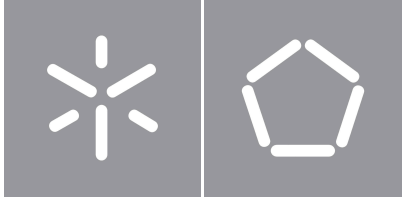

**University of Minho**
School of Engineering

Group 11

Daniel Gonçalo Silva Cardoso, PG53753
Marco Xavier Leite Costa, PG60210

# LeafSense: Real-Time Hydroponic Plant Monitoring System

January 2026



ESRG

**University of Minho**
School of Engineering

Group 11

Daniel Gonçalo Silva Cardoso, PG53753
Marco Xavier Leite Costa, PG60210

# LeafSense: Real-Time Hydroponic Plant Monitoring System

Implementation

Master's in Industrial Electronics and Computers Engineering
Embedded Systems and Computers

Project supervised by Professor
**Adriano José Conceição Tavares**

January 2026

# Contents

# List of Figures

# List of Tables

# List of Listings

# Glossary

This glossary provides definitions for key terms, acronyms, and technical concepts used throughout this document.

**AArch64** 64-bit ARM architecture (ARM64), used in the Raspberry Pi 4's Cortex-A72 processor.

**Acknowledge** User action to mark an ML recommendation as reviewed, updating the `user_acknowledged` field in the database.

**ADC (Analog-to-Digital Converter)** Hardware component (ADS1115) that converts analog sensor signals to digital values readable by the Raspberry Pi.

**ADS1115** 16-bit precision I2C-compatible ADC module used for reading analog sensors (pH, TDS).

**ADS7846** Resistive touchscreen controller chip used in the Waveshare 3.5" display.

**Alert** System-generated notification when sensor readings exceed predefined thresholds.

**Alert Threshold** Configurable confidence level (default 70%) above which ML predictions trigger system alerts.

**BCM2711** Broadcom System-on-Chip (SoC) powering the Raspberry Pi 4 Model B.

**Buildroot** Open-source tool for generating custom embedded Linux systems, used to create the LeafSense OS image (version 2025.08).

**BusyBox** Lightweight Unix utilities collection used for system initialization in embedded Linux.

**CMake** Cross-platform build system generator used for compiling the LeafSense C++ application.

**Condition Variable** POSIX synchronization primitive used for thread signaling and coordination.

**Cortex-A72** ARM processor core architecture in the Raspberry Pi 4, running at 1.8GHz.

**COTS (Commercial Off-The-Shelf)** Pre-built hardware or software components used in the project.

**Cross-Compilation** Compiling code on one architecture (x86) for execution on another (ARM64).

**Daemon** Background process running continuously without user interaction (e.g., database daemon).

**DS18B20** Digital waterproof temperature sensor using the 1-Wire protocol.

**DS3231** High-precision Real-Time Clock (RTC) module with I2C interface.

**DTO (Data Transfer Object)** Structured data container for passing information between system layers.

**EC (Electrical Conductivity)** Measurement of nutrient concentration in hydroponic solution, expressed in $\mu$S/cm.

**Entropy (Shannon)** Information theory metric measuring uncertainty in probability distribution; used for OOD detection. Formula: $H = -\sum p_i \log_2(p_i)$.

**ERD (Entity-Relationship Diagram)** Visual representation of database structure and table relationships.

**eudev** Device manager for Linux systems, handles dynamic device creation.

**Framebuffer** Memory buffer containing pixel data for display output (e.g., /dev/fb1).

**FIFO (First In, First Out)** Queue data structure used in message passing.

**glibc** GNU C Library, standard C library required for ONNX Runtime compatibility.

**GPIO (General Purpose Input/Output)** Configurable digital pins on the Raspberry Pi for hardware control.

**Green Ratio** Percentage of green/plant-like pixels in an image, used for OOD detection. Images with less than 5% green are rejected as non-plant.

**GUI (Graphical User Interface)** Qt5/Qt6-based touchscreen interface for user interaction.

**HAL (Hardware Abstraction Layer)** Software layer isolating application logic from hardware-specific code.

**HSV (Hue-Saturation-Value)** Color space used for plant color detection; green hue range is 35-85.

**Hydroponics** Soil-free cultivation method using nutrient-rich water solutions.

**Hysteresis** Control technique using upper and lower thresholds to prevent actuator oscillation.

**I2C (Inter-Integrated Circuit)** Two-wire serial communication protocol for sensors and peripherals.

**ILI9486** LCD display controller chip used in the Waveshare 3.5" touchscreen.

**IPC (Inter-Process Communication)** Mechanisms for data exchange between separate processes.

**ITC (Inter-Thread Communication)** Mechanisms for data exchange between threads within a process.

**Kernel Module** Loadable code extending Linux kernel functionality (e.g., LED driver).

**Kiosk Mode** GUI configuration running fullscreen without window manager overhead.

**LeafSense** Real-Time Hydroponic Plant Monitoring System developed in this project.

**libgpiod** Modern Linux library for GPIO control, replacing legacy sysfs interface.

**linuxfb** Linux framebuffer Qt platform plugin for direct display rendering.

**Message Queue** POSIX IPC mechanism for asynchronous data exchange between processes.

**Middleware** Software layer between hardware drivers and application logic.

**ML (Machine Learning)** AI techniques for automated plant disease and deficiency detection.

**MobileNetV3-Small** Lightweight convolutional neural network architecture optimized for mobile/embedded devices; used in LeafSense for plant classification.

**Multi-class Classification** ML task predicting one of several discrete categories (e.g., healthy, disease, deficiency, pest).

**Mutex** Mutual exclusion lock for thread synchronization and resource protection.

**NAT (Network Address Translation)** Network technique used for Raspberry Pi provisioning.

**NEON** ARM SIMD (Single Instruction, Multiple Data) extensions for image processing acceleration.

**ONNX (Open Neural Network Exchange)** Open format for ML model interoperability.

**ONNX Runtime** High-performance inference engine for executing ONNX models in C++.

**OOD (Out-of-Distribution)** Detection technique identifying inputs that differ significantly from training data; prevents false predictions on non-plant images.

**OpenCV** Open Source Computer Vision library for image capture and processing.

**OV5647** 5-megapixel camera sensor module for Raspberry Pi.

**OOP (Object-Oriented Programming)** Software design paradigm used throughout LeafSense.

**Peristaltic Pump** Precision fluid dosing pump for pH and nutrient adjustment.

**pH** Measure of acidity/alkalinity (0–14 scale), critical for plant health.

**PH-4502C** Analog pH sensor module used for water chemistry monitoring.

**POSIX** Portable Operating System Interface standard for Unix-like systems.

**pthreads** POSIX Threads library for multi-threaded programming in C/C++.

**Qt** Cross-platform application framework for GUI development (Qt5/Qt6).

**Qt Creator** Integrated development environment for Qt applications.

**Raspberry Pi** Single-board computer (Model 4B) serving as the LeafSense processing unit.

**Recommendation** Context-aware treatment suggestion generated by the ML pipeline, correlating predictions with current sensor readings.

**Recommendation Panel** Side-by-side GUI component in the Gallery tab displaying ML recommendations alongside captured images.

**RTC (Real-Time Clock)** Hardware module (DS3231) maintaining accurate time during power loss.

**SCP (Secure Copy Protocol)** SSH-based file transfer protocol for deployment.

**Softmax** Activation function converting raw model outputs (logits) into probability distribution summing to 1.0.

**SoC (System-on-Chip)** Integrated circuit combining CPU, GPU, and peripherals (BCM2711).

**SPI (Serial Peripheral Interface)** High-speed serial communication protocol for displays.

**SQLite3** Lightweight, serverless SQL database engine for persistent storage.

**SSH (Secure Shell)** Encrypted protocol for remote system access.

**TDS (Total Dissolved Solids)** Measurement of nutrient concentration in ppm (parts per million).

**Theme** Visual appearance mode (Light/Dark) for the GUI, configurable via Settings.

**Thread** Lightweight execution unit within a process, managed by pthreads.

**Toolchain** Collection of compilers and tools for cross-compilation (GCC 14.3.0).

**UML (Unified Modeling Language)** Standard notation for software design diagrams.

**USB-C** Power connector type for Raspberry Pi 4 (5V, 3A minimum).

**V4L2 (Video4Linux2)** Linux kernel API for video capture devices.

**Waveshare** Manufacturer of the 3.5” ILI9486 touchscreen display used in LeafSense.

**1-Wire** Single-wire communication protocol used by DS18B20 temperature sensor.

# Chapter 1

# Introduction

Hydroponics lets you grow more in less space and with far less water than traditional gardening, but it's also less forgiving: small shifts in pH, nutrients, or water temperature can quickly stress plants if they go unnoticed. The image below (Figure 1) shows a simple setup — seedlings in net pots with their roots hanging into a nutrient reservoir — exactly the zone where those chemistry and level changes matter most.

Figure 1: Seedlings with exposed roots in a nutrient reservoir

Because some problems show up in the solution and others appear first on the leaves, combining reservoir monitoring with leaf imaging makes a lot of sense. Sensors keep track of pH, EC/TDS, and water temperature so you can act on chemical issues, while periodic photos of leaves processed on the device can catch diseases or nutrient deficiencies early — often before yield is affected. Together, these two views allows the responsible worker to notice trouble sooner and make smarter, less frequent interventions.

## 1.1 Problem Statement

Hydroponic systems can dramatically increase yields and reduce water use, but they require continuous, coordinated control of multiple variables to keep plants healthy. Many small growers, hobbyists, and educators currently rely on occasional manual checks, which means they risk missing rapid changes or problems that can quickly impact plant health. As a result, either users miss these fast-moving issues due to infrequent monitoring or they are forced to constantly watch over the garden, which defeats the purpose of convenience and efficiency. Therefore, automating the monitoring and control process in hydroponic systems is essential to ensure optimal plant growth without demanding constant attention from the user.

## 1.2 Problem Statement Analysis

The proposed solution to this problem is a device consisting of an assortment of sensors monitoring key parameters for optimal plant growth: pH, electrical conductivity, and temperature of the water. In addition, the device will include a camera to periodically capture images of the plants' leaves, enabling the detection of early signs of diseases or nutrient deficiencies. The system will feature a touch display where the user can view the data acquired from the sensors, input the optimal ranges for these variables, and receive visual feedback about the plant health.

Actuators will be implemented to automatically adjust these parameters, making the hydroponic garden fully automated and ensuring optimal growing conditions for multiple plant types. The collected data will be continuously analyzed to maintain stability and quickly correct any deviations from the desired ranges. This automation minimizes the need for constant user supervision, allowing even inexperienced users to successfully maintain a healthy hydroponic system.

In addition, the integration of data logging and trend visualization will enable users to track changes over time, identify patterns, and make informed decisions about plant care. The system's modular design will allow for scalability and customization, adapting to different plant species or growing setups. Overall, this solution provides a reliable, intelligent, and user-friendly approach to managing hydroponic gardens, improving efficiency and maximizing crop yield.

# Chapter 2

# Analysis

## 2.1 Market Study

A market study is a process of gathering and analyzing information about a specific market to understand its size, trends, customer needs, and competitive environment. It helps ensure that any new solution is developed with a clear understanding of its context, increasing the chances for successful adoption and impact.

### 2.1.1 Market Overview

Hydroponics is a soil-free horticultural method in which plants are grown using water-based nutrient solutions. The market centers on optimizing plant growth through precise control of factors like temperature, humidity, lighting, and nutrient delivery. Hydroponic systems are designed to reduce waste of water and nutrients. Technologies such as specialized LED lighting, sensors, and data analytics help monitor and manage irrigation and nutrient supply. The sector mainly produces leafy greens, fruits, and cannabis (where legal), and its growth is propelled by the global push for sustainability, lower water use, reduced pesticides, urban food production, and smaller carbon footprints.

As it can be seen in Figure 2, the worldwide hydroponics market was valued at USD 5.53 billion in 2024 and is projected to grow from USD 6.23 billion in 2025 to nearly USD 18.12 billion by 2034, achieving a compound annual growth rate (CAGR) of 12.60% during the period from 2025 to 2034. This market expansion is fueled by growing worries over food security and a heightened demand for fresh produce that is free from pesticides.

The market's most notable insights are as follows:

- **Dominant Region:** Asia Pacific led with a 36% market share in 2024;

- **Fastest Growth:** Europe is forecasted to register the highest CAGR through 2034;
- **Type Segmentation:** Aggregate systems comprised 54% of the market in 2024 and liquid systems are expected to grow at a CAGR of 14.4%;
- **Crop Segmentation:** Tomatoes accounted for 46% of the market in 2024 and lettuce is projected for the fastest growth at a 15.6% CAGR;
- **Crop Area:** Farms above 50,000 sq.ft. dominated in 2024.

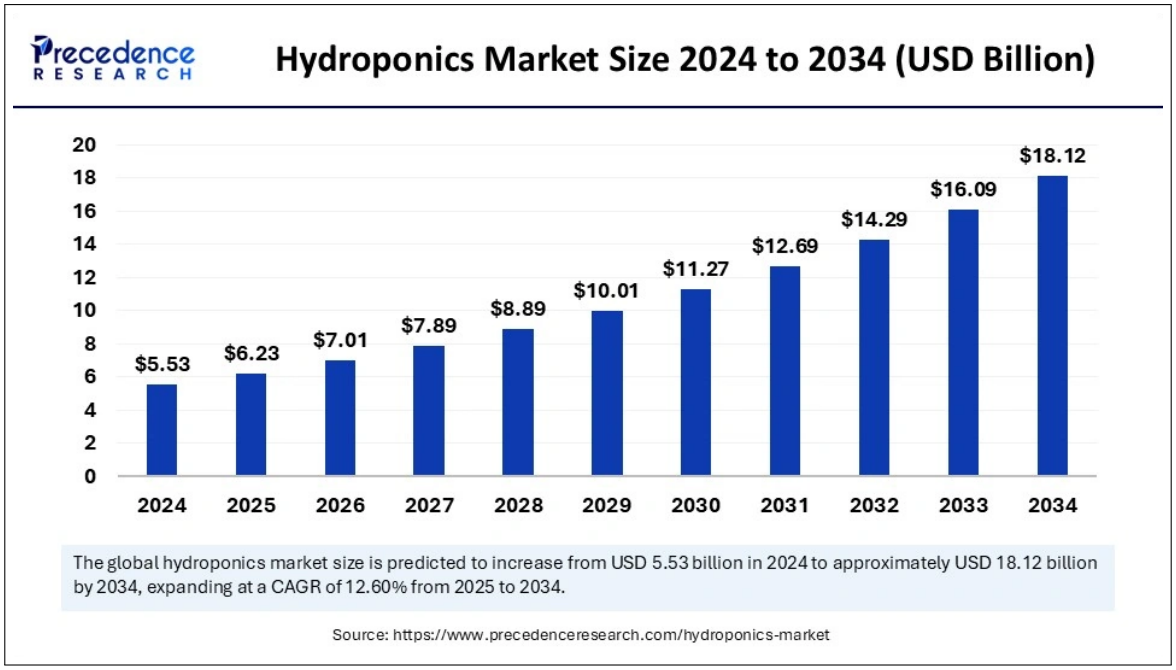



Figure 2: Hydroponics Market Size 2024 to 2034 (USD Billion)

### 2.1.2 The Role of AI in Hydroponics

Artificial intelligence is becoming a key driver of innovation in the hydroponics market. By integrating networks of sensors and cameras, hydroponic farms can continuously collect detailed data on environmental conditions and plant health. AI systems analyze this information in real time, allowing growers to automatically adjust irrigation, nutrients, and lighting to optimize conditions for each crop. This results in more efficient use of resources, higher yields, and consistent quality.

This technology also enhances early detection of plant stress, diseases, and pests, enabling timely interventions that minimize crop losses and reduce the need for chemical treatments. Furthermore, the adoption of AI-powered robotics for tasks like seeding and harvesting is lowering labor costs and improving

operational efficiency. With predictive analytics, farmers can better plan harvests and manage supply, making hydroponic production more reliable and scalable. Overall, AI is making hydroponics a smarter, more competitive solution for sustainable food production.

### 2.1.3 Hydroponics Global Market

Across the globe, a few regions have emerged as key drivers of the hydroponics market, with Asia Pacific, Europe, and North America taking the lead in 2024. While these areas currently set the pace for adoption and innovation in hydroponic farming, it is worth noting that regions such as Latin America, the Middle East, and Africa are also starting to embrace these advanced cultivation methods. Their growing interest signals promising opportunities for future expansion and diversification within the global market.

#### Asia Pacific

The Asia Pacific hydroponics market was valued at USD 1.99 billion in 2024 and is expected to reach USD 6.61 billion by 2034, growing at a CAGR of 12.75%. The region's leadership is mainly due to widespread adoption, especially in China, Australia, and South Korea. Urbanization has reduced available farmland and driven demand for city-friendly farming solutions, making hydroponics ideal for densely populated areas. With the region's fast-growing population, hydroponics offers a way to boost food output and strengthen food security. In India, where agriculture is prominent, there is growing recognition of the need for sustainable practices, and government initiatives are encouraging the use of hydroponics and other advanced farming technologies.

#### a) Trends in India

India is poised for the highest growth rate in hydroponics as its farming sector begins to embrace modern techniques to address food shortages and environmental pressures, creating new opportunities for innovative solutions.

#### b) Trends in Japan

Japan's hydroponics market is set to expand rapidly, supported by a national focus on agricultural innovation and sustainability, with government backing to maximize food production and advance farm technology.

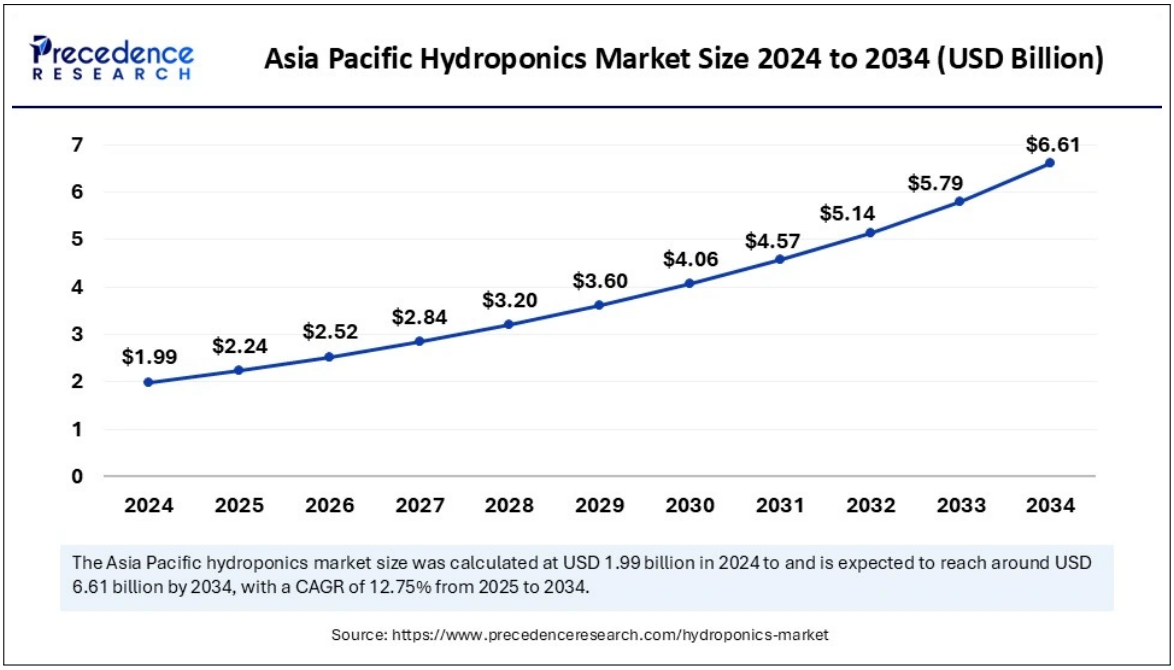



Figure 3: Asia Pacific Hydroponics Market Size 2024 to 2034 (USD Billion)

## Europe

Europe overall is forecasted to have the fastest-growing hydroponics market due to efforts to develop innovative, efficient indoor farming systems. Many European governments support sustainable agriculture through funding and initiatives encouraging hydroponic adoption. Germany's strength in engineering is advancing hydroponics technology and implementation.

### a) Trends in UK

The UK held a significant share of Europe's hydroponics market in 2024, driven by demand for eco-friendly and sustainable farming. The growing popularity of genetically modified crops is also contributing to market growth.

### b) Trends in Germany

Germany is expected to see robust growth, fueled by consumer demand for local, sustainably grown vegetables and adoption of advanced agricultural technologies for improved yields.

**North America**

North America is projected to experience substantial growth, thanks to numerous hydroponics-focused companies and rising acceptance of alternative farming in urban settings. Consumers in the region are increasingly prioritizing health, preferring fresh, locally sourced, and sustainable produce. The continent's strong tech infrastructure supports innovation in agriculture. In Canada, harsh winters make hydroponics an attractive solution for year-round indoor food production, and urban farming initiatives are expanding hydroponics' role in supplying health-conscious city populations.

