” for the prototype.

Procedure

We began by checking all wiring and connections. Power supplies were secured, relays were tested, and modules were confirmed to initialize correctly on boot. A short dry run using Serial.print() logs verified that each sensor and relay reported a successful initialize() status.

Next, we performed network checks. The device was put in Wi-Fi client mode and connected to our MQTT broker. With MQTT Explorer, we subscribed to topics such as sensor-log and pump-on to confirm that logs were being published every five seconds.

With the network stable, we moved on to issuing commands. From the app and from MQTT Explorer, we sent commands like pump-on, and lcd-on. The relays toggled as expected, the LCD updated, and we even noted the audible click of the relay and the pump starting as additional feedback.

To validate sensor logic, we simulated real-world conditions. Water level sensors were dipped in and out of water, the flow sensor was blocked to simulate a pump fault, and we observed how the firmware

responded. In each case, the device correctly logged warnings, displayed messages on the LCD, and relayed information to the backend. Importantly, the fault protection logic worked: the pump was shut down automatically when no flow was detected.

Finally, we tested resilience. Disconnecting Wi-Fi forced the system into fallback mode, where local logs and alerts were still maintained. Power cycling the device showed that it persisted its configuration, and unplugging a sensor confirmed that the firmware detected and reported a missing device gracefully.

Results

Overall, the test run confirmed that the prototype behaved as designed. Sensor data was collected and transmitted successfully, commands triggered correct physical responses, and error handling logic prevented damage during fault conditions. There were minor issues, such as occasional Wi-Fi reconnection delays and noisy sensor values, but none of them stopped the system from working.

Analysis

The test run gave us confidence that Auto Harvest can reliably manage a small hydroponic setup. At the same time, it highlighted areas for future work, such as improving Wi-Fi reconnection routines, refining calibration, and expanding logging to persistent storage. For a student-built prototype, the results were encouraging: we demonstrated real-time monitoring, control, and basic resilience, the core features of a hydroponic IoT platform.

7.3 Plant Growth Capability Test

Before considering the hydroponic system suitable for sustained operation, we needed to validate that it could support real plant growth under normal usage conditions [98], [99]. While electronic and firmware tests ensured that monitoring and control worked, this test focused on the biological performance of the system, specifically, whether the physical components (net pots, tubing, aeration, water circulation, and nutrient delivery) provided an environment capable of producing healthy plants. To keep the evaluation structured, we defined clear objectives and documented the system's performance over a five-week period.

Objectives
The main questions we sought to answer were:

- Are the handmade net pots and grow medium sufficient to physically support seedlings throughout early growth stages?
- Does the nutrient solution circulate properly through the tubing network without clogging or uneven distribution over the course of a month?
- Can the water pump and air pump maintain continuous flow and oxygenation?
- Does the fertigation mixture deliver adequate nutrients for sustained growth?
- Under stable environmental conditions, can the system grow lettuce from seedling (~4cm) to harvest-ready size (~26cm)

These objectives served as our practical definition of "biologically functional"

Procedure
We began by transplanting uniform lettuce seedlings (approximately 4 cm tall) into the system's net pots after we washed the soil off the roots under running water carefully. Each net pot was filled with expanded clay balls selected for their low water retention, and placed in the designated grow slots to ensure consistent exposure to water and oxygen. Prior to adding plants, the pump and aeration system

were tested for 24 hours to confirm that circulation, flow rate, and dissolved oxygen levels were stable, and that the tap water used had all its chlorine and fluorine added for sanitation, evaporated.

A standard hydroponic nutrient solution suitable for leafy greens (8-6-7 NPK) was prepared and monitored weekly. The water pump delivered the solutions through the main tube network, while the air pump maintained constant aeration at the reservoir. We visually inspected for leaks, flow obstruction and any signs of nutrient mineralization.

During the five-week test period, we recorded plant height and captured weekly photographs to document their progressions. In addition to plant observations, we evaluated component-level behavior, net pots were checked for root constriction or structural instability, tubing was inspected for buildup or reduced flow, water pump performance was monitored for flow consistency and noise changes (turbulent flow indication), air pump output was verified by observing the root health and growth, nutrient levels and pH were measured and adjusted based on sensor readings.

Results

Over the five-week period, the lettuce grew from approximately 4cm to 30cm, showing a steadily accelerating weekly increase in height and foliage density. Roots expanded well beyond the net pots, forming healthy white root structures indicative of sufficient oxygenation and nutrient availability. No clogging occurred in the tubing and the pump maintained stable circulation through the test.

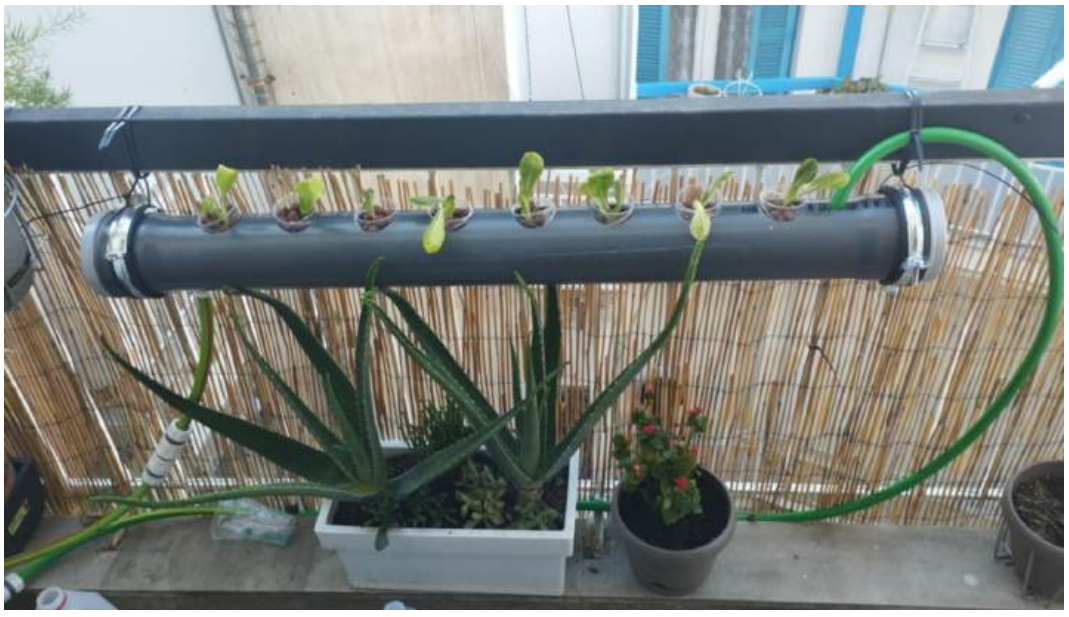

Image 7.1: Week 1 of planting lettuce onto the system

Image 7.2: Week 2 of planting lettuce onto the system

Image 7.3: Week 3 of planting lettuce onto the system

Image 7.4: Week 4 of planting lettuce onto the system

Image 7.5: Week 5 of planting lettuce onto the system

Image 7.6: A closeup of the roots inside the pots

The fertigation mixture proved suitable, as evidenced by uniform leaf color and the absence of nutrient-deficiency symptoms. Both the water pump and air pump operated continuously without failure and the net pots supported the plants without tilting, cracking or restricting root growth.

Minor issues included nutrient film and foam buildup near the pump outlet and inlet during Week 4, that was cleaned after the next water change with hydrogen peroxide. PH values were fluctuating and needed manual correction by adding citric acid to the solution, over corrections where corrected using baking soda, these interventions had minimal impact in the final TDS measurement. The system required 2 water changes over the course of the 5 week experiment.

Analysis

The test run demonstrated that the system is capable of sustaining plant growth over multiple weeks. All major physical subsystems, net pots, tubing, circulation and aeration performed reliably and created an environment suitable for leafy greens. The growth results confirmed that the design can support both nutrient delivery and root development without manual intervention beyond routine maintenance. The lettuces may appear tall and thin because in this time of year, this position in the balcony doesn't receive direct sunlight, this is the reason the lettuces grew tall but not dense.

The system met the core criteria for a functional hydroponic growth unit: maintaining consistent flow, delivering nutrients effectively and supporting healthy and measurable plant growth.

7.4 Big Data Validation Test

The final test run presents the results of a comprehensive data validation test conducted to verify the accuracy, reliability and consistency of the IoT sensor network implemented in the Auto Harvest hydroponic monitoring system. The validation test serves as an important prerequisite to ensure that the system's data collection infrastructure and logic is robust and reliable before deployment in the production environment.

The data validation test was designed to assess the operational performance of six key sensor types integrated in to the Auto Harvest IoT platform: Total Dissolved Solids (TDS) sensor, flow rate sensor (liters-per-minute), water temperature sensors, humidity sensors, air temperature sensors and calculated Vapor Pressure Deficit (VPD) values. The test ran continuously for 13 days (November 9-22, 2025) generating approximately 2.45 million sensor readings sampled at 2-second intervals.

Objectives

In this test run, the main goals were to:

- Check the system's reliability by gathering continuous data capture over an extended period without crashes or data loss.
- Validate teh data quality of the readings to be within physically plausible ranges, with consistent sampling intervals.
- Verify the cross-sensor validity and see if sensor reading correlations match known physical relationships and trends (e.g. if Humidity Temperature and VPD values make sense for the current time of day).
- Perform time-series analysis to reveal behaviors trends and cycles.

Procedure

The data validation test commenced on November 9,2025, at 03:19:44 UTC and concluded on November 22, 2025, at 11:06:34 UTC, spanning a total duration of 13 days. During this period, the system operated under normal hydroponic conditions to capture realistic operation variability and environmental fluctuations. During this period, the system operated under normal hydroponic conditions to capture realistic operation variability and environmental fluctuations.

The raw, unsanitized sensor data values were saved in a local mongodb instance in local context with the server, after the test run was concluded, the whole collection was exported and input to a data

analysis python script, that prompted us to select the analysis' time range and which sensor values to keep (float sensor values and the default value of the now broken pH sensor were omitted), the python script outputted time series charts (figure 7.7-7.11) of water temperature, air temperature, humidity, tds and liters per minute (average water flow) as well as calculated VPD values.

It is important to note that the main controller module, housing the DHT11 air temperature and humidity sensor, was placed inside a residential environment during this test. Consequently, the data for air temperature and humidity does not reflect the typical diurnal (day/night) cycles expected in a greenhouse; however, these cycles remained clearly visible in the water temperature data.

Results

The test successfully generated a massive dataset of approximately 2.45 million sensor readings, providing high-resolution insight into the system's behavior over nearly two weeks. The database successfully handled continuous write operation without exhibiting any problem.

However, upon reviewing the timeline, we noted that the system suffered from re-occuring downtimes. After investigating the error logs, we concluded that this was caused by the firmware serializing sensor values with excessive precision digits (e.g. sending 23.123456 instead of 23.12). This overflowed the microcontroller's buffer, sending it into a restart loop. This bug was subsequently fixed by strictly limiting sensor readings to a maximum of 3 precision digits.

Analysis

Despite the gaps caused by the buffer overflow, the millions of captured data points allowed us to perform meaningful analysis of the system's sensing capabilities. The impact of the sensor placement was clearly visible in the time-series charts. The air temperature and humidity graphs (Figures 7.8 and 7.9) were lacking the typical diurnal oscillation because of heat radiators operating during cold hours, disrupting both temperature and relative humidity readings (that is dependent on temperature as well). However, the diurnal cycle is clearly observed in the water temperature chart (Figure 7.7) that displays clear cyclic behavior, altho due to the thermal capacity of the water, the diurnal cycle observed in the water temperature over time appears more flat.

The TDS and flow rate data (Figures 7.10 and 7.11) demonstrated high stability during operational periods. The flow rate readings remained constant, confirming that the pump logic was not triggering false fault states. The TDS values showed a realistic, gradual trend consistent with nutrient concentration, validating the sensor's utility for long-term monitoring.

In the sensor value correlation Figure (7.12) we can clearly see that the TDS value is highly correlated with temperature, this is observed because the TDS sensor is compensated by the water temperatures sensors reading for compensation, resulting in the final TDS value being a function of both electrical conductivity of the probes, and temperature. The liters per minute reading is not correlated with any other sensor value, as it should be expected since the reading is independent from any other environmental value. Finally, figure 7.14 displays extracted statistical values of each sensor, all statistical values are in their expected ranges and from this we can conclude that the system collects sensible readings.

Overall, while the test uncovered a critical firmware bug regarding memory management, it validated that the core sensing and logging architecture is capable of producing high-fidelity data suitable for precision agriculture.

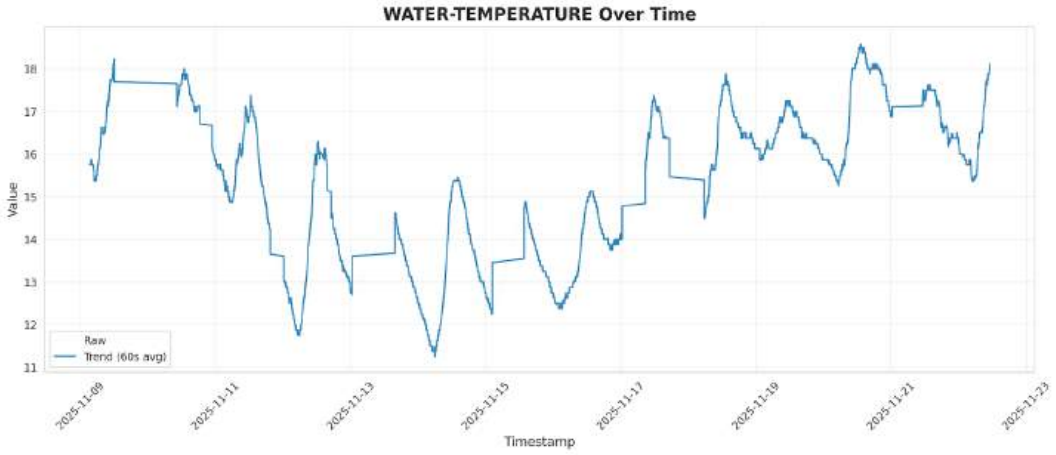

Figure 7.7: Water Temperature Over Time (13-day validation period)