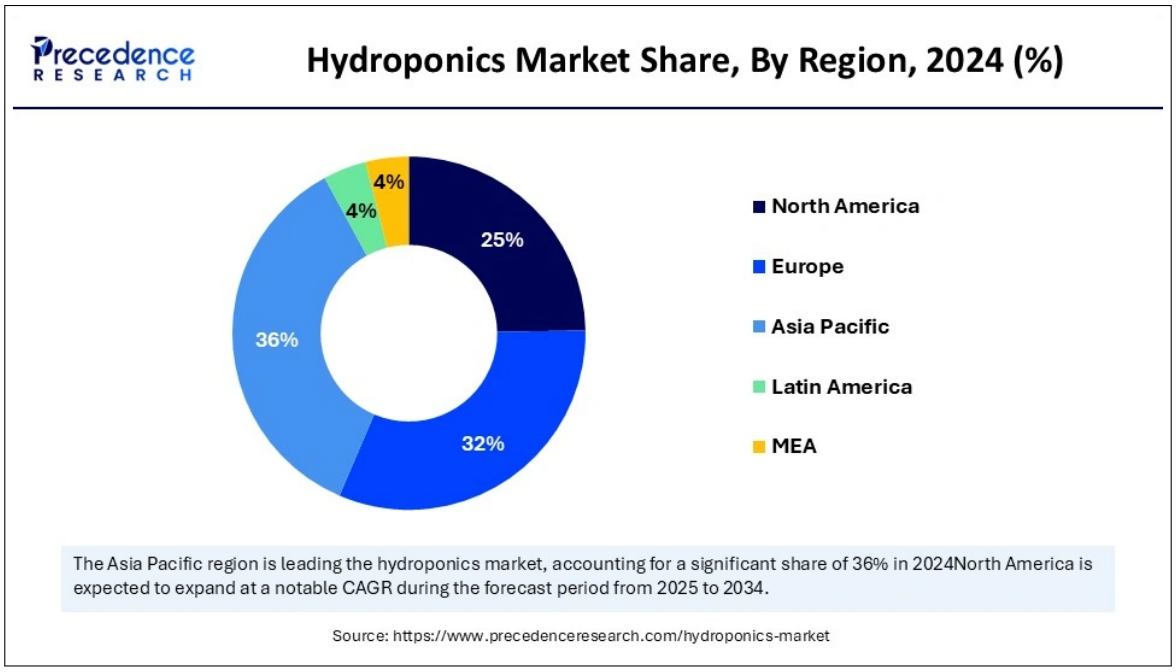



Figure 4: Hydroponics Market Share, By Region, 2024 (%)

## 2.1.4 Main Growth Factors for Hydroponics Use

- **Population Growth & Food Security:** Rising global population is increasing the need for high-quality produce. Hydroponics enables large-scale food production in limited spaces, ideal for regions with food security challenges or little arable land;
- **Declining Farmland:** Urban expansion and environmental decline are shrinking available farmland. Hydroponics offers a solution by enabling crop cultivation in urban and otherwise unsuitable locations;

- **Technological Progress:** Advances in LED lighting, nutrient delivery, automation, and sensors are making hydroponic systems more efficient. AI and data analytics allow growers to fine-tune conditions for maximum yields;
- **Premium Produce:** Consumers increasingly seek fresh, local, and pesticide-free food. Hydroponics supports year-round, high-quality crop production close to market, lowering transportation costs and environmental impact;
- **Government Support:** Governments worldwide are backing sustainable agriculture, including hydroponics, with research funding and supportive policies.

### 2.1.5 Market Dynamics

**Driver: Growing Food Demand**

Rising food requirements and shrinking farmland drive hydroponics adoption. These systems use advanced tech to grow fresh produce closer to consumers, cutting transport costs and carbon emissions. Consumer preference for high-quality, local food and awareness of hydroponics' benefits—food security, sustainability, and lower water use — are boosting market growth.

**Restraint: High Costs & Complexity**

The initial setup for hydroponics is expensive and requires technical know-how, which can be a barrier for some growers. These systems also depend on electricity for lighting, climate control, and nutrient circulation, increasing operational expenses. Public understanding of hydroponics' benefits is still limited, slowing wider adoption.

**Opportunity: Vertical Farming & Tech Innovations**

Vertical farming, which stacks crops to maximize space, offers higher yields. Continuous improvements in AI, automation, and sensor technology further enhance hydroponics. The method supports a wide range of crops, helping urban farmers meet demand for local food. Increasing government backing for sustainable agriculture and tailored nutrient solutions also present growth opportunities.

### 2.1.6 Hydroponic Systems Insights

Aggregate systems were the market leaders in 2024, thanks to their straightforward design and user-friendly setup. These systems use inert materials like peat, vermiculite, rock wool, sawdust, sand, perlite, or coconut coir to support plant roots, making them versatile for both large-scale and small-scale farms. They also incorporate technologies such as ebb and flow, drip irrigation, and wick systems, which help boost efficiency and simplify the growing process.

On the other hand, liquid systems are poised for the fastest growth. This is largely because more growers are turning to closed-system methods like deep-water culture (DWC) and nutrient film technique (NFT), especially for cultivating leafy greens. In these setups, plant roots are directly immersed in nutrient-rich solutions, which accelerates growth and leads to higher yields and better-quality produce.

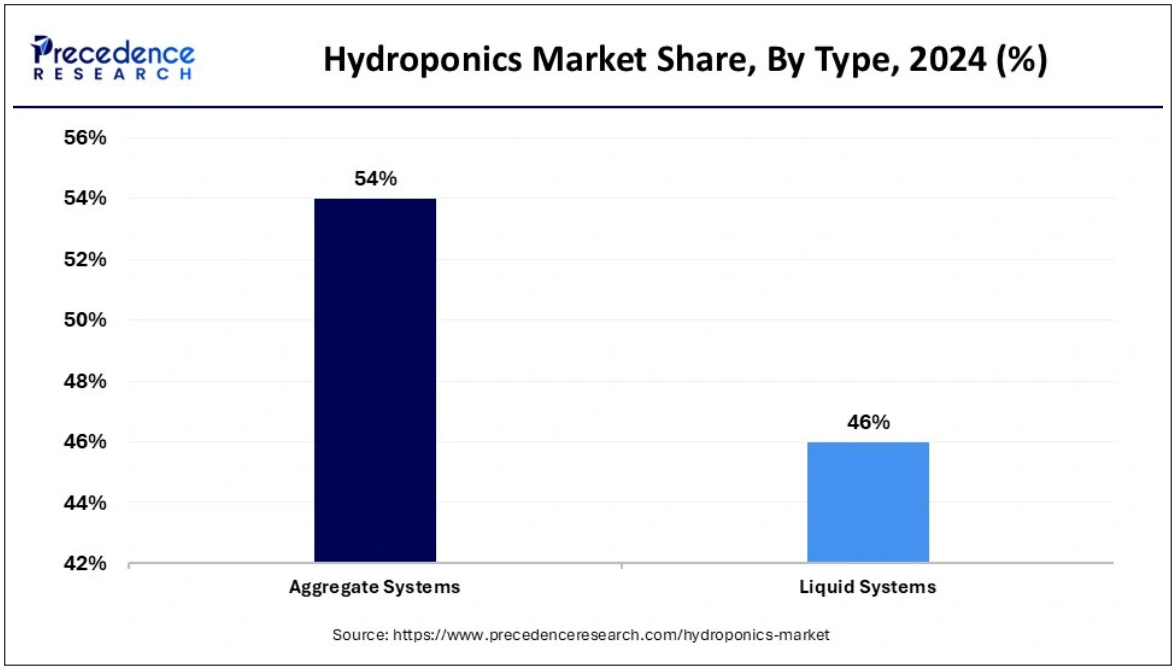



Figure 5: Hydroponics Market Share, By Type, 2024 (%)

#### Crop Type Insights

In 2024, tomatoes were the top choice for hydroponic growers, thanks to their quick growth and ability to thrive with less water than traditional farming methods. Growers often use materials like perlite, rock wool, or coconut coir to create the ideal environment for tomato plants, helping them develop strong roots and produce healthy fruit.

Meanwhile, lettuce is set to become the fastest-growing hydroponic crop in the coming years. Varieties like green and red leaf lettuce are especially popular for salads and restaurant dishes. As demand rises from both households and fast-food chains, more growers are turning to hydroponic systems to supply fresh, high-quality lettuce all year round.

**Crop Area Insights**

Farms larger than 50,000 sq. ft. led the market in 2024 and are expected to grow fastest. Bigger spaces allow for diverse, high-quality crop cultivation and the installation of multiple hydroponic systems and advanced machinery, raising efficiency and profitability. These large setups also conserve water better than traditional farming, making hydroponics attractive for scaling up sustainable agriculture.

### 2.1.7 Who are our competitors?

The hydroponics market includes a diverse range of monitoring and automation solutions designed to support plant health and optimize growing conditions. These products vary in terms of technology, integration, and user experience, with some focusing on environmental sensing and automation, while others offer customizable or modular approaches for different types of growers. The following section highlights notable solutions currently available, examining their core features and approaches to hydroponic system management.

**Growee Automated Hydroponic System**

- **Manufacturer:** Growee;
- **Features:**
  - Monitors pH, EC, temperature, and water levels;
  - Automated nutrient mixing and dosing;
  - Remote monitoring/control via cloud app;
  - User-friendly management interface;
  - No leaf imaging or AI-based visual diagnostics.
- **Price:** Starts at 379 € for the basic controller (full combos upwards of 855 €).

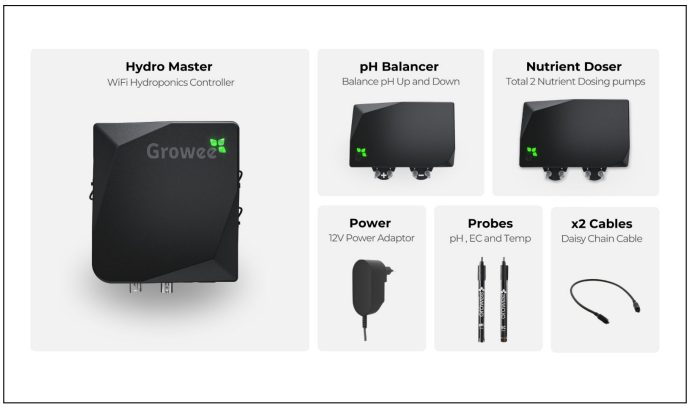



Figure 6: Growee Starter Combo – Dual pH Control & 2 Parts Dosing system

**Flora Pod**

- **Manufacturer:** Florasense;
- **Features:**

  - Monitors moisture, light, temperature, and nutrients;
  - Real-time wireless data via Bluetooth/Wi-Fi;
  - Data visualization and management in mobile app;
  - User-friendly interface;
  - No leaf imaging or advanced diagnostics.

- **Price:** Approx. 57 €–76 € per sensor.


Figure 7: Flora Pod

**Xiaomi Mi Flora Plant Monitor**

- **Manufacturer:** Xiaomi;
- **Features:**

  - Tracks moisture, light, temperature, and soil fertility;
  - Smartphone app integration for alerts and suggestions;
  - Adaptable for hydroponics;
  - Affordable and easy to use;
  - No imaging or automation beyond notifications.

- **Price:** Approx. 29 €–57 € per sensor.

Figure 8: Xiaomi Mi Flora Plant Monitor

**Flower Power**

- **Manufacturer:** Parrot;
- **Features:**

  - Tracks moisture, light, temperature, and soil fertility;
  - Smartphone app integration for notifications and suggestions;
  - Adaptable for hydroponics;
  - Affordable and easy to use;

- No leaf imaging or automation beyond alerts.

- **Price:** Approx. 29 €–57 € per sensor.


Figure 9: Parrot Flower Power

### 2.1.8 Why does our solution stand out among the others?

LeafSense differentiates itself by integrating both sensor-based reservoir monitoring and automated leaf imaging into a single, modular, and affordable system. Unlike most commercial products, LeafSense periodically captures and analyzes images of plant leaves using machine learning techniques, allowing for early detection of visual stress indicators such as disease, nutrient deficiencies, or infestations — capabilities rarely found in existing packaged solutions.

The system leverages a Raspberry Pi for data processing, local user interface, and actuator control, automating the adjustment of environmental conditions in real time. Its user-friendly touch display simplifies data visualization and input, while data logging and trend analysis support informed decision-making and ongoing optimization. LeafSense is designed for easy assembly and scalability, making it accessible for both hobbyists and small-scale growers, even with limited technical background or budget.

By combining advanced diagnostics, intelligent automation, and a cost-effective, customizable architecture, LeafSense provides a comprehensive and practical solution that stands out in the hydroponics market.


Figure 10: LeafSense Logo

## 2.2 System Requirements and Constraints

Requirements and constraints are essential in engineering projects because they define what must be achieved and set clear limits for the solution. Below, each is analyzed individually for this project.

### 2.2.1 Requirements

Requirements describe the functionalities and characteristics that the final system or product must have to satisfy the customer's needs and achieve its intended purpose. They can be either functional (defining what the system should do) or non-functional (defining how the system should perform), and are typically phrased as actions or expected behaviors.

**Functional**

- Measure pH, EC, and water temperature;
- Heat the water;
- Administer nutrients;
- Correct pH;
- Capture and process plant leaf photos;
- Warn about potential diseases and deficiencies and recommend action.

**Non-Functional**

- Low budget;
- Easy to assemble;
- Maintainable.

### 2.2.2 Constraints

Constraints are the conditions that limit or restrict the possible solution for the project. These can be technical, such as hardware or material limitations, or non-technical, like budget, schedule, or legal regulations.

**Technical**

- Raspberry PI board;
- Buildroot (for embedded Linux images generation);
- Device drivers (implementation of at least one);
- PThreads (for multitasking purpose);
- C++ programming language;
- Object-Oriented Programming (OOP) paradigm;
- UML diagrams (for design and documentation).

**Non-Technical**

- Two-member team;
- Limited budget;
- Fast time to prototype;
- Hard project deadlines.

## 2.3 System Overview

In Figure 11, an overview of our project system can be seen. The system will operate as follows: the user interacts with the touch display to input the optimal ranges for the plant they wish to grow. This information is then transferred to the processing unit, which controls the actuators to maintain the environment within these ranges. Sensors will periodically relay their readings to the processing unit, which will display the data to the user via the touch display. Simultaneously, the camera will capture leaf images at set intervals, and the processing unit will analyze them using image processing and machine learning techniques. Visual indicators such as discoloration, spots, or leaf deformities will be detected, and alerts or recommendations will be shown on the display. This integration allows for early intervention against diseases or nutrient deficiencies, improving plant health and yield. Over time, the system can learn from new images, continuously refining its ability to recognize specific issues for different plant types in the hydroponic setup.

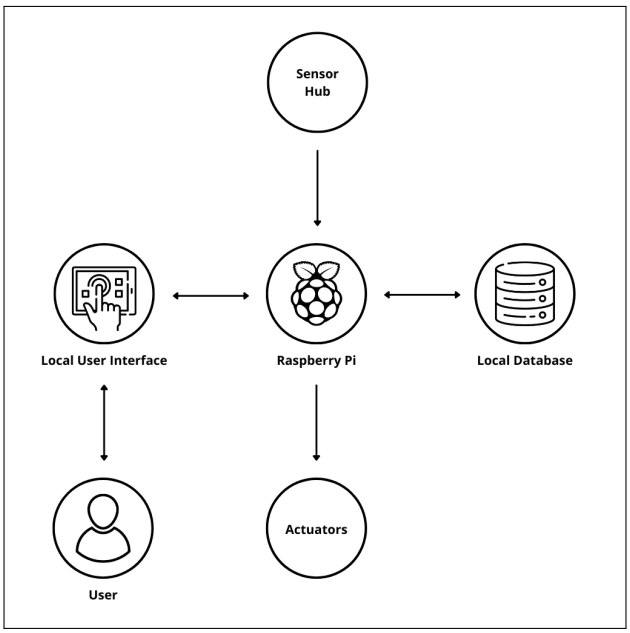



Figure 11: LeafSense System Overview

## 2.4 System Architecture

System Architecture describes the overall design and organization of a system, covering its key components, how they are connected, and how they interact to fulfill defined requirements. It offers

a broad perspective on the system, explaining its structure, the collaboration between different parts, and the technologies used. System Architecture is typically categorized into two main types: Hardware Architecture and Software Architecture.

### 2.4.1 Hardware Architecture

Hardware architecture defines how the physical components of the system are organized and connected to work together efficiently. In LeafSense, the hardware structure (Figure 12) centers around the Raspberry Pi, which serves as the main controller. It connects to the Sensor Hub — composed of a camera for leaf imaging, a pH sensor, a EC/TDS sensor, and a water temperature sensor. The Raspberry Pi also interfaces with actuators such as water heater, nutrient dosing pump, and pH dosing pump for system automation. A local user interface allows users to interact with the system, while a dedicated power supply ensures stable operation for all modules.

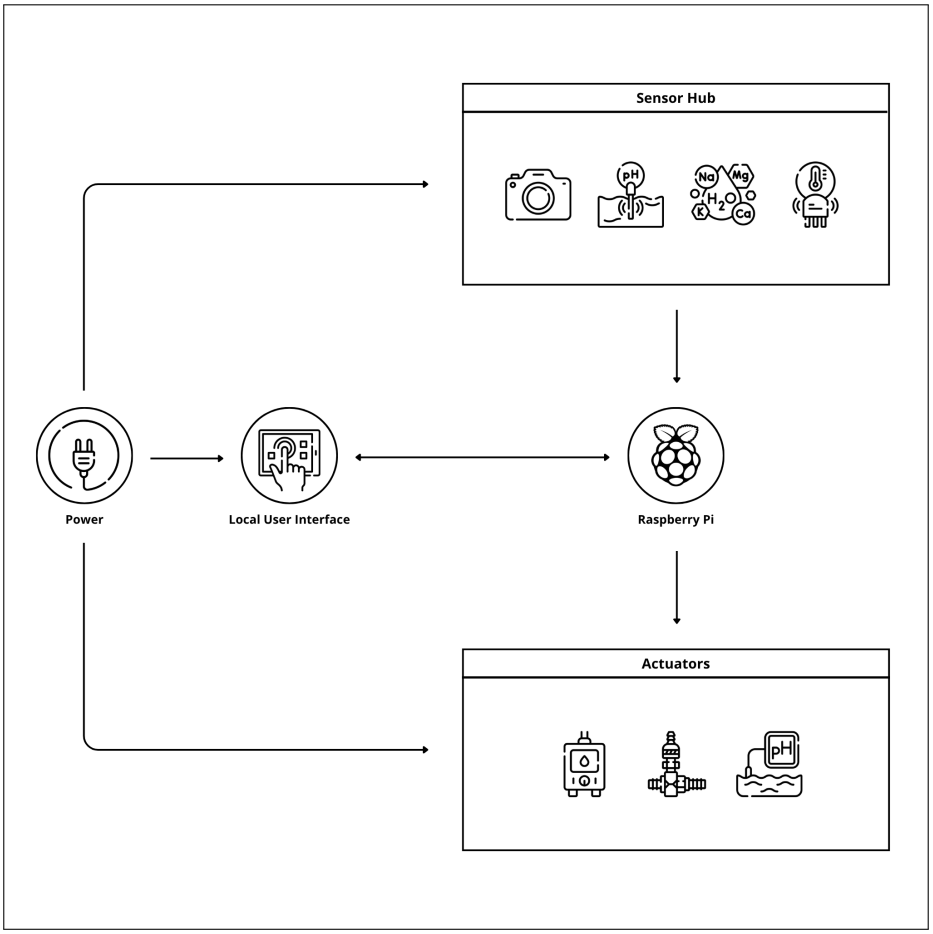



Figure 12: LeafSense Hardware Architecture

### 2.4.2 Software Architecture

Software Architecture is the high-level blueprint of a software system, outlining its main components, their relationships, and the guiding principles behind its design and development. The construction of this architecture significantly impacts system reliability, scalability, maintainability, and its ability to fulfill user requirements.

As represented in (Figure 13), our system's software architecture is organized into three distinct layers:

- **Custom Linux OS (Bottom Layer):** This foundational layer provides the interface between hardware and software, containing all the device drivers necessary to communicate with sensors, actuators, and other hardware components;
- **Middleware (Middle Layer):** Middleware acts as a bridge between the operating system and the application layer. It is responsible for managing multitasking (thread control), acquiring data from connected devices, operating the local database engine, and running ML Image Analysis Engine for leaf diagnostics. This ensures efficient data handling and processing throughout the system;
- **Application (Top Layer):** The application layer consists of the software tools that users interact with directly. In our system, this includes the graphical user interface (GUI) for monitoring and control, actuators control mechanisms for managing the hydroponic environment, and leaf imaging & disease detection tools for plant health analysis.

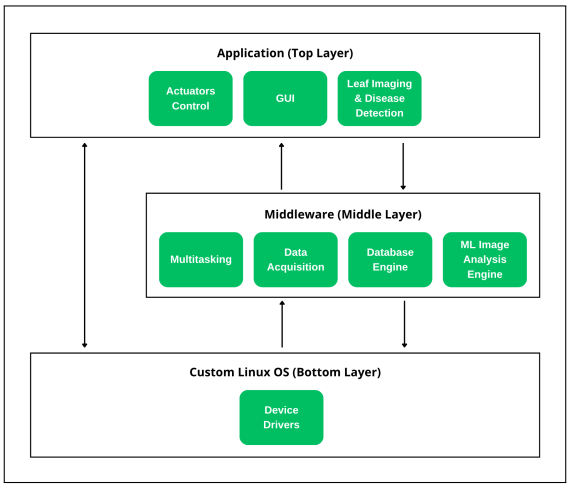



Figure 13: LeafSense Software Architecture

## 2.5 Technology Stack

Figure 14 presents the Technology Stack as a sequence of layers, each representing the distinct technologies and components that comprise the system and contribute to its efficient operation. These layers are arranged to interact with one another, both directly and indirectly, ensuring smooth system performance.