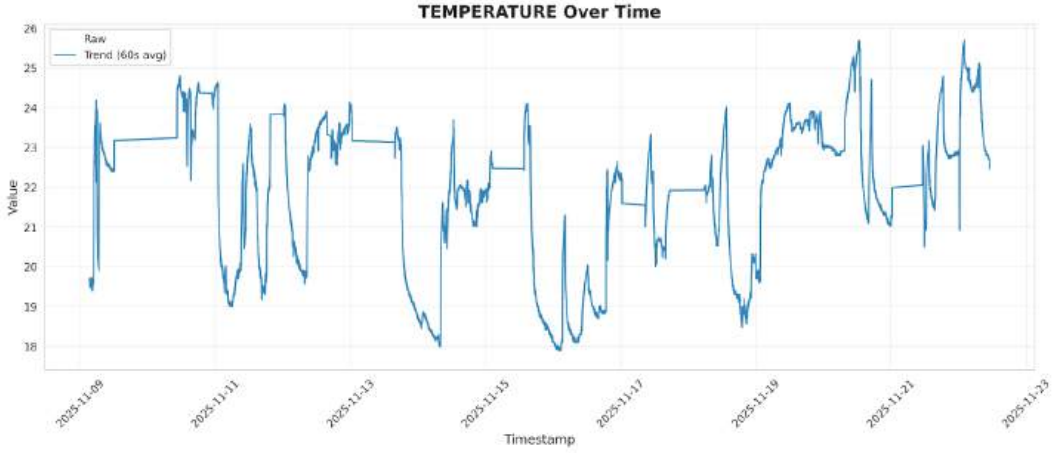

Figure 7.8: Air Temperature Over Time

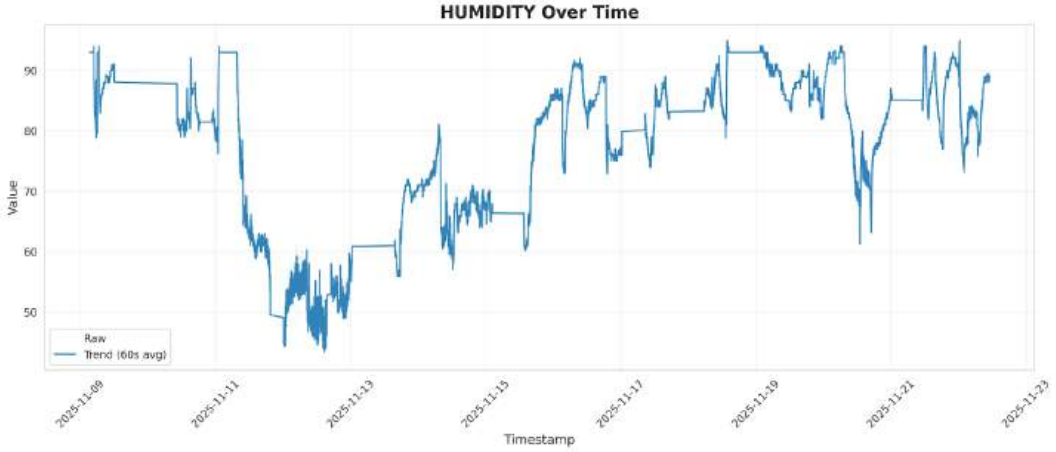

Figure 7.9: Humidity Over Time

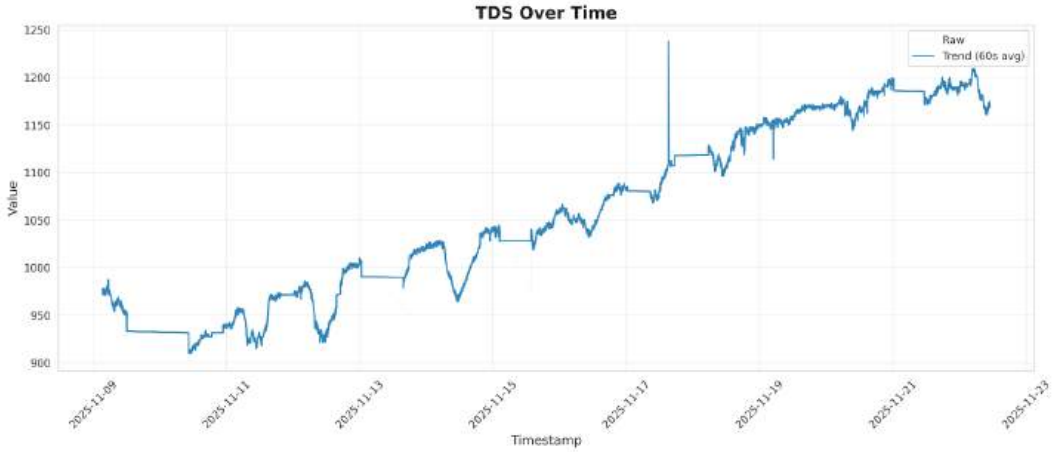

Figure 7.10: TDS Over Time

Figure 7.11: VPD Over Time

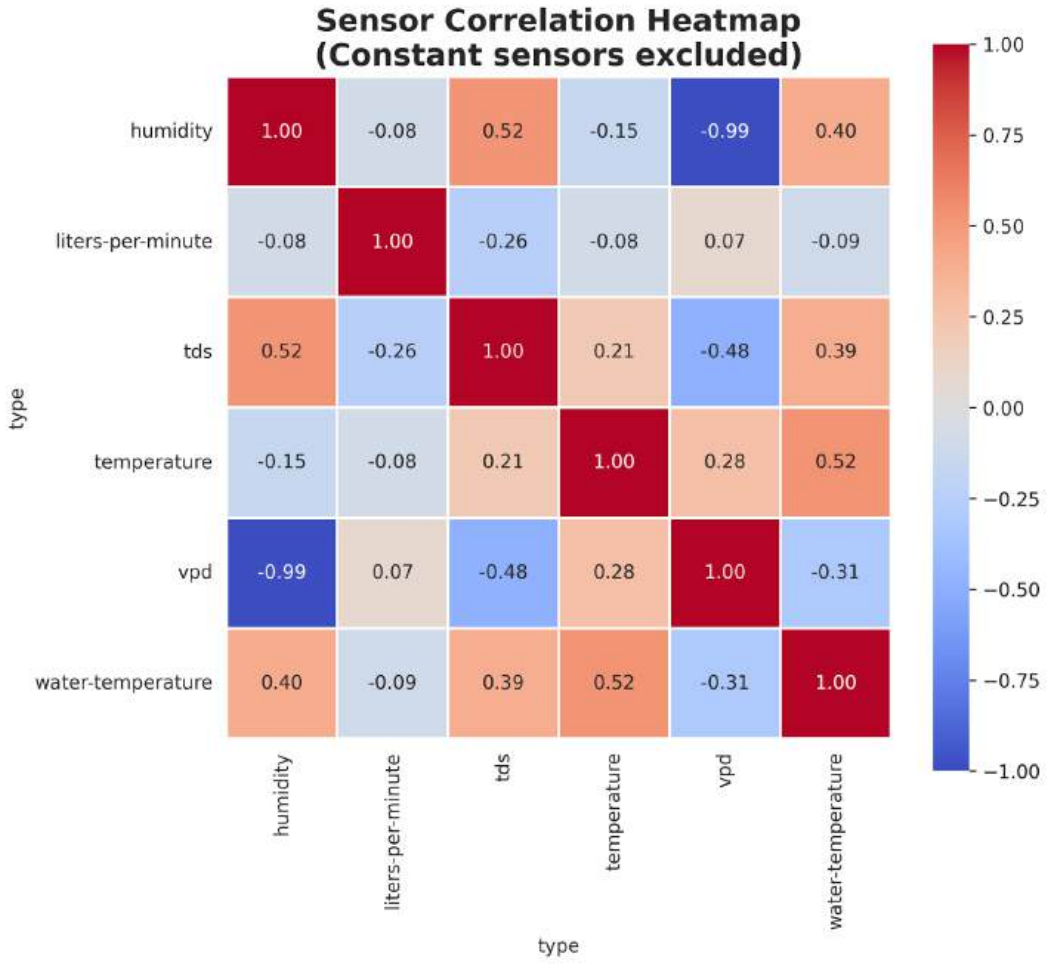

Figure 7.12: Sensor Correlation Heatmap

```
================================================================================
1. DATA QUALITY METRICS
================================================================================
Total Records: 2,448,154
Date Range: 2025-11-09T03:19:44.859000+00:00 to 2025-11-22T11:06:34.374000+00:00
Duration: 13 days
Number of Controllers: 1

Records by Sensor Type:
  tds: 408,020 records (16.7%)
    Average interval: 2.0 seconds
  liters-per-minute: 408,026 records (16.7%)
    Average interval: 2.0 seconds
  water-temperature: 408,027 records (16.7%)
    Average interval: 2.0 seconds
  humidity: 408,027 records (16.7%)
    Average interval: 2.0 seconds
  temperature: 408,027 records (16.7%)
    Average interval: 2.0 seconds
  vpd: 408,027 records (16.7%)
    Average interval: 2.0 seconds
```

Figure 7.13: Data Quality metrics

Table 7.1: Extracted statistical values of each sensor

Metric	Mean	Median	Std Dev	Range (Min–Max)	Spread	IQR (Q1–Q3)
TDS	1067.50	1056.35	90.40	907.97 – 1238.48	330.51	985.66 – 1162.51
Liters/Minute	2.21	2.22	0.25	1.80 – 5.56	3.75	2.12 – 2.30
Water Temp	15.37	15.75	1.78	11.25 – 18.62	7.38	13.88 – 16.75
Humidity	78.31	82.00	12.82	43.00 – 95.00	52.00	69.00 – 89.00
Temperature	21.69	22.00	1.97	17.80 – 25.70	7.90	19.90 – 23.20
VPD	0.42	0.31	0.35	0.00 – 1.51	1.51	0.14 – 0.63

7.5 Delivered Product

The delivered prototype of Auto Harvest is a fully functional, end to end IoT system that can monitor and manage a hydroponic environment using a combination of sensors, embedded firmware, cloud infrastructure, and a mobile/web application. What makes this meaningful to us is that the entire system was designed, built and deployed by us, from breadboard wiring and low level firmware to cloud deployments and real time dashboards.

Table 7.2: Core Capabilities

Layer	Capabilities
Firmware	Polls sensors, control relays, drives the LCD screen, manages WiFi pairing, communicates over MQTT
Hardware	Includes water temperature, TDS, pH, humidity, water level, flow sensors
Connectivity	Real time telemetry and command handling over MQTT
Safety Logic	Flow based pump shutdown, anomaly detection and local alerts
Storage	MongoDB logging of environmental data and controller states
Frontend	Mobile/web app for pairing devices, live status and basic management
Deployment	Fully containerized CI/CD pipeline deployed on GKE with Cloudflare proxy routing

Deliverables Achieved

- Modular firmware running on ESP8266, designed to be ESP32-ready
- Integration of six different sensors with stable outputs
- MQTT based bidirectional communication between sensors and cloud actuators
- Periodic data publishing and command responses with custom topic structure
- Backend server built with Express.js, MongoDB, and WebSocket API
- Device pairing through the mobile app and an embedded AP mode web interface
- Local LCD screen that rotates through telemetry and alert messages
- Local error handling logic for resilience when offline (e.g., pump shutoff logic)
- Secure authentication, encrypted data transport and automated deployments via CI/CD

What's Still Manual or Incomplete?

Even though the system works well as a prototype, there are a few areas that remain incomplete or manual:

- Firmware updates require USB flashing
- Web push notifications are not supported
- No combined multi sensor graphing UI in the app
- Hardware casing and PCB remain at the design stage
- No production level cloud monitoring tools (e.g., Prometheus)

Overall, this prototype delivers on its original promise: to be a scalable foundation for smart hydroponics. While not yet production ready, it demonstrates the core technical pieces, shows

resilience under testing, and provides a clear roadmap for expansion. For us, this is both a functional engineering achievement and a solid starting platform for future work.

7.6 Alternative Applications

One of the strengths of the Auto Harvest system is its modular design in both hardware and software. Even though it was built with hydroponics in mind, the same architecture can be reused in many other IoT domains with only minor changes to the sensors or the firmware. This makes Auto Harvest a flexible platform for remote sensing, logging, and automation, especially where real-time telemetry and local decision-making are important.

IoT Weather Station
With a different sensor suite, the system could operate as a weather monitoring station. The existing temperature and humidity sensors already provide some data, and adding modules such as a barometric pressure sensor (BMP280), a wind speed/direction sensor, and a rain gauge would extend it into a full-featured weather station. Data could be logged over time, displayed on the LCD or app, and even shared with public weather networks through MQTT or HTTP.

Soil Monitoring & Smart Irrigation
Auto Harvest can also be adapted for traditional soil-based farming. By swapping the water-level sensors with soil moisture probes and using relays to drive irrigation valves, the device could automate watering schedules. Adding a solar panel and battery pack would make it suitable for deployment in remote fields. This setup could send alerts for dry soil, monitor nutrient runoff, and scale up into large installations using mesh networking.

Smart Aquarium and Aquaponics Control
Because aquaponics shares many sensors with hydroponics (pH, TDS, temperature), Auto Harvest can also serve as a smart aquarium controller. The relays could manage pumps, heaters, or oxygenation systems, while additional sensors such as CO_2 or light detectors could balance water and plant health. In the future, adding cameras with object detection could even help track fish behavior or tank conditions.

Educational IoT Toolkit
Given its modularity and relatively low cost, Auto Harvest could work well as an educational kit for schools or universities. Students could use it to learn embedded programming with Arduino, practice calibration and signal processing, and study how data flows from sensors to the cloud and back. A simplified version of the firmware could make it beginner-friendly, turning the system into a hands-on STEM teaching tool.

Remote Lab Instrumentation
Auto Harvest could also be used in research environments as a remote data logger. By connecting different analog or digital sensors, the device could log pH values for chemistry experiments, monitor soil conditions in environmental studies, or track climate factors in ecological fieldwork. With its existing cloud backend and mobile UI, it already supports logging, alerting, and visualization, making it a ready-made edge data acquisition unit.

General Reusability
Several design choices make the system broadly reusable:

- The MQTT topic structure is abstracted and can carry any type of message
- Relay and LCD logic can be reused for almost any actuator/indicator
- The pairing flow and configuration storage work across different deployment scenarios
- Both web and mobile UIs need only small changes to adapt to new domains

Section 8: Conclusions and Future Work

8.1 Summary and Achievements

The Auto Harvest project successfully demonstrates how embedded systems, cloud infrastructure and mobile applications can converge to create a complete IoT solution for hydroponic agriculture. The system monitors critical environmental parameters (pH, TDS, temperature, humidity and water level) in real time, automates nutrient and water delivery though relay-controlled pumps and provides remote monitoring and control through a mobile application with push notifications.
The delivered prototype proved functional across multiple test cycles, successfully maintaining target pH and TDS ranges while providing reliable telemetry to users. By combining ATMega2560 with the ESP-8266 microcontrollers, utilizing MongoDB for storage, Node.js for our backend, and React Native for the mobile and web application, we created a working end-to-end hydroponics monitoring system.

Some of the projects key achievements are:

- Real-time monitoring and control: Six-sensor telemetry with WebSOcket streaming and automated actuator control.
- Full-stack IoT architecture: Embedded firmware (C++), REST/WebSocket APIs (Node.js), cloud database (MongoDB), Edge data collection (ActiveMQ) and cross-platform mobile app (React Native).
- Remote accessibility: Users can monitor conditions and control the actuator from anywhere with internet connectivity.
- Modular and extensible design: The architecture supports additional sensors and actuators with minimal code changes.
- Self-funded prototype: Delivered a functional system using free-tier cloud services, cheap or re-purposed hardware.

Coming from a software engineering background, this project pushed us significantly outside our comfort zone. We gained hands-on experience with embedded firmware development, sensor calibration theory, circuit assembly and the unique challenges of deploying hardware in realistic agricultural environments, skills that rarely surface in pure software projects.

8.2 Limitations and Challenges

Despite reaching a functional state, the prototype has several limitations that constrain its readiness for production deployment.

The most significant limitation is the absence of over-the-air (OTA) firmware updates. Currently, updates require a physical USB connection to the microcontroller, making remote maintenance impractical. The ATMega2560 + ESP-8266 combination has non-existent support for OTA updates. Migration to an ESP32 platform is necessary to enable secure wireless updates.