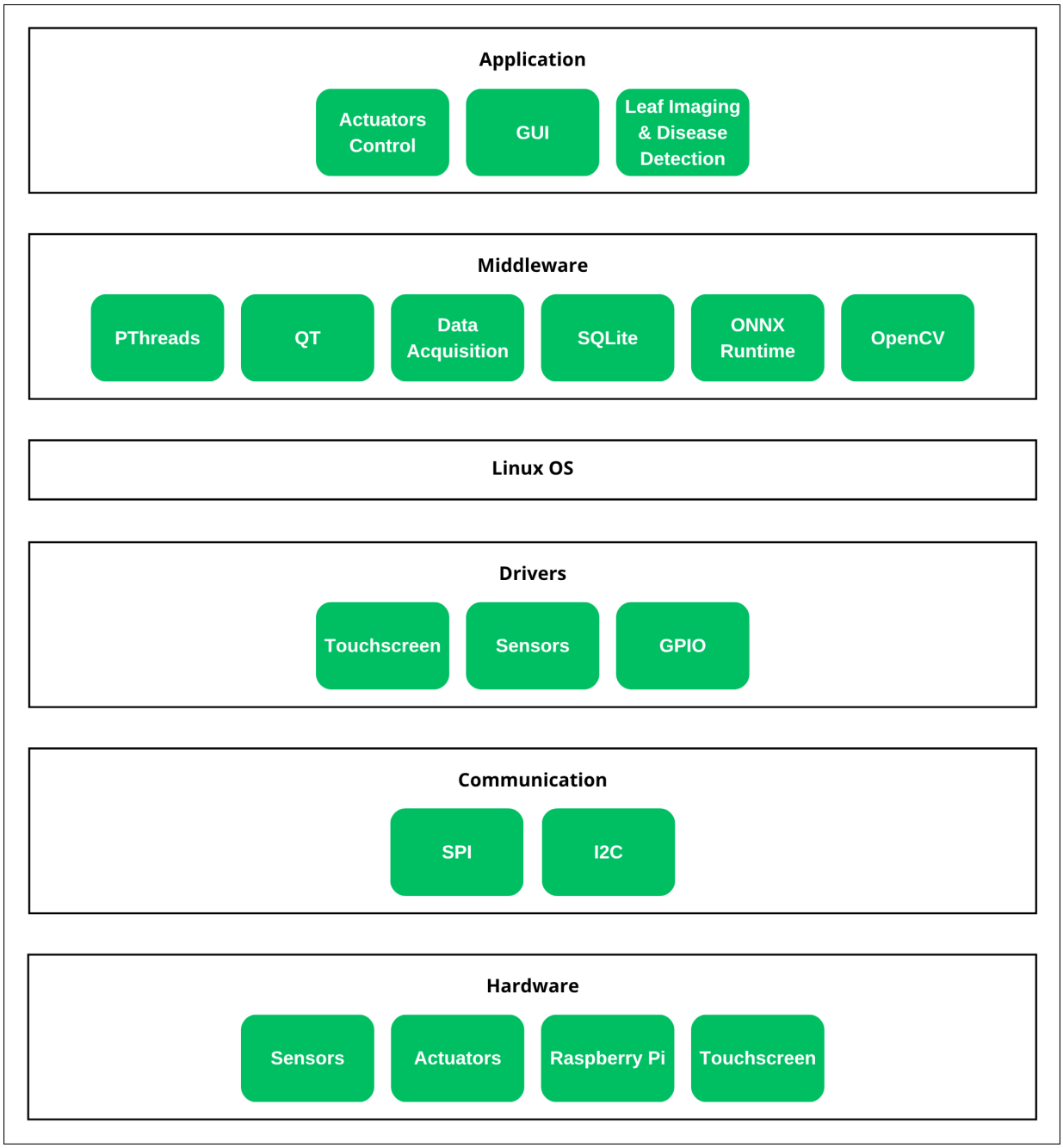



Figure 14: LeafSense Technology Stack

Each layer serves a specific function, beginning with foundational hardware and operating systems at the base, and progressing to application and user interface layers at the top. This layered structure enables modularity, making it easier to manage, update, and scale individual components as the project evolves. By clearly outlining the role of each technology and component within the stack, the architecture promotes organization, efficiency, flexibility, and reliability throughout the development and deployment of the system.

## 2.6 System Analysis

System Analysis is a comprehensive process that examines a system's requirements, structure, and behavior to ensure it fulfills its intended objectives. The use of diagrams and visual tools is integral to this approach, clarifying and communicating complex aspects of the system in an accessible manner.

Through visual representations, requirements, processes, and interactions can be effectively documented, which supports clear communication and alignment among all parties involved. This methodical analysis not only helps identify potential challenges and opportunities for improvement, but also facilitates informed decision-making throughout both the design and implementation phases. By incorporating diagrams and visual tools, System Analysis contributes to a more efficient system design, reduces misunderstandings, and ensures that everyone remains engaged and informed throughout the development lifecycle.

### 2.6.1 System Events

The system continuously monitors and identifies a variety of events that occur during its operation. Upon detection, it responds appropriately based on predefined actions associated with each event. Table 2.1 offers a comprehensive overview of these events, specifying not only the system's corresponding responses but also detailing the sources or triggers that initiate them and their respective types. This structured approach enables the system to efficiently manage dynamic situations, ensuring that all possible events are accounted for and addressed in a consistent manner. By clearly mapping the relationships between events, actions, sources, and types, the table serves as a valuable reference for understanding the system's behavior and improving both its reliability and effectiveness.

| Event | System Action | Source/Trigger | Type |
|---|---|---|---|
| Power on | Initialize all devices, establish database connection | User | Asynchronous |
| Emergency shutdown | Stop all actuators, save critical data, alert user | System/User | Asynchronous |
| Tap touchscreen | Analyze touch input, update interface | User | Asynchronous |
| Ideal Conditions changed | Validate and apply new parmeters, log changes | User | Asynchronous |
| Read pH | Acquire pH data, validate range, store value | System/Timer | A/Synchronous |
| Read EC/TDS | Acquire conductivity data, calculate TDS, store value | System/Timer | A/Synchronous |
| Read water temperature | Acquire temperature data, validate range, store value | System/Timer | A/Synchronous |
| Take photo | Activate camera, capture leaf image | Timer | Synchronous |
| Analyze image | Process image with ML, detect plant issues | System | Synchronous |
| Disease/Deficiency detected | Alert user, recommend corrective action, log detection | System | Asynchronous |
| Heat water | Activate water heater to mantain optimal temperature | System | Asynchronous |
| Regulate pH | Activate pH dosing pump (up/down) based on ph reading | System | Asynchronous |
| Dose nutrients | Activate nutrient dosing pump | System | Asynchronous |
| Sensor/Actuator failure detected | Display error, disable automation, log failure | System | Aynchronous |
| Save data | Store sensor readings and system status in database | System | Synchronous |

Table 2.1: LeafSense System Events

### 2.6.2 Use Case Diagram

A Use Case Diagram is a visual representation used in software and system design to show how various actors interact with the system. It presents the system's functionality from the perspective of the actors, emphasizing the primary actions or "use cases" that the system carries out. This diagram provides a clear overview of the system's overall scope and it's requirements.

In our case (Figure 15), the system operations are managed by 4 distinct actors — User, Timer, ML Engine, and Database Engine —, providing comprehensive cultivation support through continuous monitoring, intelligent analysis, and proactive intervention.

The User interacts with three independent use cases: Configure System for establishing optimal ranges, View System Dashboard for accessing historical data and reports and Manage Alerts for reviewing system notifications. The Timer actor drives automated operations through two scheduled use cases. Read Sensor Data executes periodically to collect environmental measurements, while Capture Plant Images operates daily to document plant development. The ML Engine actor processes visual data through ML Analysis & Assessment and creates intelligent guidance through Generate Recommendations. The Database Engine supports all operations through Manage Data Services, providing comprehensive data storage and retrieval.

The system employs include relationships to represent mandatory dependencies. Monitor Environment «include» Read Sensor Data ensures that environmental monitoring always accesses current sensor measurements. Read Sensor Data «include» Process Sensor Data creates automatic analysis of every collected reading against optimal ranges. Capture Plant Images «include» ML Analysis & Assessment guarantees that every photograph triggers computer vision processing. ML Analysis & Assessment «include» Generate Recommendations ensures that completed health analyses automatically produce actionable treatment suggestions.

Two extend relationships represent conditional functionality. Control Actuators «extend» Process Sensor Data activates only when environmental parameters exceed acceptable thresholds, implementing corrective measures such as pH adjustment or temperature regulation. Generate Recommendations «extend» Manage Alerts triggers urgent notifications only when critical conditions require immediate user attention, preventing unnecessary alert generation for routine recommendations.

The comprehensive database integration maintains sensor readings, machine learning results, actuator logs, system alerts, user configurations, and extensive reference libraries supporting disease detection and deficiency identification. This architecture delivers automated environmental control with intelligent health assessment, reducing manual monitoring requirements while improving cultivation outcomes through early problem detection and proactive intervention recommendations.

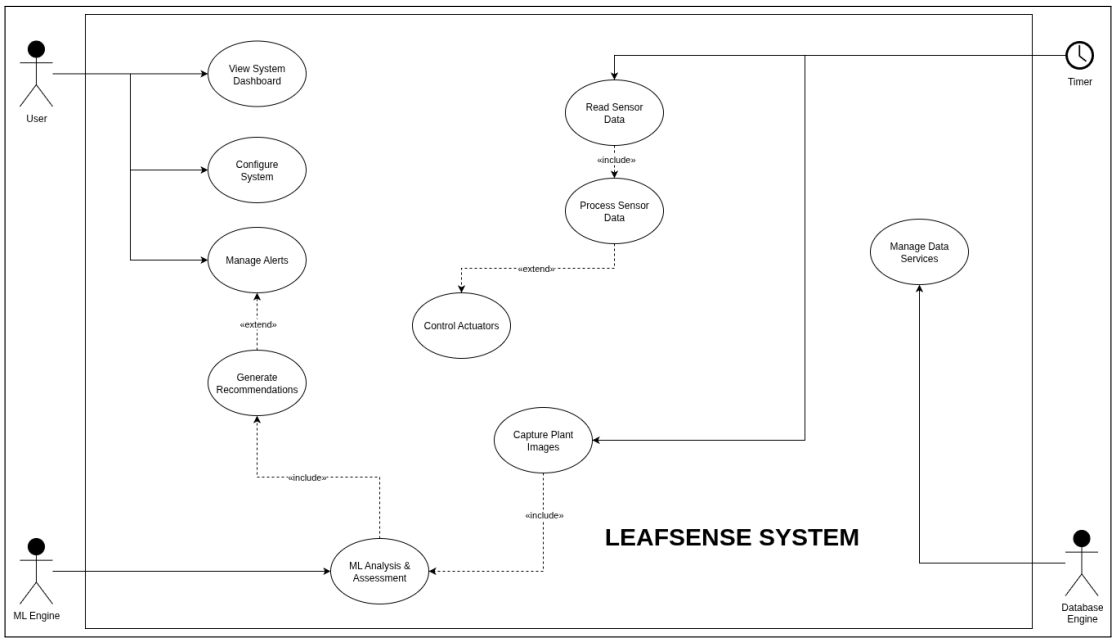



Figure 15: LeafSense Use Case Diagram

### 2.6.3 Entity-Relationship Diagram

An Entity-Relationship Diagram (ERD) is a visual representation used to model the organization of data and the connections between different components in a database system. It is composed of three key elements. Entities, which are illustrated as rectangles, represent objects or concepts within the system and typically correspond to tables in the database. Attributes, displayed as subrectangles, define the details or properties of each entity, including the primary key (PK) that uniquely identifies every instance. Relationships, shown as connecting lines, illustrate how entities are linked to one another.

To further clarify these connections, relationships are often depicted using Crow's Foot Notation. This notation uses distinct symbols at the ends of relationship lines to indicate cardinality — the number of entity instances that can be associated with another entity. For instance, a single line denotes a one relationship, while a crow's foot symbol indicates many. This enables the diagram to express different types of associations, such as one-to-one (1:1), one-to-many (1:N), or many-to-many (N:N). Additionally, optionality is indicated by a circle for optional relationships or a solid line for mandatory relationships, making the ERD a powerful tool for understanding the structure and rules governing the data within a system.

The LeafSense Entity-Relationship Diagram (Figure 16) is built on three core principles: simplicity through constraints, implicit relationships for core operational data, and explicit relationships with proper foreign keys for machine learning components. By enforcing that exactly one user and exactly one plant exist in the system at any time, the database eliminates the complexity of multi-tenancy while remaining flexible enough to expand in the future if needed.

#### Database Architecture and Core Design Philosophy

The LeafSense database operates within well-defined boundaries. There is always exactly one user — the system administrator — who monitors exactly one plant in a hydroponic system. This user never changes during the system's operation, and the plant being monitored remains constant. These constraints might seem limiting, but they are actually the source of the system's elegance and efficiency.

Rather than using complex user_id or plant_id foreign keys scattered throughout the database, LeafSense leverages SQL CHECK constraints to enforce these limits at the database level. The user table has a primary key of 1 with a CHECK constraint ensuring only id = 1 can exist. Similarly, the plant table has an identical constraint. This means no second user can ever be inserted, and no second plant can ever be added — the database itself prevents these actions.

This design philosophy cascades through the rest of the system. All alerts belong to the single user

implicitly. All sensor readings belong to the single plant implicitly. All logs, all images, all predictions — everything operates within the context of this one user and one plant. This eliminates the need for user_id and plant_id columns in dozens of places, reducing storage overhead, simplifying queries, and making the system faster and easier to understand.

However, the machine learning component introduces a different pattern. When ONNX Runtime analyzes images and produces predictions, and when users verify those predictions, we need explicit tracking and referential integrity. Here, the database uses proper foreign key constraints to ensure that predictions reference valid images, and that detection records reference valid predictions. This hybrid approach — implicit relationships for core data and explicit relationships for ML data — represents the best of both worlds.

**The USER Table: Authentication and System Foundation**

Every system needs authentication, and LeafSense is no exception. The USER table stores the credentials for the single administrator who operates the system. This table is simple by design, containing just five fields: an id that is always 1, a username for login, a password hash for security, and two timestamp fields to track when the account was created and when it was last accessed.

The user table is initialized with a default admin account during database setup. The username is "admin" and the password — for demonstration purposes only — is "admin", stored as a SHA-256 hash. In a production environment, this would be replaced with a stronger hashing algorithm like bcrypt or Argon2, which are specifically designed for password storage and naturally resist brute-force attacks.

The id field in the user table has a CHECK constraint: CHECK(id = 1). This seemingly simple constraint is actually the first line of defense against data corruption. If someone tries to insert a second user with id = 2, the database immediately rejects it with a constraint violation error. If someone tries to update the existing user to have id = 2, the database refuses. This prevents accidental multi-user scenarios that the rest of the system is not designed to handle.

The last_login timestamp serves a practical purpose. Every time the user logs in, the application updates this field. Over time, this field can be used to understand usage patterns, implement account lockout policies, or simply display to the user when they last accessed the system. The field is nullable because the user starts with no login history when the account is first created.

The relationship between USER and PLANT is a 1:1 relationship where both sides are mandatory. The single user must monitor the single plant, and the single plant must be monitored by the single user. By design and by database constraint, this relationship is locked in place and cannot be violated.

### The PLANT Table: The Entity Being Monitored

If the USER table is the administrator, the PLANT table is what the administrator is managing. It stores metadata about the single plant growing in the hydroponic system. Like the user table, the plant table has exactly one row, enforced by a CHECK constraint on its id field.

The plant table contains five fields. The id is always 1. The name field stores the type of plant being grown — perhaps "Lettuce", "Tomato", "Spinach", or any other variety suitable for hydroponics. The planted_date field records when the plant was first introduced to the system; this helps the user understand how mature the plant is and what stage of growth it should be in. The created_at and updated_at fields are audit timestamps, showing when the plant record was initially created and when it was last modified.

When a user upgrades their system to grow a different plant, they don't create a new row in the plant table. Instead, they update the existing row, changing the name from "Lettuce" to "Tomato" and resetting or updating the planted_date. The plant_id remains 1 throughout the system's operation, anchoring all sensor readings, all images, and all ML analyses to this single entity.

The relationship between PLANT and SENSOR_READINGS is a 1:M relationship where the plant side is mandatory (the plant must have a context for sensor readings) and the sensor_readings side is optional (the plant can exist before any sensors have taken readings). Similarly, PLANT has a 1:M relationship with PLANT_IMAGES, where the plant must logically own the images but can start with zero images before the camera begins capturing.

### The SENSOR_READINGS Table: Continuous Monitoring

At the heart of any monitoring system is data collection, and SENSOR_READINGS is where LeafSense stores its most fundamental data. Every few minutes — typically every 5, 15, or 30 minutes depending on system configuration — the Raspberry Pi reads three sensor values: temperature in degrees Celsius, pH value from 0 to 14, and electrical conductivity (EC) measured in microSiemens per centimeter. Each measurement is immediately inserted into the SENSOR_READINGS table as a new row with a timestamp.

Over time, SENSOR_READINGS grows into a rich time-series dataset. A system taking readings every 5 minutes generates 288 readings per day, which accumulates to approximately 105,000 rows per year. On a typical Raspberry Pi with an SD card of 16 to 32 gigabytes, this represents less than 10 megabytes of storage per year — negligible by modern standards.

The beauty of this time-series data is that it creates a complete historical record. Want to know what the temperature was three months ago? Query SENSOR_READINGS with a WHERE clause filtering by date. Want to see how pH has changed over the course of a week? Query the same table and calculate

averages. Want to find anomalous readings—temperatures that suddenly spike or drop, or pH values that drift out of acceptable range? Query SENSOR_READINGS and look for outliers.

Each row in SENSOR_READINGS has five fields. The id is an auto-incrementing primary key, giving each reading a unique identifier. The temperature, ph, and ec fields are real numbers — floats that can store decimal values. These fields are nullable because if a sensor malfunctions or temporarily disconnects, the system can still insert a row with NULL values for the broken sensor while preserving readings from functioning sensors. The timestamp field records when the reading was captured, defaulting to the current time if not explicitly provided.

The SENSOR_READINGS table has an index on the timestamp column. This index is crucial for performance because many queries filter or sort by timestamp: "Get me the last 10 readings", "Get me all readings from the last 24 hours", "Show me the temperature trend over the last week". Without the index, each of these queries would require scanning the entire table. With the index, the database can jump directly to the relevant rows, making these queries run in milliseconds instead of seconds.

All sensor readings belong to the single plant (id = 1) implicitly. There is no plant_id column in SENSOR_READINGS because the system design guarantees that all readings are for the same plant. This eliminates a redundant column, saves storage space, and simplifies the application code that inserts and queries these readings.

**The ALERTS Table: Threshold-Based Warnings**

While SENSOR_READINGS stores raw data, the ALERTS table stores automatic warnings generated when that data crosses predefined thresholds. If the pH drops below 5.5 or rises above 7.5, an alert is generated. If the temperature drops below 15°C or rises above 28°C, an alert is generated. If the electrical conductivity spikes above 2000 µS/cm, indicating an excess of nutrients, an alert is generated.

Unlike operational logs which are created by user actions, alerts are created by the system itself when it detects a problem condition. These alerts are designed to be temporary and actionable. A user sees an alert on their dashboard, acknowledges it by marking it as read, and takes corrective action — perhaps by adjusting pH using pH-up or pH-down chemicals, or by refreshing the nutrient solution if EC is too high.

The ALERTS table has six fields. The id is an auto-incrementing primary key. The type field categorizes the alert by severity: "Warning" for a condition that should be addressed soon, "Critical" for a condition that needs immediate attention, or "Info" for informational alerts. The message field contains a short, human-readable description: "pH too high", "Temperature warning", "EC out of range". The de-

tails field provides additional context: "Current pH: 8.2 (Ideal: 6.5-7.5)" or "Current Temperature: 26.5°C (Ideal: 20-25°C)".

The is_read field is perhaps the most important from a user experience perspective. It's an integer that stores 0 for unread alerts or 1 for alerts the user has acknowledged. When the user opens the dashboard, the application queries for all alerts where is_read = 0 to show unread notifications. This field allows users to track which alerts they have processed without deleting them from history. An index on is_read enables fast filtering of unread alerts, ensuring the dashboard notification count updates quickly.

The timestamp field records when the alert was generated. Over time, a complete alert history accumulates a record of every time the system detected an out-of-range condition. This history is valuable for understanding system problems and plant health trends.

All alerts implicitly belong to the single user (id = 1). When the dashboard displays alerts, it simply queries the ALERTS table without filtering by user_id, because there is only one user.

**The LOGS Table: Unified Operational History**

If alerts are automatic warnings, logs are the operational history of the entire system. The LOGS table is where all significant events are recorded: diseases detected by the ML model, nutrient deficiencies identified, maintenance actions performed, and threshold alerts that were acknowledged. This single table, rather than separate disease and deficiency and maintenance tables, serves as the unified source of truth for everything that has happened to the plant.

The LOGS table has four fields. The id is an auto-incrementing primary key. The log_type field categorizes the entry: "Disease" for detected plant diseases like Powdery Mildew or Leaf Spot, "Deficiency" for detected nutrient deficiencies like Nitrogen or Phosphorus deficiency, "Maintenance" for user-performed actions like water changes or filter cleaning, or "Alert" for significant threshold violations that were logged. The message field contains the main event description, and the details field provides additional context.

This design — a single logs table with a log_type field — perfectly matches the UI design shown in the system mockups. The Logs page displays all entries ordered by timestamp, with four button tabs labeled "Alerts", "Diseases", "Deficiencies", and "Maintenance". Clicking each tab filters the LOGS table by log_type using a simple WHERE clause. This is far more maintainable than having four separate tables and then writing complex UNION queries to display them together.

An index on the log_type column enables fast filtering when these tabs are clicked. Without the index, filtering a table with thousands of log entries by category would require scanning every row. With the index, the database can jump directly to all "Disease" entries or all "Maintenance" entries in milliseconds.

Consider a practical example. An ONNX model analyzes a leaf image and detects Powdery Mildew with 92% confidence. The user reviews the prediction, confirms that yes, it is indeed Powdery Mildew, and logs that they have applied a Sulfur spray treatment. The application automatically creates a log entry: log_type='Disease', message='Powdery Mildew', details='ML Confidence: 92%, Treatment: Sulfur spray applied'. Now when the user opens the Logs page and clicks the "Diseases" tab, this entry appears in the list along with any other disease events from the past.

All log entries implicitly belong to the single user (id = 1). There is no user_id column because the system design guarantees all logs are for the same user. This simplifies the application logic that creates and queries logs.

### The PLANT_IMAGES Table: Image Capture and Storage

The PLANT_IMAGES table introduces the machine learning component of LeafSense. This table stores metadata about images captured by an OpenCV-based camera module connected to the Raspberry Pi. Every few hours (or at user-defined intervals), the system captures a new image of the plant's leaves and stores it on the file system. The database doesn't store the image itself — that would be inefficient — but rather stores metadata about the image: where it's stored on disk, when it was captured, and a SHA256 hash for deduplication.

The table has six fields. The id is an auto-incrementing primary key. The filename field stores the image's filename, perhaps "img_20251116_155230.jpg" for an image captured on November 16, 2025 at 15:52:30. The filepath field stores the full path where the image is stored on disk, such as "/home/pi/leafsense/images/img_20251116_155230.jpg". The captured_at timestamp records when the image was taken, defaulting to the current time if not explicitly provided. The image_hash field stores a SHA256 hash of the image file, enabling deduplication — if the same image is captured twice (perhaps due to a system glitch), the hash allows the system to detect and skip the duplicate. The uploaded_at timestamp records when the image metadata was stored in the database, which may be slightly after the capture time due to processing delays.

An index on the captured_at column enables fast queries like "Get me all images from the last 24 hours" or "Find the 10 most recent images". This is important for the UI, which may display recent images for the user to review or for manually triggering ML analysis.

All images implicitly belong to the single plant (id = 1). Like sensor readings and logs, there is no plant_id column in PLANT_IMAGES, because the system design constrains all images to be of the same plant.

The relationship between PLANT_IMAGES and ML_PREDICTIONS is the first where we see an explicit foreign key constraint. This is because the machine learning component needs strict referential integrity. When the ONNX model analyzes an image, it creates a prediction record that references the image. The database enforces that a prediction can only exist if a valid image exists. If for some reason the image record is deleted, the database automatically cascades the deletion to all predictions for that image. This prevents orphaned prediction records that reference non-existent images.