Sensor calibration remains a manual and technical process. Both pH and TDS sensors require two-point calibration with reference solutions, but the current workflow involves hardcoding the calibration offset to the firmware instead of dynamically providing it upon initialization, or writing it to the EEPROM, via a user-friendly calibration flow.
Debugging capabilities were constrained throughout development. WIthout hardware debuggers, breakpoints or persistent error logos, diagnosing crashes and sensor faults was difficult. The firmware

restarted automatically after failures, obscuring root causes and extending troubleshooting cycles. The ATMega2560 doesn't support those debugging features.

Furthermore the entire project was self-funded which heavily influenced design decisions. We relied on free-tier cloud services (300$ 90-days GCP credit, self hosted database and activeMQ as well as self hosting the server after the 90-days free credit expired), secondhand hardware and a minimal budget-friendly sensor set. These constraints limited cloud capacity, restricted hardware choices and slowed development velocity. Our initial limited knowledge of electrical components, calibration theory and embedded C++ meant that early firmware iterations were more ad-hoc than planned. The learning curve was steep, and while we improved greatly over time, it had a measurable impact on development speed and code quality.

8.3 Future Work

Based on lessons learned during development and testing, we have identified a clear roadmap for improving the system’s reliability, usability and scalability

These are some of the highest priority changes in our backlog. Migration to ESP32 hardware will enable secure Wi-Fi based firmware updates, eliminating the need for physical USB connection and enabling remote bug fixes and feature deployments. App guided calibration flow within the mobile app will walk users step-by-step through the process of placing probes in calibration solutions and automatically recording reference values. Implementing data backup strategies like daily/weekly snapshots will ensure that no catastrophic data loss ever occurs.

We have also noted some enhancements of lesser priority, like: Multi-user environment and Role-Based permissions, we want to introduce user roles such as administrator, contributor, read-only to allow teams to share dashboards and manage devices collaboratively. Enable web push notifications by utilizing Firebase Cloud Messaging, ensuring that web users receive real-time alerts. While isolated graphs are functional now in both web and android, in the future we want to add a consolidated graph viewer that will allow users to identify correlations (e.g. TDS changes relative to temperature fluctuations) and make data-driven decisions. And last but not least integrate Google Cloud Monitoring with support for tracing, logging and metrics collection to enable high quality system health monitoring.

8.4 Production Readiness

Transitioning from a development prototype to a production-ready system requires several engineering refinements:

- Custom PCB design, we need to replace breadboard wiring with a PCB integrating the ESP32, dedicated sensor sockets, proper connectors, voltage regulation and fuse isolation.
- Water resistant enclosure, we need to design an IP65-rated enclosure with ventilation for environmental sensors, cable strain reliefs, and easy maintenance access.
- Standardized cabling, use color-coded, labeled cables with proper connectors to simplify installation and reduce disconnection risks.
- Power Resilience, we should investigate alternative powering solutions such as battery-backed or solar-powered options [100] -[111].
- Safety and Fault Tolerance, we should implement hardware/software watchdog timers, flow protection logic for pumps, maintenance mode switches and status LEDs for immediate visual feedback.
- Automated provisioning, controller units will be flashed and provisioned with unique identifiers, secret keys and QR codes ensuring repeatable and error-free onboarding

8.5 Closing Remarks

Auto Harvest originated as an effort to apply software engineering principles to an unfamiliar application domain: hydroponic agriculture. Over the course of development, the project expanded to encompass embedded firmware, sensor calibration, circuit design, cloud infrastructure, and mobile application development. The resulting prototype is functional, modular, and demonstrates the practical feasibility of an end-to-end IoT system for precision agriculture.

While the system has clear limitations, particularly in firmware update mechanisms, calibration usability, and production readiness, it provides a validated architectural foundation. The project confirms that with constrained budgets, limited hardware resources, and undergraduate-level experience, it is still possible to design and implement a meaningful IoT solution that integrates software, hardware, and physical processes.

The work also highlights the importance of interdisciplinary engineering. Addressing real-world agricultural problems required knowledge beyond traditional software development, including electrical design considerations, sensor behavior, and environmental variability. These challenges significantly influenced system design decisions and informed the proposed roadmap for future improvements.

Moving forward, Auto Harvest can evolve from a single-node prototype into a scalable platform suitable for larger greenhouse or controlled-environment deployments. Potential extensions include multi-node sensor networks, long-term data analytics, and more advanced nutrient modeling. With further refinement, the system could serve both educational and practical roles in smart agriculture.

In conclusion, Auto Harvest demonstrates that accessible, open, and repairable precision agriculture tools are achievable without industrial-scale resources. As such, it contributes a small but practical example to the broader field of smart and sustainable agriculture, while also serving as a foundation for continued technical development and research.

Section 9: Abbreviations

ADC	Analog-to-Digital Converter
AC	Alternating Current
API	Application Programming Interface
APK	Android Package (Kit)
AP	Access Point
AWS	Amazon Web Services
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Deployment (or Delivery)
CLI	Command Line Interface
CMS	Content Management System
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
DB	DataBase
DBaaS	DataBase as a Service
DC	Direct Current
DDoS	Distributed Denial of Service
DNS	Domain Name System
DevEx	Developer Experience
DoS	Denial of Service
EAS	Expo Application Services
EC	ElectroConductivity
EEPROM	Electrically Erasable Programmable Read-Only Memory
GCC	Google Cloud Console
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
HPA	Horizontal Pod Autoscaler
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTP(S)	HyperText Transfer Protocol (Secure)
I2C	Inter-Integrated Circuit
IAM	Identity and Access Management
IP	Internet Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
LCD	Liquid Crystal Display
MCU	MicroController Unit
MQTT	Message Queuing Telemetry Transport
MVP	Minimum Viable Product
NFT	Nutrient Film Technique
OTA	Over-The-Air (update)
PCB	Printed Circuit Board
PVC	PolyVinyl Chloride
PVC	Persistent Volume Claim
R&D	Research and Development
SCL	Serial Clock Line

SDA	Serial Data Line
SEO	Search Engine Optimization
SPA	Single Page Application
SQL	Structured Query Language
SSID	Service Set Identifier
SSL	Secure Sockets Layer
TDS	Total Dissolved Solids
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
UX	User Experience
VPA	Vertical Pod Autoscaler
WAF	Web Application Firewall
XSS	Cross-Site Scripting
IPv4	Internet Protocol version 4
ppm	Parts Per Million

Section 10: References

[1] Expo Documentation (https://docs.expo.dev/)
[2] Application of soilless culture technologies in the modern greenhouse industry – A review (https://www.researchgate.net/publication/328888013_Application_of_soilless_culture_technologies_in_the_modern_greenhouse_industry_-_A_review)
[3] Zigbee | Complete IOT Solution - CSA-IOT, (https://csa-iot.org/all-solutions/zigbee/)
[4] Understanding Zigbee and Wireless Mesh Networking - Black Hills Information Security, Inc. (https://www.blackhillsinfosec.com/understanding-zigbee-and-wireless-mesh-networking/)
[5] ELECTROCHEMICAL METHODS (https://eva.fcien.udelar.edu.uy/pluginfile.php/144317/mod_resource/content/3/Electrochemical%20Methods%20Bard%20Faulkner%20Cap1-4.pdf)
[6] Calibration and adjustment of a pH electrode, (https://www.xylemanalytics.com/en/company/blog/blog/2023/05/calibration-and-adjustment-of-a-ph-electrode)
[7] Analog TDS Sensor Meter for Arduino / ESP32 / Raspberry Pi - DFRobot Wiki , (https://wiki.dfrobot.com/Gravity__Analog_TDS_Sensor___Meter_For_Arduino_SKU__SEN0244)
[8] how TDS relates to EC, (https://web.archive.org/web/20141031171622/http://www.epa.gov/esd/cmb/pdf/JAG-TDSpublished.pdf)
[9] DHT11 Humidity & Temperature Sensor (https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf)
[10] DS18B20 - Programmable Resolution 1-Wire Digital Thermometer DS18B20 - Programmable Resolution 1-Wire Digital Thermometer (https://www.analog.com/media/en/technical-documentation/data-sheets/DS18B20.pdf)
[11] Buy P43 – Liquid and Water level float switch sensor module (https://electronicspices.com/product/p43-liquid-and-water-level-float-switch-sensor-module Buy P43 – Liquid and Water level float switch sensor module)
[12] Αισθητήρας Ροής Υγρών - Πλαστικός 1/2" NPS με Σπείρωμα (https://grobotronics.com/liquid-flow-meter-plastic-1-2-nps-threaded.html)
[13] Understanding the Pull-up/Pull-down Resistors With Arduino : 6 Steps - Instructables (https://www.instructables.com/Understanding-the-Pull-up-Resistor-With-Arduino/)
[14] PH meter SKU SEN0161 PH meter SKU SEN0161 https://wiki.dfrobot.com/ph_meter_sku__sen0161_)
[15] Analog TDS Sensor Meter for Arduino / ESP32 / Raspberry Pi - DFRobot Wiki https://wiki.dfrobot.com/Gravity__Analog_TDS_Sensor___Meter_For_Arduino_SKU__SEN0244)
[16] Mini Brushless Water Pump 12V DC 240L/h https://grobotronics.com/mini-brushless-water-pump-12v-dc-240l-h-ad20p-1230a.html)
[17] Mini Air Pump 12V DC (https://grobotronics.com/mini-air-pump-12v-dc.html)
[18] Relay Module - 4 Channel 5V (https://grobotronics.com/relay-module-4-channel.html)
[19] ESP8266 WiFi Module (https://grobotronics.com/esp8266-wifi-module.html)
[20] ESP8266 Technical Reference (https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf)
[21] ESP-01 Adapter Module 3.3-5V (https://grobotronics.com/esp-01-adapter-module-3-3-5v.html)
[24] Correlation between conductivity and total dissolved solid in various type of water: A review (https://iopscience.iop.org/article/10.1088/1755-1315/118/1/012019/pdf)

[25] Using Hydrogen Peroxide to Control Algae in Hydroponic Systems - ASHS (https://ashs.org/news/607318/Using-Hydrogen-Peroxide-to-Control-Algae-in-Hydroponic-Systems.htm)
[27] GKE Autopilot overview | Google Kubernetes Engine (GKE) https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview)
[28] ConfigMaps | Kubernetes (https://kubernetes.io/docs/concepts/configuration/configmap/)
[29] Deployments | Kubernetes (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
[30] Autoscaling Workloads | Kubernetes (https://kubernetes.io/docs/concepts/workloads/autoscaling/)
[31] Service | Kubernetes (https://kubernetes.io/docs/concepts/services-networking/service/)
[32] Persistent Volumes | Kubernetes (https://kubernetes.io/docs/concepts/storage/persistent-volumes/)
[33] Overview · Cloudflare SSL/TLS docs (https://developers.cloudflare.com/ssl/)
[34] Overview · Cloudflare Web Application Firewall (WAF) docs (https://developers.cloudflare.com/waf/)
[35] Get started · Cloudflare DDoS Protection docs (https://developers.cloudflare.com/ddos-protection/get-started/)
[36] Cloud Load Balancing (https://cloud.google.com/load-balancing)
[37] GKE Ingress for Application Load Balancers | GKE networking | Google Cloud (https://cloud.google.com/kubernetes-engine/docs/concepts/ingress)
[38] Under Attack mode · Cloudflare Fundamentals docs (https://developers.cloudflare.com/fundamentals/reference/under-attack-mode/)
[39] What is a reverse proxy? | Proxy servers explained | Cloudflare (https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/)
[40] Backend services overview | Load Balancing | Google Cloud (https://cloud.google.com/load-balancing/docs/backend-service#service_bindings)
[41] How to - Cloudflare Trace (beta) · Cloudflare Rules docs (https://developers.cloudflare.com/rules/trace-request/how-to/)
[42] URL Rewrite Rules (https://developers.cloudflare.com/rules/transform/url-rewrite/)
[43] Cloudflare Cache (CDN) docs (https://developers.cloudflare.com/cache/)
[44] Overview · Cloudflare Web Application Firewall (WAF) docs (https://developers.cloudflare.com/waf/)
[46] Cloudflare DNS docs (https://developers.cloudflare.com/dns/)
[47] Free DNS Service | Papaki (https://www.papaki.com/en/free-dns.htm)
[48] How to manage DNS Records | GoDaddy (https://www.godaddy.com/en/help/manage-dns-records-680)
[50] What are DMARC, DKIM, and SPF? | Cloudflare (https://www.cloudflare.com/learning/email-security/dmarc-dkim-spf/)
[51] Cloudflare API | overview (https://developers.cloudflare.com/api/)
[52] Terraform Documentation (https://developer.hashicorp.com/terraform/docs)
[53] SpiderFoot automates OSINT for threat intelligence and mapping your attack surface. (https://github.com/smicallef/spiderfoot)
[54] Cloudflare Bot Management & Protection (https://www.cloudflare.com/application-services/products/bot-management/)
[55] Rate limiting rules · Cloudflare Web Application Firewall (WAF) docs (https://developers.cloudflare.com/waf/rate-limiting-rules/)
[56] IETF RFC 2616 (https://www.rfc-editor.org/rfc/rfc2616.txt#vary)

[57] Introduction to Lighthouse | Chrome for Developers (https://developer.chrome.com/docs/lighthouse/overview/)
[58] Understanding Core Web Vitals and Google search results (https://developers.google.com/search/docs/appearance/core-web-vitals)
[59] Cloudflare Analytics docs (https://developers.cloudflare.com/analytics/)
[60] Google Search Console (https://search.google.com/search-console/about)
[61] Create, edit, or delete budgets and budget alerts | Cloud Billing | Google Cloud (https://cloud.google.com/billing/docs/how-to/budgets)
[62] Google Kubernetes Engine pricing (https://cloud.google.com/kubernetes-engine/pricing?hl=en)
[63] Virtual Private Cloud pricing (https://cloud.google.com/vpc/pricing)
[64] ngrok pricing | Flexible plans for production and development (https://ngrok.com/pricing)
[65] GitHub Copilot · Your AI pair programmer (https://github.com/features/copilot/plans)
[66] Atlassian. Trunk-Based Development vs Git Flow. Atlassian Git Tutorials. (https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development)
[67] Chacon, S., & Straub, B. Pro Git (2nd Edition). Apress. (2014) (https://git-scm.com/book)
[68] GitHub Doc. About pull requests. (https://docs.github.com/en/pull-requests)
[69] Expo Docs. EAS Build Overview. (https://docs.expo.dev/build/introduction)
[70] NX Dev. NX: Extensible Monorepo Tool (https://nx.dev)
[71] React Native Docs. React Native - Learn once, write anywhere. (https://reactnative.dev)
[72] Expo Docs. Expo Router. https://docs.expo.dev/router/introduction/)
[73] Redux Toolkit Docs. Redux Toolkit Quick Start (https://redux-toolkit.js.org/introduction/getting-started)
[74] PlatformIO Docs. PlatformIO IDE for VSCode (https://docs.platformio.org)
[75] Expo Docs. Development with Expo Go. (https://docs.expo.dev/get-started/expo-go/)
[76] NX Dev. NX CLI - Serve Command (https://nx.dev/packages/nx/documents/serve)
[77] Docker Docs. Get Started with Docker. (https://docs.docker.com/get-started/)
[78] PlatformIO Docs. PlatformIO IDE for VSCode. (https://docs.platformio.org)
[79] What is CI/CD? (https://www.redhat.com/en/topics/devops/what-is-ci-cd)
[80] GitHub Actions documentation (https://docs.github.com/en/actions)
[81] Nx set SHAs · Actions · GitHub Marketplace (https://github.com/marketplace/actions/nx-set-shas#background)
[82] Secrets - GitHub Docs (https://docs.github.com/en/actions/concepts/security/secrets)
[83] GitHub - google-github-actions/setup-gcloud: A GitHub Action for installing and configuring the gcloud CLI. (https://github.com/google-github-actions/setup-gcloud)
[84] Cache dependencies and build outputs in GitHub Actions (https://github.com/actions/cache)
[85] nx/eslint:lint (https://nx.dev/technologies/eslint/api/executors/lint)
[86] Command line tool (kubectl) | Kubernetes (https://kubernetes.io/docs/reference/kubectl/)
[87] Command Line Tools for Container Management | Docker CLI (https://www.docker.com/products/cli/)
[88] A GitHub Action for authenticating to Google Cloud. https://github.com/google-github-actions/auth)
[89] Namespaces | Kubernetes https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)
[90] Quality Gates — A must have thing for the code analysis process | by Tarun Prakash | Medium https://medium.com/@tarunprakash/quality-gates-a-must-have-thing-for-the-code-analysis-process-75b33d6b49dc)