### The HEALTH_ASSESSMENTS Table: Daily Plant Health Evaluation

The HEALTH_ASSESSMENTS table stores the overall plant health evaluation generated during daily machine learning analysis. While the ML_PREDICTIONS table captures specific detections — "Powdery Mildew detected with 92% confidence" or "Nitrogen deficiency detected with 78% confidence" — the HEALTH_ASSESSMENTS table synthesizes all these individual problems into a single holistic health score from 0 to 100. This score represents the plant's overall condition, accounting for all detected diseases, deficiencies, environmental sensor deviations, and health trends. A score of 95 indicates a thriving plant with no concerning issues. A score of 45 indicates a plant in distress requiring immediate intervention.

The table has six fields. The id is an auto-incrementing primary key. The image_id is a foreign key referencing the plant_images table, linking each assessment to the specific image that prompted the analysis. The health_score field stores a real number from 0.0 to 100.0, calculated by the machine learning system based on multiple factors: each detected disease reduces the score proportionally to its confidence level and severity, each detected deficiency similarly reduces the score, deviations in environmental sensor readings from ideal ranges apply additional penalties, and trend information affects the assessment (a plant whose health declined from 90 to 85 in one day receives a larger penalty than a plant whose health improved from 75 to 80). The health_status field categorizes the score into one of four text values: "Excellent" for scores 90-100, "Healthy" for scores 75-89, "Warning" for scores 60-74, and "Critical" for scores 0-59. These categorical values map to dashboard colors — green for Excellent, light green for Healthy, yellow for Warning, and red for Critical — making the plant's condition immediately apparent to users.

The assessment_date timestamp records when the health assessment was made, defaulting to the current time. An index on this column enables fast queries for trend analysis, such as "Get health assessments from the last 30 days" or "Show me the health trend over the past year". The assessment_details field is an optional text field providing explanation of why the health_score is what it is. For a plant with excellent health, the details might read "No diseases detected, all sensors optimal, plant showing healthy

growth pattern.". For a plant in warning status, it might read "Powdery Mildew detected (92%) on leaves 3-5, Nitrogen deficiency (74%), pH trending high (8.1 vs target 6.5-7.5), requires treatment and environmental adjustment.".

Every health assessment is tied to a specific image through the image_id foreign key. When the ONNX models analyze an image, they produce ML_PREDICTIONS records for individual detections and a HEALTH_ASSESSMENTS record for the overall evaluation. The foreign key constraint ensures that a health assessment cannot exist without a valid image. If an image is deleted from the database, its health assessment cascades delete automatically, maintaining referential integrity.

All health assessments implicitly belong to the single plant (id = 1). Like other tables in LeafSense, there is no plant_id column in HEALTH_ASSESSMENTS, because the system design constrains all assessments to represent the same plant. The assessment is a crucial component of the dashboard: users see a prominent health card displaying today's score, status, and explanation. Over time, the accumulated assessments enable users to see health trends — whether their plant is improving, declining, or stable — and enable the system to generate trend-based alerts such as "Health declined 15 points in 3 days — take action now.".

**The ML_PREDICTIONS Table: Machine Learning Model Output**

The ML_PREDICTIONS table stores the output from ONNX Runtime when it analyzes a plant image. This is where the system records what the AI model thinks it sees: a disease, a nutrient deficiency, or a healthy plant. The table captures not just what was detected, but how confident the model is and what part of the image was affected.

The table has eight fields. The id is an auto-incrementing primary key. The image_id field is a foreign key reference to PLANT_IMAGES — it tells the database which image this prediction is based on. The prediction_type field categorizes the prediction: "Disease" for a plant disease, "Deficiency" for a nutrient deficiency, or "Healthy" if the model detected no problems. The prediction_label field is more specific, such as "Powdery Mildew" for a disease type or "Nitrogen Deficiency" for a nutrient issue. The confidence field stores a decimal number from 0.0 to 1.0 representing the model's certainty — 0.92 means the model is 92% confident in its prediction.

The bounding_box field stores a JSON object indicating where in the image the detected problem appears. For example, if Powdery Mildew is detected on the upper left portion of the leaf, the bounding box might be "x": 100, "y": 150, "width": 50, "height": 60, giving pixel coordinates that the UI can use to draw a highlighted rectangle over the affected area. The predicted_at timestamp records when the

prediction was made, and the model_version field stores which version of the ONNX model generated the prediction, enabling tracking of model changes and performance over time.

The table has three indexes. An index on image_id enables fast lookups like "Get all predictions for image 42". An index on confidence enables filtering high-confidence predictions like "Show me all predictions where confidence > 0.8". An index on prediction_type enables queries grouped by disease, deficiency, or healthy predictions.

The foreign key constraint on image_id enforces referential integrity. If someone tries to insert a prediction for image_id = 999 when no such image exists, the database rejects the insert with an error. If an image is deleted from PLANT_IMAGES, all its predictions are automatically deleted from ML_PREDICTIONS due to the ON DELETE CASCADE clause.

### The ML_DETECTIONS Table: Verification and Action Tracking

After the ONNX model makes a prediction, something needs to happen. The ML_DETECTIONS table tracks what happens next. When a user sees a prediction in the UI, they have three choices: verify that the prediction is correct and document the action taken, mark the prediction as incorrect so the model can learn from the mistake, or leave it unverified for later review.

The ML_DETECTIONS table has nine fields. The id is an auto-incrementing primary key. The prediction_id field is a foreign key reference to ML_PREDICTIONS, linking this detection log to a specific prediction. The is_verified field is a boolean (0 or 1) indicating whether the user has confirmed this prediction. The actual_label field stores what the user says the actual problem is — this might match the ML prediction (user confirms "Yes, it is Powdery Mildew") or differ from it (user says "No, this is actually Leaf Spot").

The confidence_correct field is a boolean (0 or 1) used for model evaluation. If the ML prediction was correct, the user marks this as 1. If the prediction was wrong, they mark it as 0. Over time, tracking these values allows calculation of the model's accuracy: "Of all predictions the user has verified, what percentage were actually correct?" The treatment_applied field documents what was done about the problem: "Sulfur spray applied", "Adjusted nitrogen dosage", or "No action taken". The notes field allows additional observations: "Leaf 3-5 severely affected" or "Symptoms improved after treatment".

The verified_at timestamp records when the user verified the prediction. Finally, the action_logged field is a boolean (0 or 1) indicating whether this detection has been automatically logged to the LOGS table yet.

An index on is_verified enables fast queries like "Get me all unverified predictions awaiting user

action". An index on verified_at enables time-based filtering like "Get me all verified predictions from the last week".

The foreign key constraint on prediction_id enforces that a detection record can only exist if a valid prediction exists. If a prediction is deleted, all its detection records are cascaded deleted automatically.

The relationship between ML_DETECTIONS and LOGS is unique in that it's not enforced by the database through foreign keys, but rather through application logic. When a user verifies a prediction, the application calls a method that does two things: it inserts a record into ML_DETECTIONS to track the verification, and it inserts a corresponding record into LOGS to document the event. This way, the user's review and actions on ML predictions flow automatically into the unified log history.

### The ML_RECOMMENDATIONS Table: System-Generated Suggestions with Outcome Tracking

The ML_RECOMMENDATIONS table stores system-generated recommendations tied to each machine learning prediction, creating a complete record of what the system suggests and what the outcomes of user actions are. When the ONNX models detect Powdery Mildew with 92% confidence, the system generates a recommendation such as "Apply Sulfur spray or horticultural oil to affected leaves, spray both upper and lower surfaces, repeat in 7 days if symptoms persist." When nitrogen deficiency is detected with 78% confidence, the system recommends "Increase nitrogen fertilizer dosage by 20%, test after 3 days, monitor new growth for improvement." These recommendations are stored with full lifecycle tracking from generation through user action through outcome.

The table has ten fields. The id is an auto-incrementing primary key. The prediction_id is a foreign key referencing the ML_PREDICTIONS table, linking each recommendation to the specific prediction that prompted it. The recommendation_type field stores one of five text values categorizing the type of suggestion: "Disease Treatment" for treating detected diseases, "Deficiency Adjustment" for nutrient adjustments, "Environmental Control" for sensor parameter adjustments, "Monitoring Alert" for recommendations to monitor without immediate action, or "Preventive Measure" for proactive prevention. The recommendation_text field contains the actual human-readable suggestion the user sees on their dashboard, generated intelligently based on what was detected, the plant's current conditions, and the severity of the issue.

The confidence field stores a real number from 0.0 to 1.0 representing the system's confidence that this recommendation will help resolve the problem. This differs from the prediction's confidence — which represents how sure the model is about the detection — and can be adjusted over time based on

historical outcome data. The generated_at timestamp records when the recommendation was created by the system, typically within milliseconds of the prediction being made. The user_acknowledged field is an integer (0 or 1) tracking whether the user has seen the recommendation, starting at 0 when generated and updating to 1 when the user reviews it on the dashboard. An index on this field enables fast "Show me new recommendations" queries for displaying notification counts.

The action_taken field is an optional text field recording what the user actually did in response to the recommendation: "Applied Sulfur spray to leaves 3-5 at 5:30 PM" or "Increased nitrogen dosage by 20%, added 2L nutrient solution" or "Improved air circulation by positioning fan toward plant." The action_date field stores the timestamp of when the user took that action. The outcome field is an optional text field recording the result of the action: "Excellent - Powdery Mildew spots disappeared after 3 days" or "Good - Yellowing stopped after 3 days" or "Partial - Some improvement but not complete" or "No change - Problem persisted despite treatment." The outcome_date field records when the outcome was observed, which may be days after the action was taken.

Every recommendation is tied to a specific prediction through the prediction_id foreign key. The foreign key constraint ensures that a recommendation cannot exist without a valid prediction. If a prediction is deleted from the database, all its recommendations cascade delete automatically. The accumulated outcome data enables the system to learn and improve. The system can calculate success rates by recommendation type: "Of all Sulfur spray recommendations for Powdery Mildew, 42 succeeded and 8 failed = 84% success rate." It can identify patterns: "Disease treatment recommendations succeed 95% of the time while preventive recommendations succeed only 60% of the time." Using this data, the system can adjust recommendation confidence scores, refine recommendation text, prioritize which recommendations to make first, and demonstrate its effectiveness to users.

All recommendations implicitly belong to the single plant (id = 1). Like other components of the machine learning pipeline, there is no plant_id column in ML_RECOMMENDATIONS, because the system design constrains all recommendations to address the same plant. The ML_RECOMMENDATIONS table is essential because it completes the feedback loop for the entire system: the ML models detect problems, the system suggests actions, users follow recommendations and record outcomes, and the system analyzes outcomes to improve future recommendations. Without this table, recommendations would exist only in the moment and provide no data for learning. With it, the system creates a closed-loop learning system where real-world results continuously inform and improve future guidance.

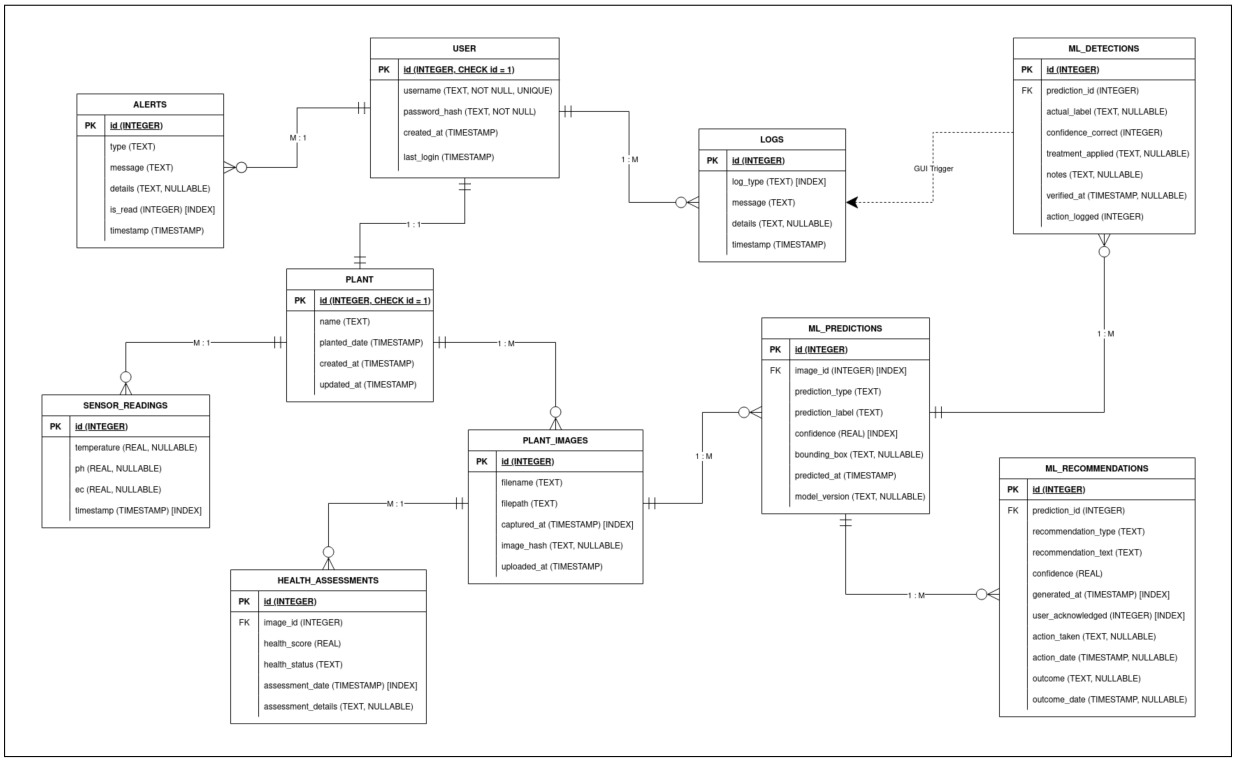



Figure 16: LeafSense Entity-Relationship Diagram

While the ERD shows the ten physical tables, the database implementation includes four database views that optimize data access for the API and dashboard. Views are virtual tables based on saved queries — they don't store data physically but provide convenient, pre-filtered data retrieval.

- **vw_latest_sensor_reading:** Returns the most recent sensor reading (temperature, pH, EC) for dashboard display.
- **vw_unread_alerts:** Filters alerts where is_read = 0, used for notification badges and alert panels.
- **vw_daily_sensor_summary:** Aggregates daily sensor statistics (averages, minimums, maximums) for analytics and charts.
- **vw_pending_recommendations:** Shows unacknowledged recommendations from ML models, used for the "Actions Needed" panel.

### 2.6.4 State Chart

State machines are structured models that define and represent a system's behavior by outlining a set of distinct states and the transitions between them. Widely used in computer science, engineering, and other disciplines, state machines help simulate sequential operations, manage control logic, perform parsing tasks, and model a variety of system behaviors.

The LeafSense State Chart (Figure 17) represents the complete operational flow of the real-time hydroponic monitoring system, illustrating how the system transitions between different operational states to maintain optimal plant growing conditions through automated monitoring, analysis, and control.

The system begins in System Configuration where hardware components initialize and database connections are established. After successful setup, the system transitions to the Idle state, which serves as the central waiting point for user operation.

From idle, the system follows two main operational paths: Power Off and User Input (where the user interacts with the touch display). In the case of a User's Input being recognized it can be Processed as doing three different actions: Updating the Ideal Conditions, Viewing Alerts or Viewing the Dashboard.

Starting with Updating the Ideal Conditions, the first step is to save the user-inputted parameters, then requesting an all encompassing sensor measurement. These two data will be compared which will result either in Bad Values (sensor readings not in the ideal ranges) or Good Values. Bad Values will enter a loop of activating the actuators accordingly to correct them, then sensing and comparing until the data is acceptable. After being considered Good the actuators are told to stop and to return to idle. This way a change in Ideal Parameters are immediately acted upon not necessitating a wait for the periodic sensor readings.

Viewing Alerts is acted upon by Displaying said Alerts, with the caveat that if there are none the system is too return to idle. In that screen the user can choose an Alert to view its details, then acknowledging it which will update the alert appropriately.

Viewing the Dashboard does exactly that: opens a screen with the current sensor readings and ideal parameters. The user can exit this dashboard anytime and the system will return to idle.

Powering Off not only happens when the user says so but also in case of an emergency. The protocol for this should be to save all currently unsaved data first.

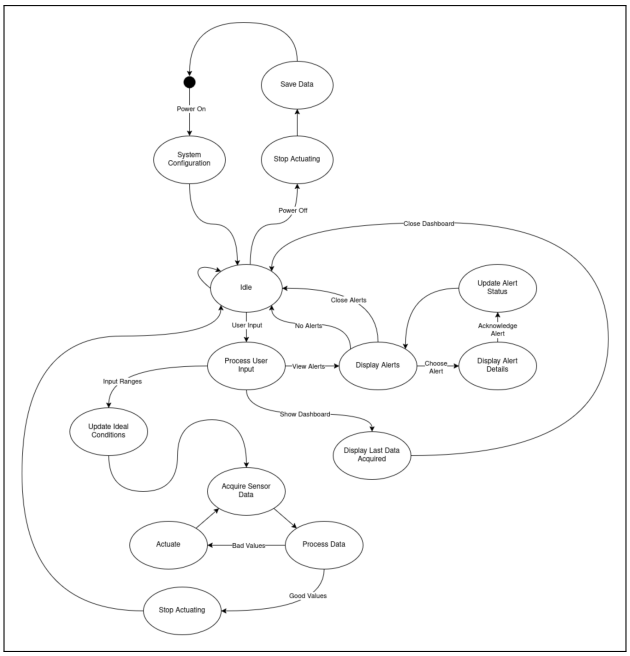



Figure 17: LeafSense State Chart

### 2.6.5 Sequence Diagrams

The sequence diagrams provide a detailed visualization of the system's operation by mapping the interactions between different objects over time. They illustrate the order of events and show how messages are exchanged among system components to accomplish specific tasks. By presenting the flow of operations step by step, the diagrams help in understanding the system's dynamic behavior, clarifying the roles of individual objects, and revealing dependencies between various processes. This comprehensive overview enhances communication among team members and facilitates the identification of potential improvements or issues within the system's workflow.

The LeafSense system operates through a sophisticated integration of automated monitoring, intelligent analysis, and user oversight capabilities. The sequence diagrams presented in this section capture essential operational flows that collectively represent the complete system behavior from initial configuration through ongoing intelligent plant cultivation management.

#### Timer-Triggered Monitoring Sequence

The Timer-Triggered Monitoring Sequence represents the core automated functionality of the LeafSense system, operating on a periodic cycle to ensure continuous environmental oversight. The sequence begins when the Timer actor sends a timed signal to the Raspberry Pi, initiating the automated monitoring

cycle. The Raspberry Pi immediately responds by triggering the Sensors to collect current environmental measurements including pH levels, electrical conductivity, and water temperature.

Upon receiving sensor readings, the Raspberry Pi processes these data internally through its Compare function, evaluating current measurements against predefined optimal ranges stored in the system configuration. The processed readings are simultaneously transmitted to the Display for real-time visualization, enabling immediate system status awareness. When the comparison reveals environmental parameters exceeding acceptable thresholds, the system enters a conditional branch marked "If Needed," where the Raspberry Pi triggers the Actuators to implement corrective measures.

The actuator activation initiates a continuous feedback loop where the Raspberry Pi maintains ongoing communication with the Actuators through continuous reading triggers and response transmissions. This sustained interaction ensures that environmental corrections are properly implemented and monitored until the parameters return to optimal ranges. The sequence concludes with actuator deactivation once environmental conditions stabilize, and the Display receives updated readings reflecting the corrected values.

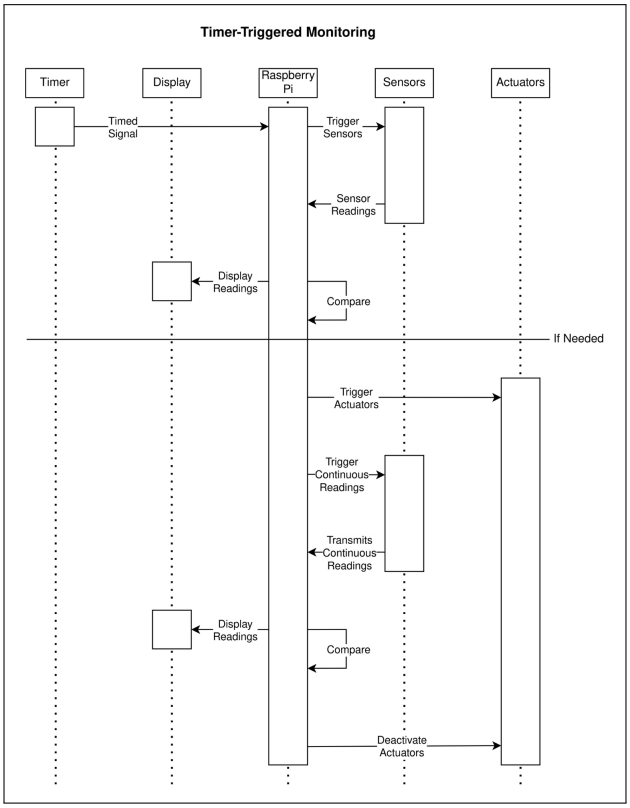



Figure 18: LeafSense Timer-Triggered Monitoring Sequence Diagram

**Daily ML Analysis & Plant Health Assessment Sequence**

The Daily ML Analysis Sequence represents the most sophisticated component of the LeafSense system, integrating computer vision technology with comprehensive plant health assessment capabilities that extend far beyond traditional environmental monitoring approaches. This advanced workflow orchestrates multiple artificial intelligence components to deliver intelligent health monitoring through a sophisticated diagnostic system that combines automated image analysis with expert agricultural knowledge, creating actionable insights for optimal plant care.

The sequence begins with a daily trigger from the Timer actor, initiating the automated plant photography and analysis cycle that provides intelligent health monitoring beyond basic environmental control. This scheduled approach ensures consistent monitoring intervals while supporting flexible scheduling that can accommodate different analysis frequencies for various plant types and growth phases, with the timer integration allowing system adaptation based on plant development stages and environmental conditions.

Upon receiving the daily trigger, the Raspberry Pi activates the Camera to capture high-resolution plant images that serve as input data for the machine learning analysis pipeline. The camera system employs advanced OpenCV preprocessing to optimize image quality through automated exposure adjustment, color balance correction, and noise reduction algorithms that ensure consistent image quality regardless of ambient lighting conditions. The system captures multiple images from different angles to provide comprehensive visual coverage of each plant, with captured images undergoing immediate quality assessment using computer vision algorithms that evaluate focus sharpness, lighting adequacy, color saturation, and overall image clarity.