[91] Deployment Rollback | Kubernetes
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment)
[92] Use health checks | Load Balancing | Google Cloud
https://cloud.google.com/load-balancing/docs/health-checks)
[93] A. D. Boursianis, M.S. Papadopoulou, et al., "Advancing Rational Exploitation of Water Irrigation Using 5G-IoT Capabilities: The ARETHOU5A Project," 2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Rhodes, Greece, 2019, pp. 127-132. doi: 10.1109/PATMOS.2019.8862146.
[94] A. D. Boursianis, M.S. Papadopoulou, A. Gotsis, S. Wan, P. Sarigiannidis, S. Nikolaidis, and S.K. Goudos, "Smart Irrigation System for Precision Agriculture—The ARETHOU5A IoT Platform," in IEEE Sensors Journal, vol. 21, no. 16, pp. 17539-17547, 15 Aug.15, 2021, doi: 10.1109/JSEN.2020.3033526.
[95] Achilles D. Boursianis, Maria S. Papadopoulou, Panagiotis Diamantoulakis, Aglaia Liopa-Tsakalidi, Pantelis Barouchas, George Salahas, George Karagiannidis, Shaohua Wan, Sotirios K. Goudos, "Internet of Things (IoT) and Agricultural Unmanned Aerial Vehicles (UAVs) in smart farming: A comprehensive review," Internet of Things, 2022. doi: 10.1016/j.iot.2020.100187.
[96] George Xenofontos, Maria S. Papadopoulou, “Automated Plant Irrigation System Using Arduino Microcontroller,” 4th International Conference in Electronic Engineering, Information Technology & Education, Chania, Crete, Greece, 2023.
[97] George Xenofontos, Chrysostomos Koumides, Kyriakos Tsiakmakis, Argyrios T. Hatzopoulos, Maria S. Papadopoulou, “Smart-Irrigation Monitoring System using IoT Technology,” 9th International Workshop on Microsystems, Thessaloniki, Greece, 2024.
[98] F. T. Stoupas, A. T. Hatzopoulos, V.D. Vassios, M.S. Papadopoulou, “Conversion of a Climate Chamber to a Plant Growth Chamber,” 7th International Workshop on Microsystems, Thessaloniki, Greece, 2022.
[99] D. Liamos, K. Tsiakmakis, A.T. Hatzopoulos, Vassios V., M. Papadopoulou, “Study of Vegetation Health Indicators: GNDVI and Leaf Area Index in Precision Agriculture Applications,” 9th International Workshop on Microsystems, Thessaloniki, Greece, 2024.
[100] M. S. Papadopoulou, et al., "Dual-Band RF-to-DC Rectifier with High Efficiency for RF Energy Harvesting Applications," 2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, Germany, 2020, pp. 1-4, doi: 10.1109/MOCAST49295.2020.9200262.
[101] A. D. Boursianis, M. S. Papadopoulou, S. Nikolaidis and S. K. Goudos, "Modified Printed Bow-Tie Antenna for RF Energy Harvesting Applications," 2020 IEEE Microwave Theory and Techniques in Wireless Communications (MTTW), Riga, Latvia, 2020, pp. 67-71, doi: 10.1109/MTTW51045.2020.9245049.
[102] I. D. Bougas, M. S. Papadopoulou, K. Psannis, P. Sarigiannidis and S. K. Goudos, "State-of-the-Art Technologies in RF Energy Harvesting Circuits – A Review," 2020 3rd World Symposium on Communication Engineering (WSCE), 2020, pp. 18-22, doi: 10.1109/WSCE51339.2020.9275507.
[103] M. S. Papadopoulou, A. D. Boursianis, S. K. Goudos and K. Psannis, "Dual-Band Rectifier Design for Ambient RF Energy Harvesting," 2020 3rd World Symposium on Communication Engineering (WSCE), Thessaloniki, 2020, pp. 7-11, doi: 10.1109/WSCE51339.2020.9275569.
[104] Achilles D. Boursianis, Maria S. Papadopoulou, Aikaterini I. Griva, Vasileios P. Rekkas, Lazaros A. Iliadis, Sotirios P. Sotiroudis, Panagiotis Diamantoulakis, Thomas Lagkas, Panagiotis Sarigiannidis, Sotirios K. Goudos, Christos Christodoulou, George K. Karagiannidis, "Modified Bow-Tie Antenna

Design Using Artificial Hummingbird Algorithm for Wireless Power Transfer IoT Applications," 2023 17th European Conference on Antennas and Propagation (EuCAP), Florence, Italy, 2023, pp. 1-5, doi: 10.23919/EuCAP57121.2023.10133339.
[105] A.D. Boursianis, M. S. Papadopoulou, S. Nikolaidis, and S. K. Goudos, "Dual-Band Single-Layered Modified E-shaped Patch Antenna for RF Energy Harvesting Systems," 2020 European Conference on Circuit Theory and Design (ECCTD), Sofia, Bulgaria, 2020, pp. 1-4, doi: 10.1109/ECCTD49232.2020.9218354.
[106] Achilles D. Boursianis, Maria S. Papadopoulou, Marco Salucci, Alessandro Polo, Panagiotis Sarigiannidis, Stavros Koulouridis, Sotirios K. Goudos, "Frequency Selective Surface Design Using Coot Optimization Algorithm for 5G Applications," 2022 International Workshop on Antenna Technology (iWAT), Dublin, Ireland, 2022, pp. 184-187, doi: 10.1109/iWAT54881.2022.9810997.
[107] I. D. Bougas, M. S. Papadopoulou, A. D. Boursianis, P. Sarigiannidis, S. Nikolaidis and S. K. Goudos, "Rectifier circuit design for 5G energy harvesting applications," 2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST), Bremen, Germany, 2022, pp. 1-4, doi: 10.1109/MOCAST54814.2022.9837524.
[108] A. D. Boursianis, M. S. Papadopoulou, S. Koulouridis, A. Georgiadis, M. M. Tentzeris and S. K. Goudos, "Dual-Band Frequency Selective Surface Design Using Artificial Hummingbird Algorithm," 2022 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting (AP-S/URSI), Denver, CO, USA, 2022, pp. 982-983, doi: 10.1109/AP-S/USNC-URSI47032.2022.9886108.
[109] Ioannis D Bougas, Maria S Papadopoulou, Achilles D Boursianis, Panagiotis Sarigiannidis, Spyridon Nikolaidis, Sotirios K Goudos, "Energy Harvesting & Autonomous Energy Systems: A Proposal for RF Energy Harvesting," 2024 13th International Conference on Modern Circuits and Systems Technologies (MOCAST), Sofia, Bulgaria, 2024, pp. 1-5, doi: 10.1109/MOCAST61810.2024.10615601.
[110] A. D. Boursianis, M. S. Papadopoulou, S. Koulouridis, P. Rocca, A. Georgiadis, M. M. Tentzeris, and S. K. Goudos, "Triple-Band Single-Layer Rectenna for Outdoor RF Energy Harvesting Applications," Sensors, vol. 21, no. 10, p. 3460, May 2021, doi: 10.3390/s21103460.
[111] M. S. Papadopoulou, A. D. Boursianis, C. K. Volos, I. N. Stouboulos, S. Nikolaidis, and S. K. Goudos, "High-Efficiency Triple-Band RF-to-DC Rectifier Primary Design for RF Energy-Harvesting Systems," Telecom, vol. 2, no. 3, pp. 271–284, Aug. 2021, doi: 10.3390/telecom2030018.
[112] Espressif. ESP32 OTA Updates Documentation (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html)
[113] Firebase Docs. Firebase Cloud Messaging (https://firebase.google.com//cloud-messaging)
[114] Barbosa, G.L. et al. Comparison of land, water, and energy requirements of lettuce grown using hydroponic vs. conventional agricultural methods. Int J Environ Res Public Health. 2015. (https://www.mdpi.com/1660-4601/12/6/6879)
[115] Shamshiri, R.R. et al. Advances in greenhouse automation and controlled environment agriculture: A review. Computers and Electronics in Agriculture. 2018 (https://www.researchgate.net/publication/322834975_Advances_in_greenhouse_automation_and_controlled_environment_agriculture_A_transition_to_plant_factories_and_urban_agriculture)