The captured images are immediately stored in the Database with comprehensive metadata including capture timestamps, camera settings, quality scores, environmental conditions at time of capture, and plant growth stage information, creating a permanent visual record of plant development while ensuring that image data is preserved for historical analysis and system improvement purposes. This metadata enables correlation analysis between image characteristics and environmental factors, supporting system optimization and troubleshooting.

The Raspberry Pi then transmits the captured images to the ML Engine, which begins the sophisticated analysis process by retrieving reference libraries from the Database. These reference libraries contain extensive collections of plant images representing healthy conditions, various disease manifestations, and nutrient deficiency symptoms that enable accurate comparative analysis. The reference libraries incorporate expert-validated examples from agricultural specialists, research institutions, and field observations, creating a comprehensive knowledge base that combines scientific understanding with practical

growing experience.

The ML Engine processes the current plant images against these reference datasets, employing multiple computer vision algorithms including convolutional neural networks for pattern recognition, feature extraction algorithms for detailed characteristic analysis, and similarity matching systems for comparative assessment. The system utilizes ensemble learning approaches that combine predictions from multiple specialized models, each trained to identify specific types of health issues including fungal diseases, bacterial infections, viral symptoms, and various nutrient deficiency patterns. Advanced feature extraction techniques analyze color patterns, texture characteristics, leaf shape variations, and growth abnormalities that indicate different health conditions through sophisticated image segmentation that isolates individual leaves and plant components for detailed analysis.

Following the completion of machine learning analysis, the ML Engine returns detailed analysis results to the Raspberry Pi, including confidence scores for various health assessments and specific identification of any detected issues. The analysis results include comprehensive health categorization with probability distributions across multiple potential conditions, enabling nuanced assessment that accounts for uncertainty and provides multiple diagnostic possibilities when symptoms might indicate several potential issues. The confidence scoring system incorporates multiple validation mechanisms including cross-validation against reference libraries, consistency checking across multiple analysis models, and temporal analysis that considers previous health assessments to identify concerning trends or sudden changes.

The Raspberry Pi stores these comprehensive ML results in the Database with complete analysis details including model versions used, processing times, confidence metrics, and supporting evidence from reference library matches. The system proceeds to generate intelligent recommendations based on the analysis findings, utilizing decision trees and expert system logic to translate health assessments into actionable care instructions that may include specific treatment protocols such as targeted nutrient adjustments, environmental modification suggestions, disease treatment procedures, or preventive measures tailored to the identified conditions and plant growth stages.

The sequence includes a critical decision point where the system evaluates whether the analysis results indicate conditions requiring immediate user attention through sophisticated alert classification algorithms that consider issue severity, progression speed, treatment urgency, and potential impact on plant survival or productivity. When critical issues are detected, such as rapid disease progression, severe nutrient deficiencies, or environmental stress indicators, the Raspberry Pi creates alerts in the Database and transmits urgent notifications through the Display, ensuring that severe health problems receive prompt user response.

The Display shows both the detailed analysis results and any critical alerts, providing comprehensive information for informed decision-making through an intuitive interface that presents complex analytical information in accessible formats. The display system includes visual indicators for health status, trend analysis graphs showing health progression over time, and interactive elements that enable users to access detailed diagnostic information and treatment recommendations. Advanced visualization capabilities include side-by-side comparisons with reference images, annotated displays highlighting detected symptoms, and progress tracking that shows improvement or deterioration following treatments.

The Daily ML Analysis Sequence incorporates continuous learning mechanisms that improve system performance over time through outcome tracking, user feedback integration, and model refinement based on accumulated experience. Treatment outcome data feeds back into the machine learning models, enabling continuous improvement of diagnostic accuracy and recommendation effectiveness, while comprehensive performance metrics support systematic optimization of analytical algorithms and decision-making processes. This sophisticated integration of artificial intelligence, expert knowledge, and automated control systems represents a significant advancement in hydroponic technology, providing farmers and researchers with powerful tools for optimizing plant care and maximizing productivity.

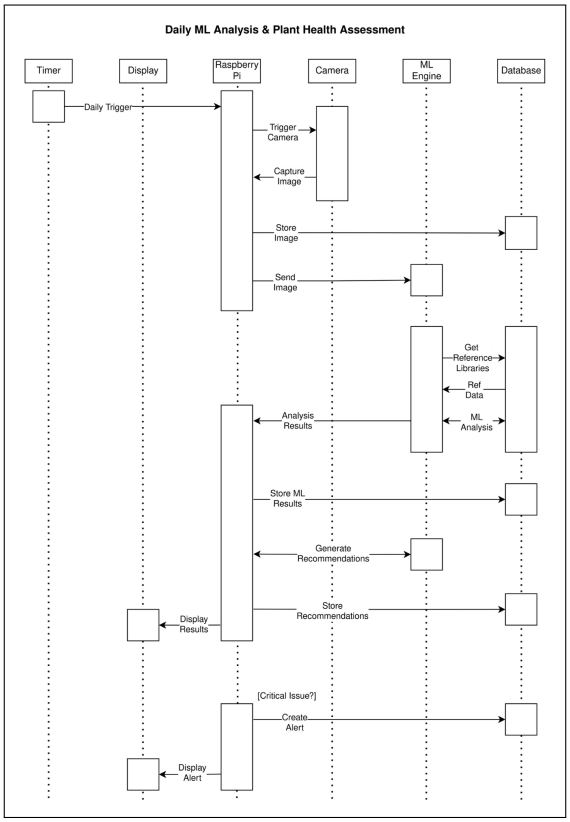



Figure 19: LeafSense Daily ML Analysis & Plant Health Assessment Sequence Diagram

## 2.7 Estimated Budget

We're a small team of two students, and we're working hard to keep our project costs as low as possible while still achieving our goals. This budget includes only the essential expenses — we're focused on getting what we truly need and finding the best prices. Since we're working with limited student funds, we want to make sure every euro is spent wisely. This budget will help us make thoughtful decisions and stay organized as we move through the project.

| **Item** | **Quantity** | **Price (EUR)** |
|---|---|---|
| Raspberry Pi 4 B | 1 | 66 |
| Micro SD Card | 1 | 6 |
| Power Supply | 1 | 6 |
| Touchscreen | 1 | ≈ 20 |
| Camera | 1 | ≈ 24 |
| pH Sensor | 1 | ≈ 15 |
| TDS Sensor | 1 | ≈ 15 |
| Water Temperature Sensor | 1 | ≈ 8 |
| Water Heater | 1 | ≈ 10 |
| Peristaltic Pump | 3 | ≈ 9 |
| Real Time Clock Module | 1 | ≈ 8 |
| ADC Module | 1 | ≈ 8 |
| Relay Module | 1 | ≈ 4 |
| Cover Structure | Not Applicable | ≈ 10 |
| Prototype Testing Environment | Not Applicable | ≈ 30 |
| **Total Estimated Cost (EUR)** | ≈ 239 | |

Table 2.2: LeafSense Estimated Budget

# Chapter 3

# Foundational Theory

Embedded Systems and Computer Science are built on core theories and principles that guide the design, optimization, and implementation of digital systems. These foundational concepts enable the reliable operation of complex, resource-constrained systems, many of which must function seamlessly in real-time. This chapter introduces and explores these essential topics — covering Embedded Systems, Processes, Threads, Communication Technologies, Databases, Device Drivers, Signals, and Daemons— each playing a vital role in the field and its practical applications. By studying these subjects, we gain the insights needed to approach our project with a clearer understanding of what we must accomplish and how best to achieve it.

## 3.1 Embedded Systems

Embedded Systems are specialized computing systems designed to perform specific tasks within a larger device or system.

### 3.1.1 Embedded Systems vs. General-Purpose Computers

Unlike General-Purpose Computers, which can run various applications, Embedded Systems are dedicated to a particular function and are often optimized for efficiency, reliability, and speed. They typically consist of both hardware and software tailored for this dedicated role and operate under resource constraints, such as limited memory or processing power.

Embedded Systems typically offer limited programmability for the end user. They operate under strict, fixed run-time requirements designed for their specific function. In contrast, General-Purpose Computers provide users with significant programmability and flexibility, allowing them to install and modify software without the constraints of time-critical operations. This key difference influences how each system is used and adapted for various applications, from household gadgets to advanced industrial solutions.

## 3.2 Processes

A Process is a core operating system concept representing a running instance of a program. It includes the program's code, its current state, and the resources required for execution. Understanding processes is essential for grasping how modern operating systems manage multiple tasks simultaneously, enabling efficient computing. This concept explains the background work that allows devices to run smoothly while users browse the web, edit documents, or stream video.

### 3.2.1 Life-Cycle

Processes experience several states throughout their life cycle:

- **New:** The initial stage where the process is being created. The operating system allocates necessary resources (e.g., memory, process ID);
- **Ready:** The process is loaded in memory and is waiting to be assigned to a CPU by the scheduler;
- **Running:** The process's instructions are actively being executed by a CPU;
- **Waiting (or Blocked):** The process is paused, waiting for an event to occur (such as I/O completion or receiving a signal) before it can return to the ready state;
- **Terminated:** The process has finished execution. The operating system reclaims all allocated resources.

### 3.2.2 IPC (Inter-Process Communication)

Inter-Process Communication (IPC) is a set of techniques allowing processes to communicate and synchronize their actions. In multitasking operating systems, where multiple processes run concurrently, IPC is vital for coordinating operations, sharing data, and exchanging information.

#### Importance

IPC is essential for coordinating processes in a multitasking environment. It enables effective data sharing and synchronization, which promotes modular application design, enhances resource management, and helps prevent conflicts. Ultimately, IPC contributes to more efficient, responsive, and reliable systems.

**Mechanisms**

- **Pipes:** Creates a one-way (unidirectional) data channel, typically between related processes. Named pipes (FIFOs) allow communication between unrelated processes;
- **Message Queues:** Allow processes to exchange messages asynchronously. Senders and receivers do not need to be active simultaneously, which helps decouple communicating processes;
- **Shared Memory:** Allows multiple processes to access the same region of memory. This is a very fast communication method but requires external synchronization (like semaphores) to prevent race conditions;
- **Semaphores:** A synchronization primitive that acts as a counter to control access to shared resources, used to prevent race conditions and manage concurrent access;
- **Sockets:** Provide a communication endpoint, enabling processes to communicate both on the same machine and across different machines on a network (e.g., using TCP or UDP).

## 3.3 Threads

Threads are the smallest units of execution within a process, representing a series of instructions that can be independently scheduled and executed by the CPU. Unlike full processes, multiple threads within the same process share the same memory space and resources, allowing them to work closely together and perform tasks concurrently.

### 3.3.1 Features

- **Lightweight:** Threads are called "lightweight processes" as they use fewer resources. Threads within a process share memory and other resources, making them faster to create and manage than separate processes;
- **Shared Memory:** Threads within a process share the same address space, allowing them to access the same data. This simplifies communication but requires careful synchronization to avoid data conflicts;
- **Independent Execution:** Each thread has its own program counter and stack, allowing it to execute independently. This enables a single process to perform multiple tasks concurrently (e.g.,

a server handling multiple client requests).

### 3.3.2 Advantages

- **Concurrency:** Threads allow a process to perform multiple tasks simultaneously, improving performance, especially on multi-core processors where threads can run in parallel;
- **Efficient Resource Usage:** Since threads share resources, they are less resource-intensive than creating multiple processes, each requiring its own memory space;
- **Responsive Applications:** Threads can improve application responsiveness. Time-consuming tasks (like I/O) can run in a separate thread, allowing the main application thread (e.g., user interface) to remain interactive.

### 3.3.3 Life-Cycle

Threads move through several states, similar to processes:

- **New:** The thread is created but has not yet started execution;
- **Runnable (Ready):** The thread is ready to execute and waiting for the scheduler to assign it to a CPU;
- **Blocked (Waiting):** The thread is paused, waiting for an event (e.g., I/O completion, a lock to be released);
- **Terminated:** The thread has completed its execution and is no longer active.

### 3.3.4 Threads vs. Processes

Processes are independent execution environments, each with its own memory space. In contrast, Threads exist within a process and share its resources (like memory). This makes threads ideal for tasks requiring frequent communication but also necessitates careful synchronization to prevent data corruption. Threads enable concurrent execution of tasks within a single process, leading to more efficient and responsive applications.

**Multitasking Scheduling Models**

Multitasking Scheduling Models determine how multiple tasks share CPU time, balancing responsiveness, fairness, and resource management. This section overviews three common models: Preemptive, Round-Robin, and Round-Robin with Time Slice, detailing their advantages, drawbacks, and typical use cases.

- **Preemptive**

  In preemptive scheduling, the operating system can interrupt (preempt) a running task to execute another, often based on priority. This allows high-priority tasks to run immediately, keeping the system responsive to urgent needs.

  - **Advantages:** Quick response for priority tasks and balanced CPU use.
  - **Drawbacks:** Higher overhead and complexity.
  - **Best for:** Real-time systems, operating systems (like Windows or Linux), and environments with varied task priorities.

- **Round-Robin**

  Round-Robin scheduling cycles through tasks in a queue, giving each an equal opportunity to run. Tasks are processed in a fixed order. When the last task finishes its turn, the scheduler returns to the first, ensuring all tasks receive CPU time.

  - **Advantages:** Fairness and simplicity.
  - **Drawbacks:** Not great for urgent tasks and high switching overhead.
  - **Best for:** Time-sharing systems, environments with equal-priority tasks, or where fairness is more important than response time

- **Round-Robin with Time Slice (or Quantum)**

  This model refines Round-Robin by assigning a fixed time slice (or quantum) to each task. A task runs until it completes, blocks, or its time slice expires. If the time slice expires, the task is preempted and moved to the back of the queue. Adjusting the time slice length impacts system responsiveness and overhead.

  - **Advantages:** Adjustable timing and efficient resource use.
  - **Drawbacks:** Finding the right balance and limited real-time use.

- **Best for:** General-purpose systems, scenarios with tasks of varying lengths, and environments where a blend of fairness and responsiveness is needed.

Each model suits specific needs: Preemptive for responsive, priority-driven systems; Round-Robin for fairness; and Round-Robin with Time Slice for balancing efficiency with equitable CPU access.

### Thread Synchronization

Thread synchronization coordinates thread actions to ensure safe access and modification of shared resources. When multiple threads interact with shared data (variables, files, etc.), synchronization manages their access by protecting critical sections. This prevents race conditions and ensures data integrity.

There are numerous thread synchronization mechanisms, but here we will focus on a few key ones: Mutexes, Condition Variables, and Interlocked Variable Access.

- **Mutexes (Mutual Exclusion):** A mutex is a locking mechanism that allows only one thread to access a shared resource at any given time. When a thread locks a mutex, other threads that request the same resource are blocked until the mutex is released.
- **Condition Variables:** Condition variables work with mutexes to manage complex synchronization tasks. A thread can wait on a condition variable until a specific condition is met, allowing for more flexible coordination between threads.
- **Interlocked Variable Access:** Interlocked operations provide atomic access to shared variables, allowing threads to perform simple operations (like increments or comparisons) safely without needing to use locks. These operations are executed as a single, indivisible step by the processor, which prevents race conditions.

### Signals

In operating systems, signals serve as a vital tool for Inter-Process Communication (IPC) and managing threads. They act as asynchronous notifications that alert processes or threads to events or conditions needing their attention, facilitating responsive and efficient multitasking. However, the application and management of signals vary significantly between processes and threads.

- **Signals in Processes**

  Signals in the context of processes are asynchronous alerts dispatched to a process to inform it

about events like interrupts, termination requests, or specific conditions such as illegal memory access. Some common signals include:

- **SIGINT:** An interrupt signal, typically generated by the user (like pressing Ctrl+C) to request an interruption;
- **SIGKILL:** A signal that forcibly terminates a process without allowing it to clean up or handle the termination;
- **SIGTERM:** A request for a process to terminate gracefully, allowing it to perform cleanup operations;
- **SIGALRM:** A signal generated when a timer set by the alarm() function expires, useful for handling timeouts in applications;
- **SIGSTOP:** A signal that stops (pauses) a process's execution. This signal cannot be caught, blocked, or ignored, making it a reliable way to halt a process;
- **SIGCONT:** A signal used to continue a process that has been stopped by a SIGSTOP signal or other means, allowing it to resume execution.

A process can define a signal handler, a specific function that dictates its response to a signal. When a signal is received, the OS interrupts the process's execution and triggers the handler. A process can ignore the signal, execute its custom handler, or perform the default action (e.g., termination). Signals are typically directed at the entire process, and each signal has a default action unless a custom handler is defined.

- **Signals in Threads**

  In multithreaded applications, signal handling is more complex. While signals are sent to the entire process, individual threads can set signal masks to block specific signals. When an unblocked signal is sent to a process, the kernel delivers it to just one of the threads that has not blocked it. This allows for dedicated threads to handle signals while others continue executing. Some systems also allow signals to be directed to a specific thread. Managing this requires careful coordination to ensure signals are processed correctly without disrupting other threads.

## 3.4 Communication Technologies

In today's embedded systems and electronics, seamless communication between devices is essential for efficient and coordinated operation. Communication technologies enable devices to exchange data, synchronize activities, and work together to achieve shared objectives. This section explores the fundamental communication protocols, each designed for particular applications and environments, that form the backbone of embedded system design and implementation. Understanding these protocols is crucial for building systems that operate reliably and efficiently in real-world conditions.

Specifically, protocols like SPI and I2C serve distinct roles in facilitating data exchange and device coordination within embedded systems.

### 3.4.1 SPI

SPI (Serial Peripheral Interface) is a widely-used synchronous serial communication protocol designed for short-distance communication within embedded systems. Originally developed by Motorola, SPI enables high-speed data transfer between a master device and one or more slave devices. Its speed and efficiency make it particularly suitable for applications that require quick data exchange, such as connecting microcontrollers to sensors, memory devices, displays, and various peripherals.

#### Principle of Operation

SPI utilizes a master-slave architecture, where a single master device controls one or more slave devices through four main lines. The MOSI (Master Out Slave In) line transmits data from the master to the slave, while the MISO (Master In Slave Out) line carries data from the slave back to the master. The SCLK (Serial Clock) line, generated by the master, synchronizes data transmission; slave devices use this clock to determine the appropriate moments for reading or sending data. Lastly, the SS/CS (Slave Select/Chip Select) line enables the master to specify which slave device it wants to communicate with; when the SS line for a specific slave is activated — usually by pulling it low — that slave is enabled for communication.

#### Features

- **Full-Duplex Communication:** SPI supports simultaneous data transmission and reception, allowing for full-duplex communication, which enhances speed and efficiency;
- **High Speed:** SPI can achieve data rates significantly higher than many other communication

protocols, making it ideal for applications demanding rapid data exchange;

- **Flexible Configuration:** The protocol supports various data formats and clock polarities/edges, providing flexibility in how data is transmitted and received;
- **Multiple Slave Devices:** By utilizing additional SS lines, a single master device can manage multiple slave devices, making SPI versatile for complex system designs.

**Advantages**

- **High Data Rate:** SPI can operate at data rates reaching several megabits per second, making it considerably faster than protocols like I2C;
- **Simplicity:** The protocol is straightforward, easy to implement, and incurs minimal overhead, allowing for quick integration into projects;
- **Full-Duplex Communication:** The ability to send and receive data simultaneously significantly boosts overall communication efficiency.

**Disadvantages**

- **More Wires Required:** Unlike I2C, which uses fewer wires, SPI requires more connections, especially when managing multiple slave devices (each needing its own SS line);
- **No Standardized Protocol:** Although SPI is widely adopted, it lacks an official standard, leading to variations in implementation that can cause compatibility issues between different devices;
- **Limited Distance:** SPI is best suited for short-distance communication, typically within the same device or across a PCB, as signal integrity can diminish over longer distances.

**Common Applications**

- **Sensor Interfaces:** SPI is frequently employed to connect microcontrollers with high-speed sensors, such as accelerometers, gyroscopes, and temperature sensors;
- **Memory Devices:** Flash memory and EEPROM devices often utilize SPI for rapid data access and storage;
- **Display Modules:** SPI is commonly used to interface with graphic displays, OLED screens, and other visual output devices due to its high-speed capabilities;

- **Analog-to-Digital Converters (ADCs):** Many ADCs utilize SPI for swift data transfer, enabling real-time data acquisition in applications like audio processing or sensor data collection.

### 3.4.2 I2C

I2C, short for Inter-Integrated Circuit and also referred to as TWI (Two Wire Interface), is a synchronous communication protocol designed for multi-master and multi-slave configurations. Developed by Philips Semiconductors (now part of NXP Semiconductors), I2C is widely utilized in embedded systems to connect low-speed peripherals to microcontrollers and processors. Its appeal lies in its simplicity, efficiency, and the ability to connect multiple devices using just two wires, making it an excellent choice for various applications.

#### Principle of Operation

I2C communication operates using two main lines: SDA (Serial Data Line) for carrying data and SCL (Serial Clock Line) for synchronization. The master device generates the SCL clock, which dictates when data is transferred on the SDA line, ensuring both master and slave devices are synchronized. This two-wire interaction allows for efficient data exchange between multiple devices on the same bus.

#### Features

- **Multi-Master and Multi-Slave Architecture:** I2C allows several master devices to control multiple slave devices on the same bus, making it highly adaptable for complex systems that require communication among various components;
- **Addressing:** Each device on the I2C bus is assigned a unique address, enabling the master to specify which slave device to communicate with. Addresses can be either 7 or 10 bits long, supporting a broad range of devices on the same bus;
- **Packet-Based Communication:** Data is transmitted in packets that include a start condition, device address, data bytes, and a stop condition, ensuring reliable and organized communication;
- **Simple Wiring:** By utilizing just two wires (SDA and SCL) along with a ground connection, I2C minimizes wiring complexity, making it particularly advantageous for applications with space constraints.

**Advantages**

- **Simplicity:** The straightforward two-wire connection simplifies the implementation process and reduces circuit design complexity;
- **Support for Multiple Devices:** I2C's ability to accommodate multiple masters and slaves on the same bus allows for a flexible architecture in embedded systems;
- **Built-in Acknowledgment:** Each byte transmitted must be acknowledged by the receiving device, enhancing the integrity of the data being exchanged.

**Disadvantages**

- **Speed Limitations:** I2C operates at lower speeds compared to other protocols like SPI, with standard speeds of 100 kbit/s (standard mode), 400 kbit/s (fast mode), and up to 3.4 Mbit/s (high-speed mode). While these speeds are adequate for many applications, they may fall short for high-speed data transfer needs;
- **Limited Bus Length:** I2C is best suited for short-distance communication, typically limited to a few meters. Over longer distances, signal degradation can occur, affecting the reliability of communication;
- **Complexity with Multiple Masters:** Although I2C supports multiple masters, managing bus arbitration can introduce design complexity during implementation.

**Common Applications**

- **Sensor Interfacing:** I2C is commonly employed to connect sensors, including temperature sensors, accelerometers, and humidity sensors, to microcontrollers;
- **Memory Devices:** Many EEPROM and Flash memory devices use I2C for efficient data storage and retrieval, making it ideal for non-volatile memory applications;
- **Displays and User Interfaces:** I2C is frequently used to connect LCDs and OLED displays to microcontrollers, facilitating easy control of visual output;
- **Real-Time Clocks (RTCs):** Numerous real-time clock modules utilize I2C for communication, enabling accurate timekeeping within embedded systems.

## 3.5 Databases

A database is a systematically organized collection of data or information, usually stored electronically within a computer system. These databases are overseen by Database Management Systems (DBMS), which enable users to create, read, update, and delete data efficiently. They are essential in a wide range of applications, from simple web services to complex enterprise-level solutions, as they facilitate effective data management, retrieval, and manipulation.

Databases arrange data in a structured format, typically utilizing tables made up of rows and columns. Each table represents a distinct type of entity, with individual rows corresponding to unique records and columns representing the entity's attributes. For instance, in a library database, a table titled "Books" might include columns for Title, Author, ISBN, and Published Year, where each row denotes a specific book entry.

### Features

- **Structured Data Storage:** Databases organize information in a manner that simplifies retrieval and manipulation, allowing for efficient data queries;
- **Data Integrity:** Databases enforce rules to ensure accuracy and consistency of data. Mechanisms such as constraints, primary keys, and foreign keys help maintain valid relationships between data entities;
- **Transaction Management:** Most databases support transactions, enabling multiple operations to be treated as a single unit of work. This ensures that either all operations are successfully completed or none are, preserving data integrity;
- **Data Security:** Databases implement various security measures to safeguard data access. Common practices include user authentication, authorization, and data encryption to protect sensitive information.

### Types

- **Relational Databases:** These store data in tables and use Structured Query Language (SQL) for querying. Popular examples include MySQL, PostgreSQL, and Oracle Database;
- **NoSQL Databases:** Designed for unstructured or semi-structured data, these databases can

accommodate a range of data models, including document, key-value, graph, or column-family stores. Notable examples are MongoDB, Cassandra, and Redis;

- **In-Memory Databases:** These store data directly in the system's main memory (RAM) rather than on disk, allowing for faster access. Examples include Redis and Memcached;
- **Time-Series Databases:** Optimized for handling time-stamped data points, these databases are perfect for applications involving time-series data. Examples include InfluxDB and TimescaleDB;
- **Graph Databases:** Utilizing graph structures to represent data, these databases focus on relationships between entities. Examples include Neo4j and Amazon Neptune.

**Advantages**

- **Efficient Data Retrieval:** By employing indexing and query optimization techniques, databases can quickly locate and retrieve information;
- **Scalability:** Many databases can scale both vertically (adding resources to a single server) and horizontally (distributing data across multiple servers) to manage increased workloads effectively;
- **Data Sharing:** Databases enable multiple users to access and interact with the same data simultaneously, fostering collaboration across teams and applications.

**Drawbacks**

- **Complexity:** Designing, implementing, and managing a database can be intricate and often requires specialized skills and knowledge;
- **Cost:** Some database systems may entail significant expenses for setup and maintenance, especially when considering licensing fees, hardware costs, and administrative overhead;
- **Performance Overhead:** Although databases are optimized for data retrieval, complex queries, especially with large datasets, can introduce performance overhead.

**Common Applications**

- **Business Applications:** Databases play a crucial role in managing essential business data such as customer information, transactions, and inventory;

- **Web Applications:** Many websites depend on databases for storing user profiles, content management systems, and e-commerce information;
- **Data Analysis:** Databases facilitate data analysis and reporting, allowing organizations to derive insights from extensive data collections;
- **IoT Applications:** In the realm of the Internet of Things (IoT), databases are vital for storing data generated by connected devices, enabling effective monitoring and analytics.

## 3.6 I/O System

The Input/Output (I/O) System is a vital aspect of computer architecture, functioning as the communication link between a computer's central processing unit (CPU) and the various peripheral devices that interact with the external environment. This system is made up of an array of components and mechanisms that facilitate smooth communication, allowing for efficient data exchange and meaningful user interactions.

### 3.6.1 Components

The I/O system is composed of several key components that manage communication between the CPU and peripheral devices. This section reviews these components, including I/O devices, device drivers, the I/O subsystem, and the kernel, which collaborate to provide a unified method for accessing hardware.

#### I/O Devices

These physical components are essential for user interaction with the computer system and can be classified into four main categories:

- **Input Devices:** These allow users to feed data into the computer. Common examples include keyboards, mouses, scanners, and microphones, each designed to capture specific types of input;
- **Output Devices:** These components display or produce data for users or other systems. Monitors, printers, speakers, and projectors are typical examples that help convey information from the computer;
- **Storage Devices:** This category encompasses hard disk drives (HDDs), solid-state drives (SSDs), and USB flash drives, all of which enable data storage and retrieval, ensuring that information is

preserved for future use;

- **Network Interfaces:** Components like Ethernet cards and Wi-Fi adapters fall into this category, enabling communication with other computers and devices over various networks.

### Device Drivers

These are specialized software programs that serve as intermediaries between the operating system and hardware devices. Device drivers translate high-level commands from the operating system into specific instructions that hardware can understand. This allows for the effective management of diverse devices, enabling applications to interact with hardware without delving into the complexities of its operation. They can be divided into two main categories:

- **Character Device Drivers:** Manage devices that transfer data as a continuous stream of characters (e.g., keyboards, mice, serial ports). They handle I/O one character at a time, providing a simple interface for sequential data access;
- **Block Device Drivers:** Manage devices that read and write data in fixed-size blocks, allowing random access (e.g., hard drives, SSDs). They are crucial for storage devices and often use caching to improve performance.

### I/O Subsystem

The I/O subsystem includes all hardware and software components responsible for managing I/O operations. Key elements of this subsystem are:

- **I/O Controllers:** Specialized hardware units that manage data flow between the CPU and peripherals, handling tasks like data transfer, buffering, and error detection;
- **Buffering Mechanisms:** Buffers are temporary storage areas that hold data during transfer. They help reconcile speed differences between the CPU and I/O devices, preventing data loss and improving performance.

### Operating System's Kernel

The kernel serves as the heart of the operating system, acting as a bridge between the hardware and software components of a computer. It is essential for managing system resources effectively and

plays a crucial role in the functioning of the I/O system. One of its key responsibilities is to provide a standardized interface for accessing various I/O devices. This abstraction allows applications to communicate seamlessly with different hardware without needing in-depth knowledge of each device's specifics.

Additionally, the kernel is responsible for handling interrupts from I/O devices. When a device requires attention or has completed a task, it sends an interrupt signal to the kernel. This mechanism ensures that the system responds promptly to device needs, enhancing overall efficiency and performance.

Moreover, the kernel oversees resource allocation and conflict resolution. By managing how multiple applications access I/O devices, it ensures that these applications can operate simultaneously without interfering with one another. This careful coordination is vital for maintaining system stability and optimizing resource use, enabling a smooth user experience across various applications.

#### Applications

Applications are the end-user programs that rely on the I/O system for interaction with devices. By utilizing the uniform interface provided by the operating system, these applications can perform tasks like capturing user input, displaying output, and accessing stored data. This abstraction allows developers to concentrate on enhancing application functionality rather than grappling with the intricacies of hardware interactions.

## 3.7 Daemons

Daemons are background processes that run continuously on a system, typically starting at boot and operating without direct user input. They play an essential role in Unix-like systems, such as Linux and macOS, where they manage tasks like network requests, system services, and maintenance, often automatically and on a predefined schedule.

### 3.7.1 Features

- **Background Operation:** Daemons run independently in the background, detached from user sessions and often without a graphical interface, allowing them to perform functions continuously without direct interaction;
- **Persistent Processes:** Designed to run long-term, daemons are always available to handle events or requests. For instance, a network daemon listens for connections, while a print daemon monitors and processes print jobs;

- **Automated System Support:** Daemons simplify system management by automating routine tasks, reducing manual input. They usually start with the system boot, restart if they fail, and can operate at scheduled intervals for maintenance tasks like backups and software updates;
- **Naming Convention:** In Unix-based systems, daemon names traditionally end in "d" to denote their background nature, such as sshd (Secure Shell Daemon), httpd (HTTP web server daemon), and cron (task scheduler).

### 3.7.2 Common Examples

Common examples of daemons include System Daemons, Network Daemons, Device and Hardware Daemons, and Database Daemons, each performing vital roles to ensure the system's smooth operation.

- **System Daemons:** Manage core tasks essential to the operating system. For example, init or systemd serves as the parent of all processes, responsible for starting and managing other system services. The cron daemon is another crucial system component, scheduling and executing tasks at specified times to handle tasks like system backups or routine maintenance;
- **Network Daemons:** Facilitate network-related services. The sshd daemon, for instance, manages SSH (Secure Shell) connections, allowing secure remote logins and command execution. Similarly, the ftpd daemon oversees file transfers using the File Transfer Protocol (FTP), while the httpd daemon acts as a web server, processing HTTP requests and delivering web content;
- **Device and Hardware Daemons:** Oversee interactions with hardware components. cupsd manages printing tasks and connects with printers to queue and execute print jobs. Meanwhile, udevd handles dynamic device management by detecting and configuring hardware as it connects to or disconnects from the system, enabling the operating system to adapt to new or removed devices on the fly;
- **Database Daemons:** Support database management systems, enabling seamless data access. For example, mysqld is the daemon for MySQL databases, which processes client queries and connections. Similarly, postgres is the daemon for PostgreSQL, responsible for managing data storage and SQL queries. Together, these daemons ensure databases are accessible, responsive, and able to handle concurrent requests.

### 3.7.3 Advantages

- **Automation:** They automate regular tasks, boosting efficiency and freeing up user time;
- **Resource Optimization:** Running only as needed, daemons conserve resources, entering idle states when inactive;
- **Reliability:** Daemons are often monitored for automatic restarts, ensuring continuous availability even in the event of an error.

### 3.7.4 Disadvantages

- **Resource Demand:** Each daemon uses memory and CPU, and excessive or unnecessary daemons may slow down the system;
- **Security Vulnerabilities:** Network-interfacing daemons, like sshd or httpd, require proper configuration and maintenance to minimize security risks;
- **Complex Management:** Managing multiple daemons requires administration knowledge; incorrect setup can lead to instability.

### 3.7.5 Life-Cycle

- **Startup:** Daemons usually launch automatically at system boot, managed by an initialization system like init, systemd, or launchd on macOS;
- **Execution:** After starting, daemons operate in a loop, "listening" for triggers. For example, cron watches for scheduled tasks, while a network daemon listens for incoming connections;
- **Termination:** Daemons can be stopped or restarted by administrators, and the system itself terminates them during shutdown or reboot;
- **Restart and Resilience:** Daemons are typically configured to restart if they encounter errors, ensuring critical services remain available. Modern systems, such as systemd, can monitor and auto-restart essential daemons.

# Chapter 4

# Design

## 4.1 Analysis Review

The Analysis Phase for this hydroponic monitoring project reveals a well-researched and strategically planned approach to addressing genguine market needs. The proposed system intelligently combines environmental monitoring with AI-powered visual plant health assessment, targeting an underserved but growing market segment.

### 4.1.1 Strong Foundation and Market Opportunity

The problem identification demonstrates clear understanding of real user pain points: small-scale hydroponic growers need reliable automation without constant oversight, yet existing solutions either lack intelligence or remain prohibitively expensive. Market research validates strong growth potential (USD 5.53 billion to 18.12 billion by 2034) with clear gaps in accessible, intelligent monitoring systems.

The competitive analysis reveals substantial differentiation opportunities. While products like Growee (379-855 €) provide basic automation without visual diagnostics, and cheaper alternatives (29-76 €) offer only passive monitoring, the proposed 490 € solution with integrated AI analysis fills a genguine market void.

### 4.1.2 Thoughtful Technical Architecture

The proposed three-layer software architecture and comprehensive database design show sophisticated engineering thinking. The modular approach is particularly smart, allowing for incremental development and testing while maintaining scalability for future enhancements.

The choice of Raspberry Pi as the central platform demonstrates practical thinking about balancing computational capability with cost constraints. The integration of established technologies like OpenCV,

SQLite, and ONNX Runtime shows awareness of leveraging proven tools rather than reinventing solutions.

### 4.1.3 Realistic Approach to Complexity

While the machine learning component represents the most challenging aspect, the modular architecture supports a practical development strategy. Starting with reliable environmental monitoring and basic automation provides a solid foundation before tackling advanced visual analysis features.

The comprehensive system analysis, including detailed use cases, entity relationships, and sequence diagrams, suggests thorough planning that should help navigate implementation challenges. The clear separation of functional and non-functional reqguirements provides good gguidance for prioritizing development efforts.

### 4.1.4 Smart Budget Management

The 239 € budget reflects careful cost analysis and component selection. While tight, the allocation demonstrates understanding of where to invest (acceptable quality sensors) versus where to economize (structure and testing environment). This constraint-driven thinking often leads to more innovative and efficient solutions.

### 4.1.5 Promising Outlook

The Analysis phase establishes a solid foundation for successful development. The combination of genguine market opportunity, thoughtful technical design, and realistic constraint acknowledgment suggests strong potential for meaningful outcomes.

The team's adaptive approach and willingness to learn complex technologies while bguilding positions them well for tackling implementation challenges. Their comprehensive planning provides clear direction while maintaining flexibility to adjust as development realities emerge.

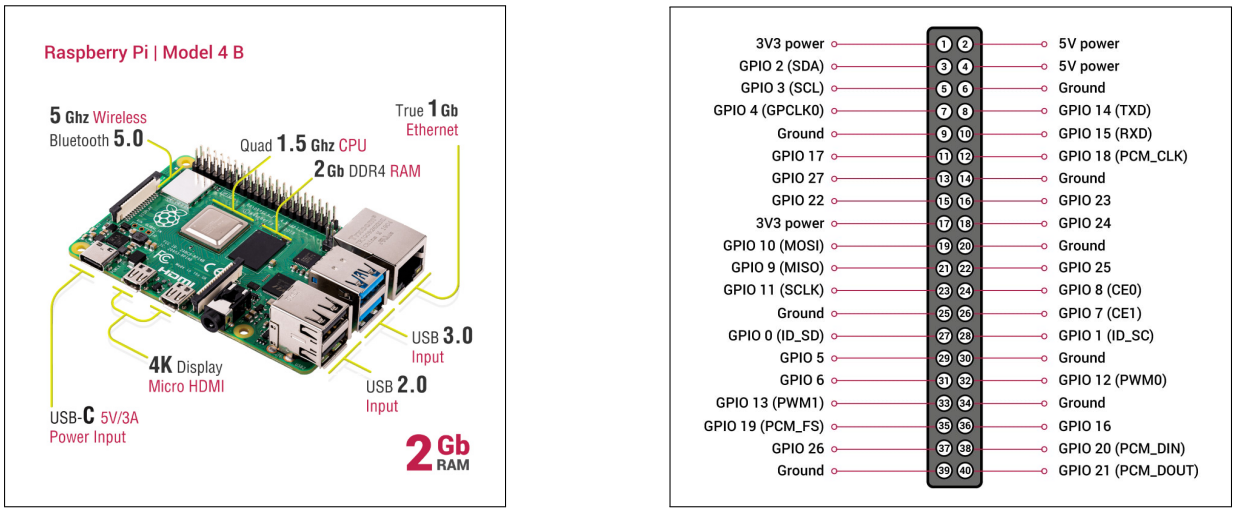
Most importantly, the focus on accessibility and practical value creation aligns well with current market needs, suggesting the final system could genguinely benefit the intended user community of hobbyists and small-scale growers.

## 4.2 Hardware Components Specification

In order to fulfill all the reqguirements, certain changes had to be made at the hardware level. The most egregious of all is the absence of a Dissolved Oxygen sensor and air pump, as this subsystem is not mission critical and very expensive. Other changes include the replacement of the water cooler for a water heater, based on the price and the fact that heating is more necessary than cooling in most house environments. This section will detail the main hardware components needed as well as their specifications.

### 4.2.1 Development Board

The Raspberry Pi 4 Model B serves as the "brains" of the system. It is responsible for executing computational tasks, managing connected peripherals, processing collected data, and running the necessary applications that enable the system's core functionality. Its compact form factor allows for easy integration into space-constrained environments, while its low power consumption ensures energy-efficient operation — an important factor for systems that reqguire continuous or remote operation. Additionally, the Raspberry Pi 4 B's support for a wide range of programming environments and community-developed libraries makes it a versatile and cost-effective choice for rapid prototyping and deployment. Overall, it provides a balanced combination of performance, affordability, and flexibility that meets the needs of both development and production stages.




(a) Board overview

(b) Board GPIO pinout

Figure 20: Raspberry Pi 4 B overview and GPIO pinout

**Specifications:**

- **Processor**
  - Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- **Memory**
  - 2GB LPDDR4-3200 SDRAM
- **Connectivity**
  - 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless
  - Bluetooth 5.0, BLE
  - Gigabit Ethernet
  - 2 USB 3.0 ports
  - 2 USB 2.0 ports
- **GPIO**
  - 40 pins
  - Output Voltage 3.3V
  - Max Output Current 16mA
- **Video**
  - 2 × micro-HDMI ports
  - 2-lane MIPI DSI display port
- **Card Support**
  - Micro SD
- **Power**
  - 5V DC via USB-C connector (minimum 3A)

**microSD Card**

The microSD Card operates as the Raspberry Pi's primary non-volatile storage device, containing the operating system, boot loader, executable programs, and user data. It facilitates both system startup and ongoing data management, effectively acting as the device's internal hard drive. Since performance and longevity depend heavily on the card's speed class and endurance rating, selecting an appropriate microSD card is vital for ensuring consistent and reliable operation under continuous use.

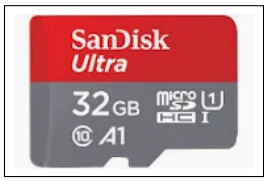



Figure 21: microSD Card

**Specifications:**

- **Capacity**
  - 32GB

**Power Supply Unit**

The Power Supply used in this system is not the official Raspberry Pi adapter. However, it provides eqguivalent performance and ensures stable operation of the Raspberry Pi 4 B and its connected peripherals. It serves as the primary energy source for the system, delivering consistent and reliable power reqguired for all components to function as intended. By maintaining a stable power output, the supply helps prevent issues such as system instability, data corruption, or unexpected shutdowns that can occur due to power fluctuations.

The chosen power supply was selected based on its proven compatibility and reliability, ensuring that it can support both continuous operation and varying system loads without degradation in performance. Its compact design and efficiency make it well-sguited for integration into the system's overall setup, contributing to a streamlined and dependable hardware configuration. Ensuring clean and consistent power delivery is a critical factor in maintaining the long-term stability and safety of the Raspberry Pi and the components it manages.

Figure 22: Power Supply Unit

**Specifications:**

- **Input Voltage**
  - 220 - 240 (V)
- **Output Voltage**
  - 5.1V
- **Current**
  - 3A
- **Connector**
  - USB Type-C

**Touch Display**

The touch display serves as the primary user interface, allowing intguitive and direct interaction with the program. It enables users to monitor data, navigate menus, and control system functions easily without the need for external input devices such as keyboards or mice. For this purpose, a Waveshare

display was selected due to its seamless compatibility and straightforward integration with the Raspberry Pi platform.

This display offers a responsive touch interface and clear visual output, enhancing both usability and accessibility. Its plug-and-play functionality simplifies setup and reduces development time, making it a practical choice for embedded and interactive applications. Additionally, its compact design aligns well with the system's overall goal of maintaining a small footprint while ensuring reliable and user-friendly operation.

Figure 23: Touch Display

**Specifications:**

- **Resolution**
  - 480x320
- **Dimensions**
  - 85x56 (mm)
- **Weight**
  - 54g
- **Communication Interface**
  - GPIO

### 4.2.2 Sensors

**pH Sensor**

The PH-4502C sensor module is designed to measure the pH level of liqguid solutions accurately and reliably. It is commonly used in applications such as aquariums, hydroponic systems, and water quality monitoring, where maintaining appropriate pH levels is essential. In this system, the PH-4502C provides real-time pH readings that can be processed and logged by the Raspberry Pi, allowing for automated analysis and control based on the measured values.

The module was selected due to its wide availability, ease of integration, and compatibility with microcontroller-based systems like the Raspberry Pi. Its straightforward calibration process and stable signal output make it sguitable for continuous monitoring applications, ensuring consistent and dependable pH measurement over time.

Figure 24: PH-4502C

**Specifications:**

- **Supply**
  - 5V
- **Range**
  - pH 0 - 14

- **Resolution**
  - 0.15pH
- **Communication Interface**
  - Analog

**Temperature Sensor**

The DS18B20 sensor is designed to accurately measure temperature in a variety of environments, including liqguid solutions. It is widely used in scientific, industrial, and environmental applications due to its reliability, precision, and ease of integration with digital systems. In this project, the DS18B20 is used to monitor the temperature of liqguids, providing essential data that can be processed and analyzed by the Raspberry Pi.

This sensor was chosen for its digital output, which simplifies data acqguisition and reduces the need for complex signal conditioning. Its waterproof design makes it sguitable for immersion in liqguids, ensuring accurate and consistent readings even in demanding conditions. The DS18B20's compatibility with the Raspberry Pi and its straightforward implementation make it a practical and efficient choice for temperature monitoring within the system.

Figure 25: DS18B20

**Specifications:**

- **Supply**
  - 3.3 - 5 (V)
- **Range**
  - -10 - 85 (°C)
- **Resolution**
  - 0.5°C
- **Communication Interface**
  - One Wire

**TDS Sensor**

A Total Dissolved Solids (TDS) sensor is one of the most common and effective methods for measuring the concentration of nutrients in hydroponic systems. Maintaining an appropriate nutrient concentration is essential to ensure optimal plant growth and health. In this project, the Seeed Grove TDS Sensor Module is used to monitor nutrient levels within the hydroponic solution, providing accurate and consistent readings that can be processed by the Raspberry Pi for data logging and control purposes.