Section 11: Diagrams/Images

Section 12: Tables

Section 13: Appendix

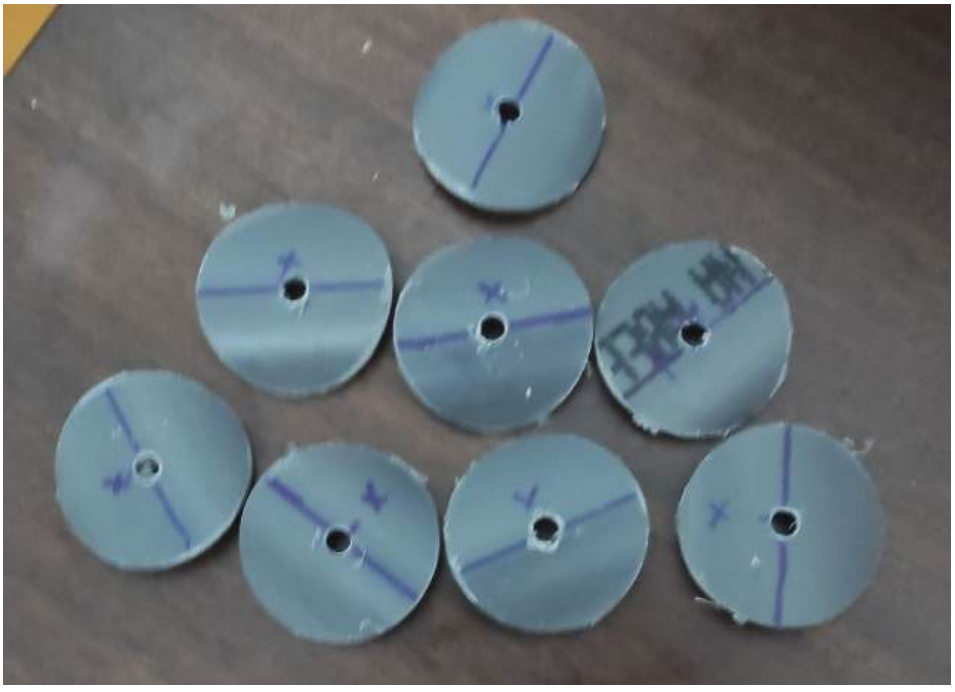

Image 13.1: PVC holes that were drilled for the pots

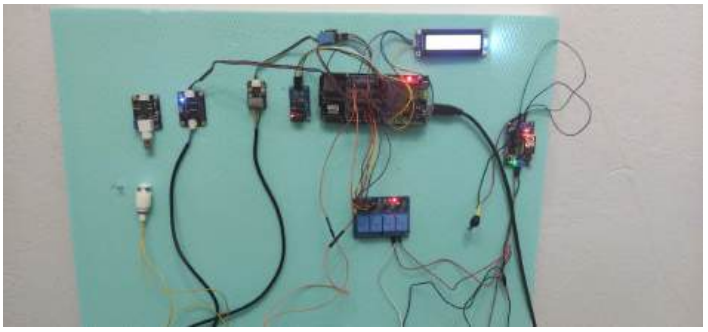

Image 13.2: First configuration of the sensors onto the shield pins

Image 13.3: Holes carved onto plastic cup to use as pot

Image 13.4: Construction Base

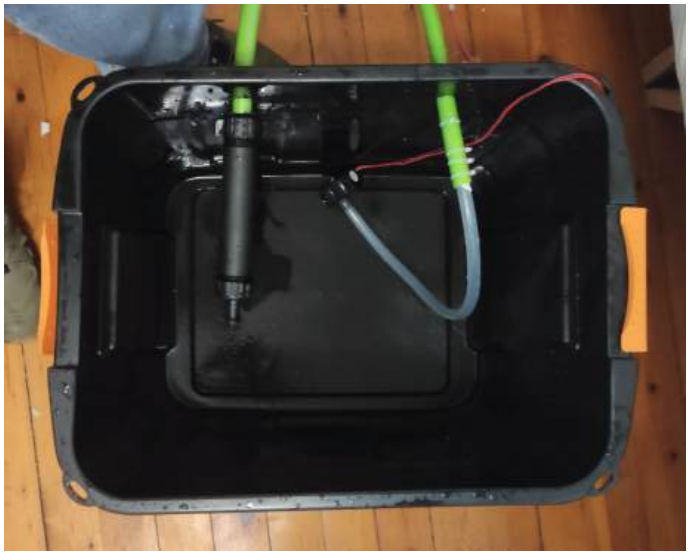

Image 13.5: The inside of the solution container (Output pipe is connected to pump and input pipe has the filter)

Image 13.6: Images of the test run when we first set it up with the sensors running

Image 13.7: First configuration of the fuze box with the sensors that go inside the container

Image 13.8: PH connector setted up at a block of styrofoam in order to float inside the container

Image 13.9: Construction of the two float sensors that will go inside the container

Image 13.10: The Arduino MEGA

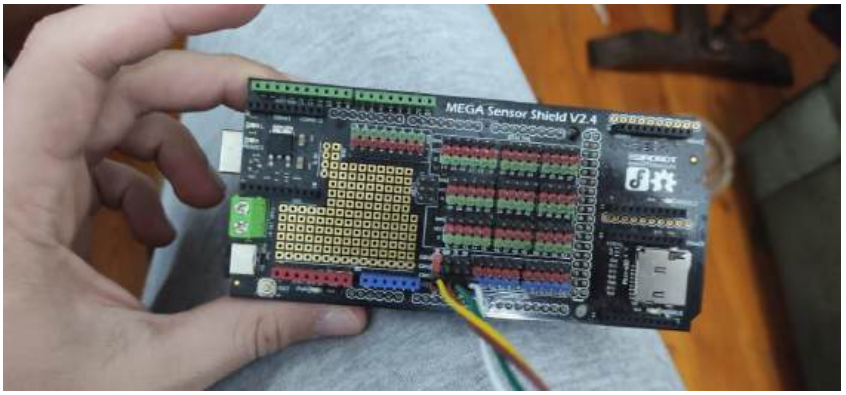

Image 13.11: MEGA Senior Shield V2.4

Image 13.12: Some of the tools used