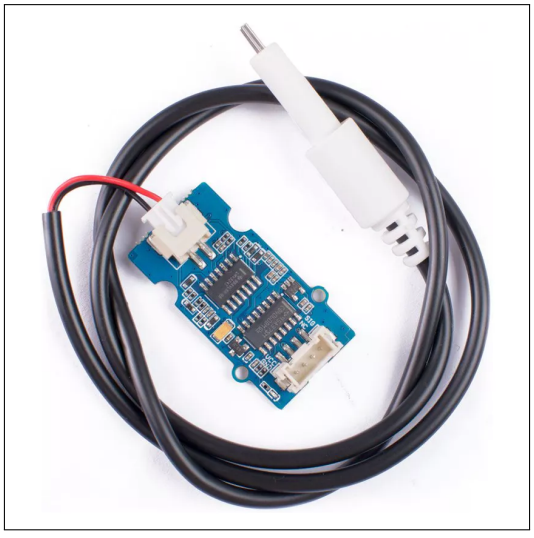


Figure 26: TDS Sensor

**Specifications:**

- **Supply**
  - 3.3 - 5 (V)
- **Output**
  - 0 - 2.3 (V)
- **Range**
  - 0 - 1000 (ppm)
- **Resolution**
  - 0.01 ppm
- **Communication Interface**
  - Analog

**Camera**

Since one of the key distingguishing features of the system is the ability to analyze plant leaves, a lightweight and capable camera is essential. For this purpose, the AZ-Delivery Raspberry Pi Camera Module was selected due to its compatibility, compact design, and high-quality image capture capabilities. The camera enables the system to take detailed photographs of plant leaves, which can then be processed for analysis of parameters such as color, texture, and potential signs of stress or disease.

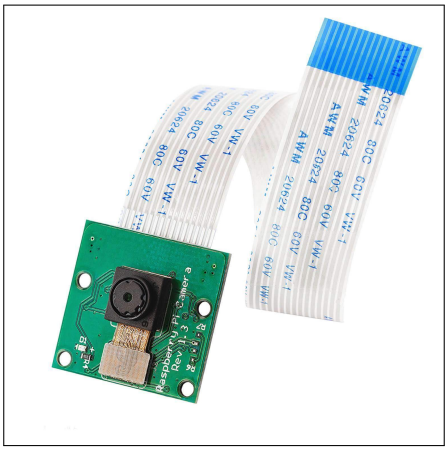


Figure 27: Camera

**Specifications:**

- **Dimensions**
  - 20x25x10 (mm)
- **Resolution**
  - 5MP
- **Communication Interface**
  - Serial Camera Interface

### 4.2.3 Actuators

**Water Heater**

Initially, the system design included a water cooler as the primary method for regulating the temperature of the nutrient solution. However, after further consideration and analysis, it was determined that heating the water would be a more efficient and cost-effective approach. In most hydroponic environments, maintaining the temperature within an optimal range is more frequently challenged by low rather than high temperatures, making water heating a simpler and more practical solution.

By focusing on heating instead of cooling, the system can achieve better energy efficiency, lower operational costs, and reduced hardware complexity.

Figure 28: Submersible Water Heater

**Specifications:**

- **Supply**
  - 220V and 200W

**Peristaltic Pumps**

To control both pH levels and nutrient dosing within the hydroponic system, peristaltic pumps will be used. These pumps were selected for their precision, reliability, and ability to handle liqguid dosing without contamination. The peristaltic mechanism allows accurate and repeatable delivery of solutions by compressing and releasing flexible tubing, ensuring that only the intended liqguid comes into contact with the tube's interior.

The use of peristaltic pumps also simplifies maintenance, as the flguid pathway can be easily replaced or cleaned when necessary. Their compatibility with the Raspberry Pi's control interface makes them well-sguited for automation, enabling the system to adjust chemical levels dynamically based on sensor feedback. Overall, this approach ensures a precise, hygienic, and efficient method for managing both nutrient and pH balance in the hydroponic environment.

Figure 29: Peristaltic Pump

**Specifications:**

- **Supply**
  - 3.3V

### 4.2.4 Auxiliar Components

#### Real Time Clock Module

Since the system performs scheduled sensor readings and timed operations, a Real-Time Clock (RTC) module is essential to maintain accurate timekeeping, even when the main power supply is disconnected. For this purpose, the DS3231 RTC module was chosen due to its high precision, stability, and proven reliability in embedded applications.

The DS3231 provides accurate tracking of time and date, including automatic leap-year compensation up to the year 2100. Its low power consumption allows it to operate efficiently, while an onboard backup battery ensures that the clock continues running in the event of a power outage. This capability ensures that all sensor readings, data logs, and automated actions remain chronologically consistent and properly timestamped.

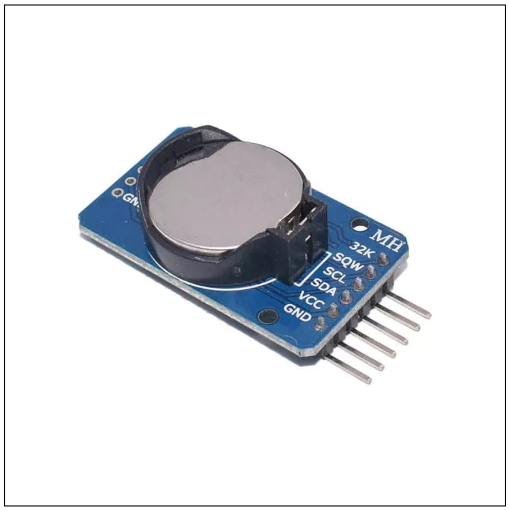



Figure 30: DS3231

**Specifications:**

- **Supply**
  - 2.3 - 5.5 (V)
- **Battery**
  - CR2032

- **Communication Interface**
  - I2C

**ADC Module**

Some of the sensors used in the system transmit their data in analog form, which the Raspberry Pi cannot process directly. To bridge this gap, an Analog-to-Digital Converter (ADC) is reqguired to translate these analog signals into digital values that can be interpreted and processed by the Raspberry Pi. For this purpose, the AZ-Delivery ADS1115 module was selected, as it provides high precision, excellent reliability, and straightforward integration.

The ADS1115 offers a simple yet effective interface for connecting multiple analog sensors, ensuring accurate and consistent data conversion without the need for complex circguitry. Its strong balance between performance, ease of use, and affordability makes it an ideal choice for this project. By incorporating the ADC, the system can seamlessly process readings from a variety of analog sensors, thereby enhancing the overall flexibility and accuracy of data acqguisition.

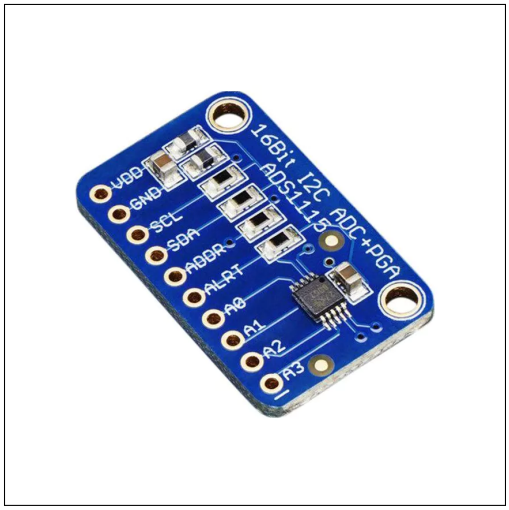



Figure 31: ADS1115

**Specifications:**

- **Supply**
  - 2 - 5.5 (V)
- **4 Analog Pins**

- **16 Bit Precision**
- **Communication Interface**
  - I2C

**Relay Module**

Certain actuators in the system reqguire more electrical power than the Raspberry Pi can safely supply directly. To manage these components, a relay module is used to control the power flow between the Raspberry Pi and the high-power devices. The relay acts as an electronically operated switch that allows the Raspberry Pi to turn actuators ON or OFF using only a low control voltage.

This approach provides both safety and efficiency, ensuring that the Raspberry Pi remains electrically isolated from the higher currents reqguired by pumps, heaters, or other power-demanding components. The relay module's ease of integration and reliable switching capability make it an essential part of the control architecture, allowing the system to automate key functions while protecting the Raspberry Pi from electrical overloads.

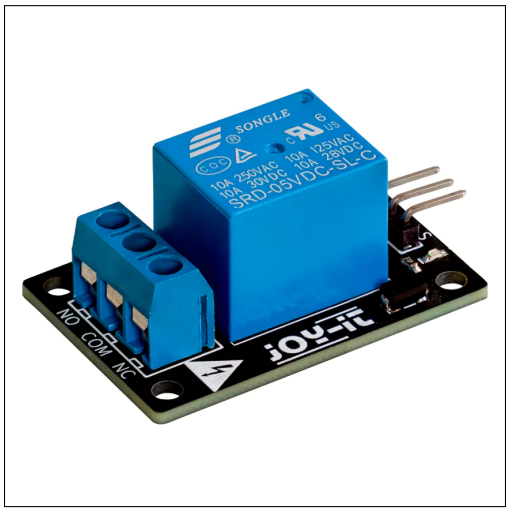



Figure 32: Relay Module

**Specifications:**

- **Maximum Load**
  - 6A - 230V AC

- 6A - 30V DC

- **Supply**
  - 5V

## 4.3 Hardware Interfaces Definition

To better illustrate the extent and connections of the hardware, the following figure and GPIO specification table were created.

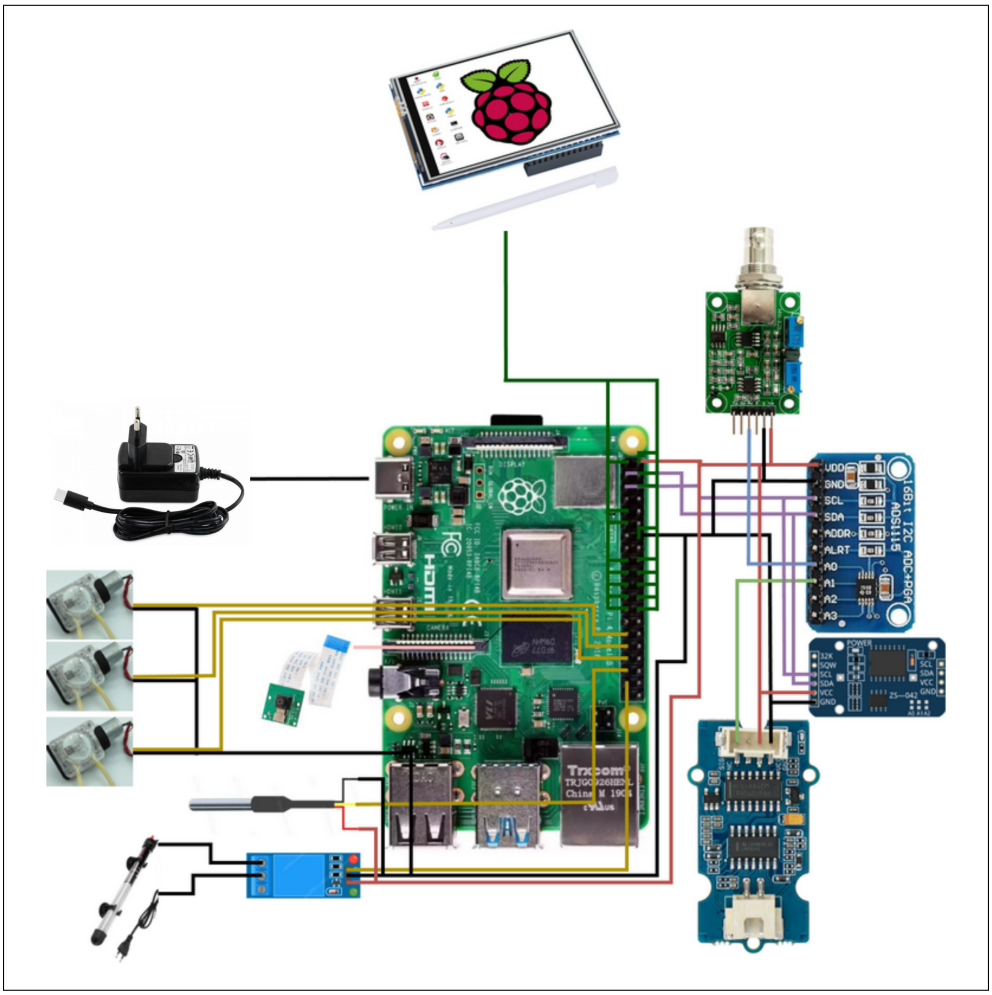



Figure 33: Hardware Connections

| Pin Number | Description |
|---|---|
| 2 | SDA for I²C Communication - ADC and RTC |
| 3 | SCL for I²C Communication - ADC and RTC |
| 17 | Display |
| 10 | Display |
| 9 | Display |
| 11 | Display |
| 5 | Nutrient Dosing Pump |
| 6 | PHU Pump |
| 13 | PHD Pump |
| 19 | One Wire - Temperature Sensor |
| 26 | Control of the Relay |
| 24 | Display |
| 25 | Display |
| 8 | Display |
| 7 | Display |

Table 4.1: GPIO Pin Connections

## 4.4 Tools and COTS used

This project utilizes a variety of tools and Commercial Off-The-Shelf (COTS) products to streamline workflows, accelerate development, and enhance software quality. This chapter highlights these components, detailing their specific roles and contributions throughout the development lifecycle.

By leveraging these tools, the team can utilize pre-bguilt functionalities and established practices, significantly minimizing the complexity and risks associated with bguilding a system from scratch. These resources span development, testing, documentation, and design, each playing a unique role in supporting the project. The following sections detail the specific tools employed, including Pthreads, Bguildroot, VS Code, SQLite, QT Creator, Overleaf, Draw.io, ONNX Runtime, OpenCV, and Canva, explaining their contributions to the design and implementation stages.

### 4.4.1 pthreads

pthreads, or POSIX Threads, is a standardized C-language API for creating and managing threads on Unix-like operating systems, including Linux and macOS. Defined by the POSIX (Portable Operating System Interface) standard, these APIs offer a reliable and portable way to implement multithreading, allowing different parts of a program to run concurrently. This capability enables more efficient program execution by improving resource use and enhancing performance in concurrent operations.


Figure 34: pthreads Logo

### 4.4.2 Bguildroot

Bguildroot is an open-source tool that simplifies the creation of custom Linux-based systems for embedded devices. It provides an efficient and flexible way to generate a complete Linux environment, including the kernel, a root filesystem, libraries, and application binaries. Widely used in the embedded community, Bguildroot significantly reduces the complexity and time reqguired to develop minimal, stable Linux systems tailored to specific hardware.

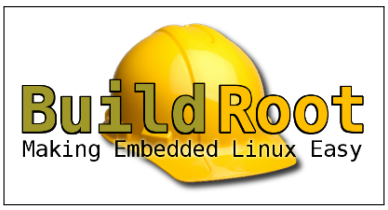



Figure 35: Bguildroot Logo

### 4.4.3 VS Code

Visual Studio Code (VS Code) is a free open-source code editor from Microsoft that is popular among developers for its robust features and flexibility. It supports a wide variety of programming languages and tasks, offering a lightweight editing experience combined with powerful development tools. This makes it ideal for projects ranging from small scripts to large-scale software development.

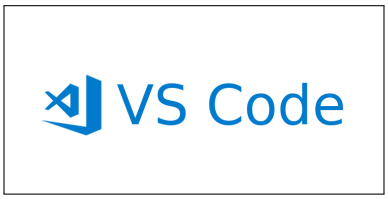



Figure 36: VS Code Logo

### 4.4.4 SQLite

SQLite is a C-language library that provides a lightweight, serverless, self-contained, and transactional SQL database engine. Unlike traditional client-server databases, SQLite reqguires no separate server process or complex configuration. Instead, it stores the entire database in a single cross-platform file on disk. This simplicity, combined with its small footprint and reliability, makes it an extremely popular choice for embedded systems, mobile applications, web browsers, and other applications needing a simple, embedded data storage solution.


Figure 37: SQLite Logo

### 4.4.5 QT Creator

Qt Creator is an integrated development environment (IDE) designed for bguilding applications with the Qt framework. It supports Qt's primary benefit of cross-platform development, enabling a single code-base to target Windows, macOS, Linux, Android, and iOS. Qt Creator simplifies this process by providing intguitive tools for designing, bguilding, and debugging Qt applications, making development more efficient.


Figure 38: QT Creator Logo

### 4.4.6 Overleaf

Overleaf is a cloud-based platform for collaborative writing and publishing, specializing in LaTeX documents. It features an easy-to-use interface that streamlines the LaTeX typesetting process, making it accessible to both new and experienced users. Overleaf is popular in academic and research circles for creating scientific papers, theses, and technical documents, offering robust support for complex formatting, citations, and mathematical equations. Additionally, its real-time collaboration and version control features enable teams to work efficiently and maintain consistency across projects.


Figure 39: Overleaf Logo

### 4.4.7 Draw.io

Draw.io is a free, online diagramming tool used to create flowcharts, organizational charts, network diagrams, and other visualizations. Its user-friendly interface makes it easy for individuals and teams to visualize complex concepts and processes. Its comprehensive features and collaborative functions make it a popular choice for professionals, educators, and students.


Figure 40: Draw.io Logo

### 4.4.8 ONNX Runtime

ONNX Runtime (Open Neural Network Exchange Runtime) is a high-performance inference engine for machine learning models. It is designed to execute models that have been exported to the open-standard ONNX format. This allows developers to train a model in one framework (such as PyTorch or TensorFlow) and deploy it for efficient inference across various platforms, including Windows, Linux, and embedded devices. ONNX Runtime abstracts the underlying hardware, optimizing execution on different accelerators like CPUs or GPUs to provide high speed and flexibility for real-world applications.


Figure 41: ONNX Runtime Logo

### 4.4.9 OpenCV

OpenCV (Open Source Computer Vision Library) is an extensive open-source library designed for real-time computer vision and machine learning applications. It provides thousands of optimized algorithms for a wide array of tasks, including image processing, feature detection, object tracking, and facial recognition. With interfaces for C++, Python, and Java, OpenCV is highly cross-platform, running on desktop and mobile operating systems. Its comprehensive toolset and high performance make it a fundamental tool in fields like robotics, augmented reality, medical imaging, and autonomous vehicles.

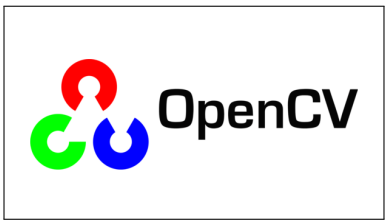



Figure 42: OpenCV Logo

### 4.4.10 Canva

Canva is a versatile and user-friendly graphic design platform that enables users to create professional-quality visuals with ease. Designed for both beginners and experienced designers, it provides an extensive library of customizable templates, images, icons, and fonts for a wide range of applications, including social media posts, presentations, posters, and marketing materials. Available through web and mobile interfaces, Canva allows for seamless design collaboration and cloud-based project management. Its intguitive drag-and-drop functionality, combined with powerful editing tools, empowers users to produce visually appealing content without the need for advanced design skills.


Figure 43: OpenCV Logo

## 4.5 Software Components Specification

Defining software components is essential to describe the purpose, behavior, and constraints of each part of a system. These specifications act as a detailed roadmap for developers, ensuring that everyone involved — designers, engineers, and stakeholders — shares a unified understanding of the system's reqguirements. Well-structured specifications promote effective communication, reduce the risk of misinterpretation, and form a strong foundation for development, testing, and long-term maintenance. By clearly outlining the software's objectives, they guide the entire development lifecycle and help ensure the final product aligns with the client's expectations and needs.

### 4.5.1 Project Classes Overview

To better understand the system architecture, the relationships between classes are detailed below based on the UML overview:

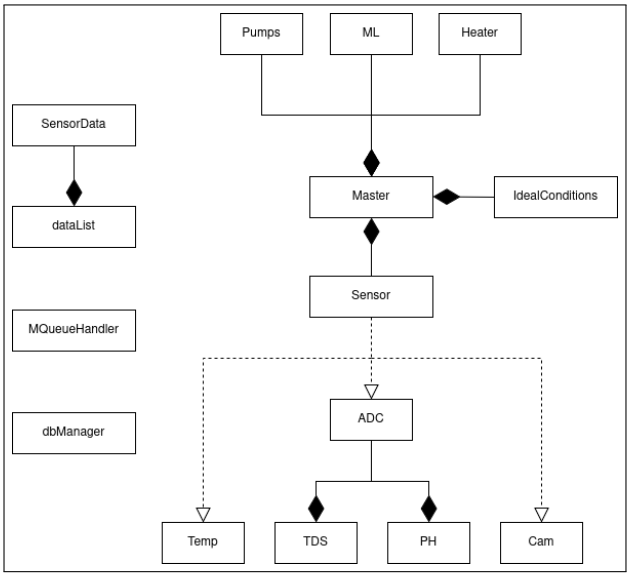



Figure 44: Project Classes UML Diagram

The Master class acts as the central coordinator for the system. It maintains a composition relationship with several functional components: Pumps, Heater, ML, and IdealConditions. This indicates that the Master class is responsible for the lifecycle and management of these instances.

The Master is also composed of the Sensor class, which functions as the abstract interface for the sensing subsystem. Implementing a realization relationship, the Temp, Cam, and ADC classes implement

the Sensor interface. This implies that these classes do not merely inherit state, but strictly fulfill the contract (methods and behaviors) defined by the Sensor abstraction.

Furthermore, the TDS and PH classes share a composition relationship with the ADC class, indicating that these specific analog sensor types are constructed using and managed by the ADC hardware wrapper.

### 4.5.2 Master Class

To adhere to Object-Oriented programming principles and ensure robust system encapsulation, the Master class acts as the central orchestrator for the LeafSense project. It is designed to manage the lifecycle of hardware components and coordinate parallel execution flows.

#### Structural Relationships (Composition & Realization)

The Master class maintains a compositional relationship with the system's actuators and processing units, directly managing instances of the Pumps, Heater, ML, and IdealConditions classes. Regarding the sensing architecture, the system implements a realization relationship to define a standardized interface. The Temp, Cam, and ADC classes implement the contract defined by the abstract Sensor interface. The Master class instantiates these specific objects (along with TDS and PH, which are physically interfaced via the ADC) to aggregate environmental data.

#### Concurrency and Resource Management

The Master class abstracts the complexity of the system's multitasking architecture using the POSIX threading (pthread) library:

- **Thread Management:** The class defines dedicated thread handles (e.g., tReadSensors, tCam, tNutrients) to execute distinct tasks—such as computer vision processing and nutrient dosing—in parallel.
- **Synchronization:** To ensure thread safety and data integrity, the class utilizes mutexes (e.g., mutexCam, mutexRS) and condition variables. This prevents race conditions when accessing shared resources like sensor data or hardware drivers.
- **Scheduling:** The inclusion of sched_param attributes indicates the implementation of priority-based scheduling, ensuring that critical control loops receive the necessary CPU time relative to lower-priority tasks.

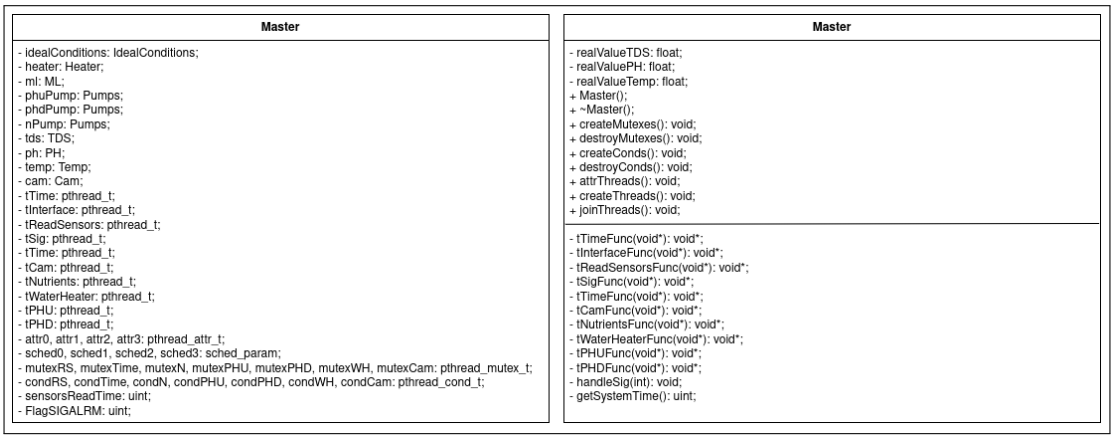



Figure 45: Master Class UML Diagram (Part 1 - Left and Part 2 - Right)

### 4.5.3 Sensors Classes

To effectively monitor environmental variables (water quality) and plant health (leaf status), the system employs a hierarchical sensor architecture. The design centers on the Sensor class, which acts as an abstract interface to establish a polymorphic contract for data acquisition.

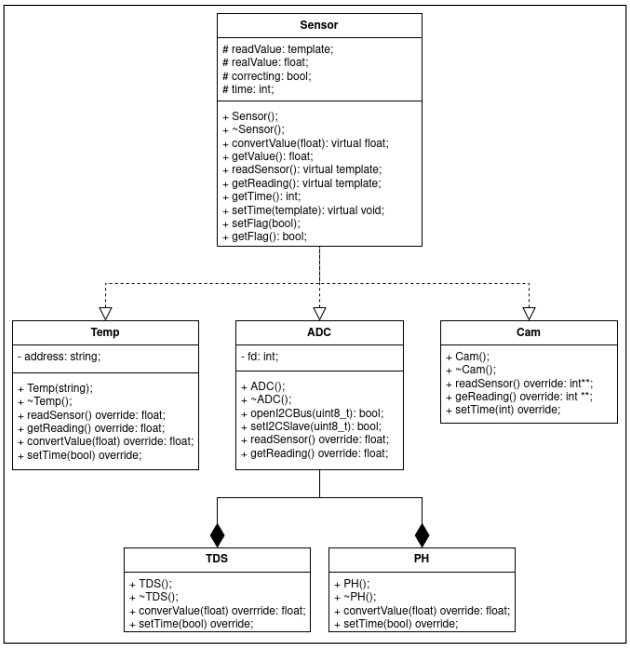



Figure 46: Sensors Classes UML Diagram

### Interface Realization and Implementation

To maximize code reuse and enforce a standard interaction model, the Temp, Cam, and ADC classes define a realization relationship with the Sensor interface.

- **Temp:** Implements the interface for the 1-Wire protocol, providing digital temperature readings.
- **Cam:** Implements the interface for high-dimensional data, returning an image matrix (int**) rather than a scalar value.
- **ADC:** Serves as a hardware abstraction layer for the system's I2C bus. Unlike the direct implementations, the ADC class maintains a compositional relationship with the TDS and PH classes. This design encapsulates the complexity of analog sensor calibration and channel switching within the ADC wrapper, while still adhering to the high-level Sensor contract.

### Data Acquisition Pipeline

The system leverages virtual functions and templates to handle heterogeneous data types (voltage, float, image arrays) through a unified flow:

- **readSensor():** A virtual method that triggers the physical reading. It captures the raw data—whether it be a voltage signal or a picture frame—and updates the object's internal state.
- **convertValue():** Transforms raw hardware readings into normalized, human-readable units (e.g., converting ADC voltage to pH level or ppm).
- **getValue():** Returns the final, processed data to the Master class for decision-making.

### Dynamic Timing & State Control

The Sensor interface includes a synchronization mechanism via the time attribute and setTime() method to manage sampling intervals:

- **Camera:** Operates on a fixed, long-interval schedule (e.g., daily) initialized at startup.
- **Environmental Sensors:** The sampling rate for Temp, TDS, and pH is dynamic. It is governed by the correcting boolean flag (a protected attribute). When actuators are active (the "correcting" state), the system shortens the sampling interval to provide real-time feedback; otherwise, it reverts to a standard monitoring interval to conserve system resources.

### 4.5.4 Actuators Classes

The system's output capabilities are modeled through distinct classes representing both physical actuation and computational inference. While functionally distinct, the Pumps, Heater, and ML classes are grouped as the system's execution units, responsible for acting upon the environment or processing complex data streams.

#### Physical Actuators (Pumps & Heater)

The hardware interface for these components is standardized through GPIO management. Both the Pumps and Heater classes enforce hardware-specific instantiation, requiring a gpioPin (uint8) to be passed to the constructor to establish the physical control line.

- **Pumps:** This class implements a command-based interface designed for precise fluid dosing. The pump(uint) method activates the designated GPIO pin for a specific duration (passed as an unsigned integer), abstracting the timing logic required to dispense exact nutrient or pH solution volumes.
- **Heater:** Unlike the transient operation of the pumps, the Heater class implements a stateful control model. It encapsulates a boolean state attribute and provides setState() and getState() accessors. This allows the system to maintain and query the persistent status (ON/OFF) of the heating element without redundant hardware checks.

#### Computational Inference (ML)

The ML class acts as a high-level processing unit. While not a physical actuator, it serves as a critical system output that transforms raw data into actionable intelligence. The class encapsulates the computer vision pipeline via the analyze(photo) method. This function accepts captured image data and returns a discrete numerical value (uint), which corresponds to a specific classification of the plant's well-being (e.g., healthy, nutrient-deficient, pest-detected).

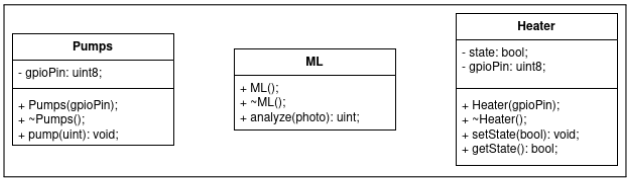



Figure 47: Actuators Classes UML Diagram

### 4.5.5 IdealConditions Class

To facilitate robust system configuration and user customizability, the IdealConditions class acts as a dedicated data container for the project's control parameters. This class encapsulates the 'business logic' of the environment by centralizing the storage of user-defined thresholds.

#### Range-Based Control Logic

Rather than storing single setpoints, the class attributes are designed to define operational ranges (e.g., tds_min / tds_max, pH_min / pH_max). This structure is critical for the system's control algorithms, as it allows for the implementation of hysteresis. By defining upper and lower bounds for Temperature, TDS, and pH, the system prevents the rapid oscillation ('chattering') of actuators (pumps and heaters) that would occur if triggered by a single threshold value.

#### Encapsulated Interface

The class provides a strict interface for parameter modification and retrieval:

- **Setters:** Methods such as setTDS(float, float) allow the Master class to update the minimum and maximum bounds atomically, ensuring that a valid range is always maintained.
- **Getters:** The accessor methods (getTDS(), getPH()) are designed to return pointers (float*) to the data. This allows the calling process to efficiently retrieve the full range array (both min and max values) in a single method call, simplifying the comparison logic within the main control loops.

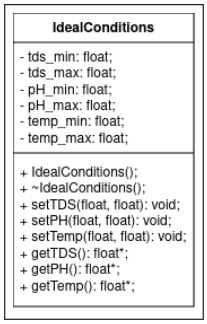



Figure 48: IdealConditions Class UML Diagram

### 4.5.6 MQueueHandler Class

To facilitate reliable data persistence without compromising system responsiveness, the MQueueHandler class implements a thread-safe message queue. This component serves as the asynchronous communication bridge between the real-time LeafSense control loop and the background Database daemon.

#### Architectural Role: Decoupling & Buffering

The primary architectural goal of this class is decoupling. Database write operations have unpredictable latency; performing them synchronously would block the main application thread and disrupt critical sensor timing. The MQueueHandler acts as an intermediate buffer, implementing the Producer-Consumer pattern to allow the system to 'fire and forget' data packets while a separate consumer thread handles storage operations.

#### Concurrency Primitives (POSIX Standard)

To maintain consistency with the system's core architecture, the class utilizes the POSIX threading library (pthreads) for synchronization:

- **Mutual Exclusion:** A pthread_mutex_t (queueMutex) is employed to enforce critical sections. This lock protects the underlying std::queue during push and pop operations, preventing race conditions when multiple threads attempt to log data simultaneously.
- **Signaling:** A pthread_cond_t (queueCondition) is used to efficiently manage thread states. It enables the consumer thread to sleep (block) on the condition variable when the queue is empty and be woken up via a signal immediately when a new message is produced, optimizing CPU usage.

#### Operational Interface

The class exposes a streamlined API for message management:

- **sendMessage(std::string):** The 'produce' operation. It locks the mutex, pushes the message, unlocks, and signals the condition variable.
- **receiveMessage():** The 'consume' operation. It retrieves the next message. If the queue is empty, it utilizes pthread_cond_wait to block execution until data arrives.

- **Lifecycle Management:** Unlike C++ wrappers, the class constructor and destructor are explicitly responsible for initializing (pthread_mutex_init) and destroying (pthread_mutex_destroy) the synchronization primitives to prevent resource leaks.

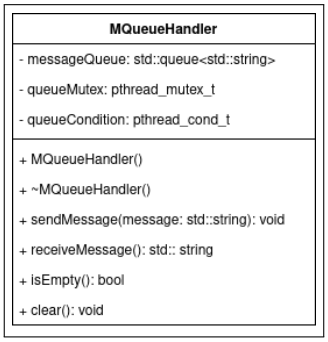



Figure 49: MQueueHandler Class UML Diagram

### 4.5.7 dataList and SensorData Classes

To manage the continuous stream of environmental metrics collected by the LeafSense system, the dataList class functions as a centralized, thread-safe repository. This component is responsible for the aggregation, storage, and statistical analysis of sensor inputs throughout the system's runtime.

#### Data Encapsulation (SensorData)

The foundation of this module is the SensorData structure, which acts as a standardized Data Transfer Object (DTO). It encapsulates a complete snapshot of the environment at a specific moment, containing floating-point fields for temperature, TDS, and pH. Additionally, it stores a cameraImage string (representing the file path to the captured visual data) and a timestamp (long) for temporal tagging. This structured approach ensures data consistency across the application.

#### Thread-Safe Storage (POSIX Implementation)

The dataList class manages a collection of these snapshots using a standard std::vector<SensorData>. To ensure data integrity in a multi-threaded environment—where the GUI might query history while the Master thread appends new readings—the class implements POSIX synchronization primitives:

- **Mutual Exclusion:** A pthread_mutex_t (dataMutex) is utilized to protect the vector. Every read (getLatestReading) or write (addSensorReading) operation first acquires this lock, preventing race conditions and iterator invalidation.

**Analytical Interface**

The class exposes a comprehensive API for both real-time monitoring and historical trend analysis:

- **Accessors:** Methods such as getLatestReading() provide immediate access to current conditions for the UI, while getAllReadings() retrieves the full dataset.
- **Statistical Processing:** The class includes built-in analytical methods—getAverageTemperature(), getAverageTDS(), and getAveragePH(). These compute mean values across the stored dataset, facilitating the detection of long-term stability issues or environmental drifts.
- **Lifecycle Control:** Utility methods like getDataCount() and clearData() allow the system to monitor memory usage and flush old records periodically to optimize performance on the embedded hardware.

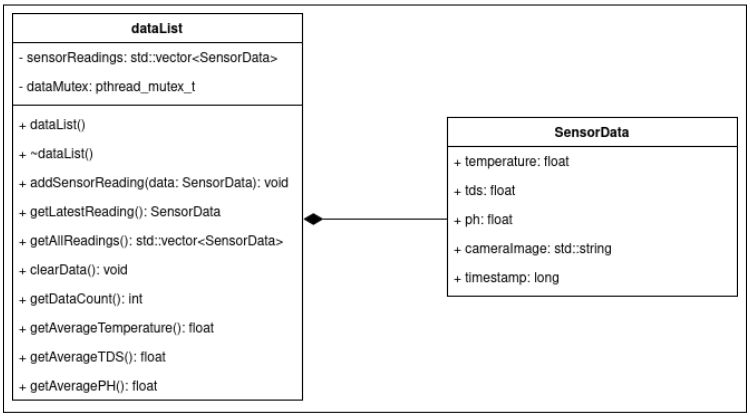



Figure 50: dataList and SensorData Classes UML Diagram

### 4.5.8 dbManager Class

The dbManager class acts as the dedicated persistence layer for the LeafSense system, abstracting the complexities of direct database interaction. By centralizing all storage operations within this class, the architecture ensures a clear separation of concerns: the application logic focuses on data generation,

while the dbManager handles the specifics of data serialization and storage (e.g., managing SQLite file handles or SQL query construction).

### Configuration and State

The class maintains the database connection state through the private info attribute (string). This attribute typically holds the database connection string, file path, or configuration parameters required to establish a valid link to the underlying storage engine.

### Operational Interface (CRUD)

The class exposes a simplified, high-level API for data manipulation, effectively implementing a subset of standard CRUD (Create, Read, Update, Delete) operations:

- **insert(info):** Handles the Create aspect of persistence. This method accepts a data payload (likely a formatted string or SQL command), validates it, and commits the new record to the database.
- **read(info):** Handles the Read aspect. It executes queries against the database to retrieve historical records or specific configuration settings based on the provided query parameters.
- **remove(info):** Handles the Delete aspect. This method ensures the secure removal of obsolete or erroneous data records based on the identifiers or criteria passed in the argument.

### Architectural Integration

In the broader system context, the dbManager typically acts as the 'Consumer' for the MQueue-Handler. It runs in a background process or thread, picking up messages from the queue and using its insert() method to permanently store sensor readings without blocking the main real-time control loop.

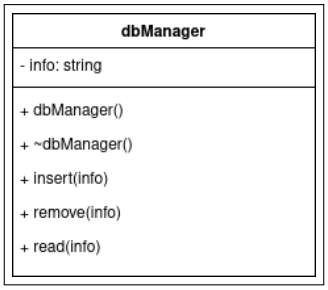



Figure 51: DatabaseManager Class

### 4.5.9 Thread Overview and Priorities

Effective resource management in a real-time system relies on a clear hierarchy of execution. As depicted in the provided overview, the LeafSense architecture stratifies its threading model into four distinct tiers, ensuring that time-sensitive hardware operations always take precedence over non-critical tasks.

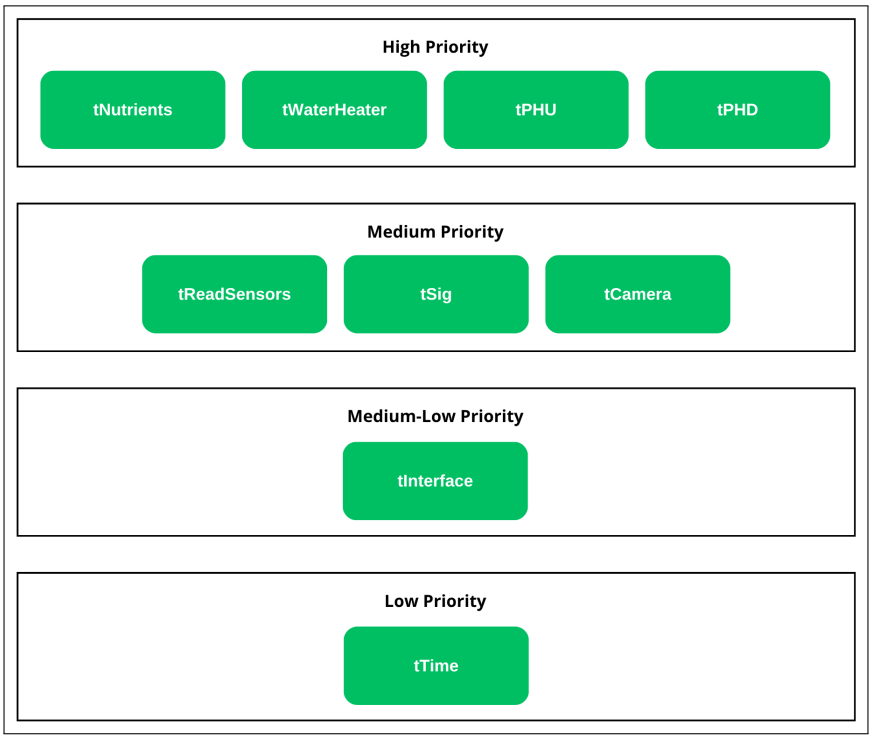



Figure 52: Threads Overview and Priorities

- **High Priority (Actuation and Control):** At the apex of the hierarchy are the threads responsible for physical intervention: tNutrients, tWaterHeater, tPHU, and tPHD. These threads directly manipulate the system's hardware outputs. By assigning them the highest priority, the scheduler ensures that any command to dispense chemicals or adjust temperature is executed immediately, minimizing latency and preventing dangerous overshoot conditions that could harm the biological environment.

- **Medium Priority (Sensing and Timing):** The second tier supports the system's awareness and rhythm. This layer includes tReadSensors and tCamera, which require consistent CPU access to maintain regular sampling intervals. Crucially, this level also contains tSig, the signal management thread. While tSig orchestrates the timing for other threads, it sits slightly below the actuators to

guarantee that a "stop" command to a pump is never delayed by a routine sensor check.

- **Medium-Low Priority (User Interface):** tInterface occupies the third layer, managing the interaction between the system and the user. Since human reaction times are significantly slower than machine cycles, the User Interface is deprioritized relative to the control loops. This ensures that complex display updates or console logging never block the core environmental monitoring processes.

- **Low Priority (Background Tasks):** The foundation of the stack is tTime. Functioning essentially as an idle or background task, this thread consumes CPU cycles only when no other higher-priority operations are pending. It handles low-resolution timekeeping and general housekeeping duties without impacting system performance.

### 4.5.10 Threads Behavior

#### captureSignal and triggerSignal

To streamline thread synchronization and mitigate the risks associated with raw concurrency management (such as deadlocks or missed signals), the system encapsulates the handling of condition variables within two dedicated wrapper functions: captureSignal and triggerSignal.

The captureSignal function as illustrated in the flowchart below, standardizes the 'consumer' side of the synchronization event. This function is designed to block the calling thread until a specific condition is met. It implements a strict three-step POSIX locking protocol:

- **Acquire Lock:** The function begins by calling pthread_mutex_lock on the passed mutex reference (&mutex). This ensures the thread enters the critical section safely.

- **Wait State:** It then invokes pthread_cond_wait, passing both the condition variable (&cond) and the mutex. This atomically releases the mutex and suspends the thread's execution, placing it in a waiting queue until a signal is received.

- **Release Lock:** Upon receiving a signal and waking up (which automatically re-acquires the mutex), the function executes pthread_mutex_unlock to release the resource before terminating.

This encapsulation ensures that every thread in the LeafSense system suspends and resumes execution using a consistent, thread-safe methodology.

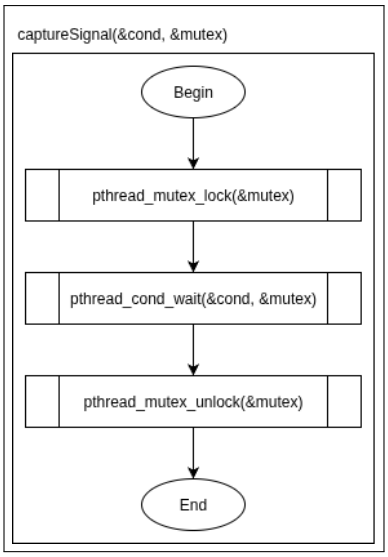



Figure 53: captureSignal

Complementing the consumer logic, the triggerSignal function encapsulates the 'producer' side of the synchronization event. Its primary role is to notify and wake up threads that are currently suspended in the captureSignal state (e.g., alerting tReadSensors that the timer has elapsed).

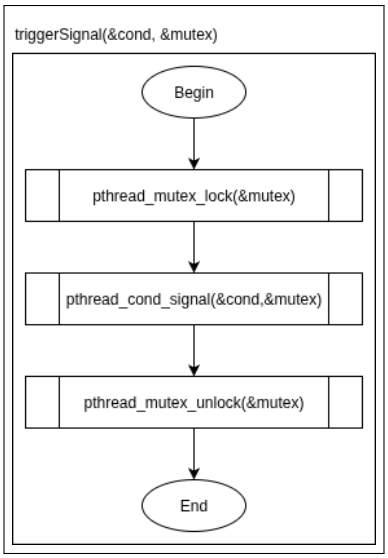



Figure 54: triggerSignal

The flowchart above demonstrates the triggerSignal function's execution path, which mirrors the locking protocol of its counterpart:

- **Lock Acquisition:** Execution begins by calling pthread_mutex_lock. Acquiring the mutex before

signaling is a best practice that ensures predictable scheduling and prevents potential race conditions (such as 'lost wake-up' signals) where a signal might be sent before the consumer thread has fully established its wait state.

- **Signaling:** The core operation, pthread_cond_signal, is executed to unblock at least one thread currently waiting on the specified condition variable.
- **Lock Release:** Finally, pthread_mutex_unlock is called to release the resource. This step is vital because the awakened thread (in captureSignal) cannot proceed until it can re-acquire this specific mutex.

### tTime and tSig

The system's temporal architecture relies on a producer-consumer relationship between the tTime and tSig threads. This design decouples the hardware timer interrupts from the application logic, ensuring stable scheduling.

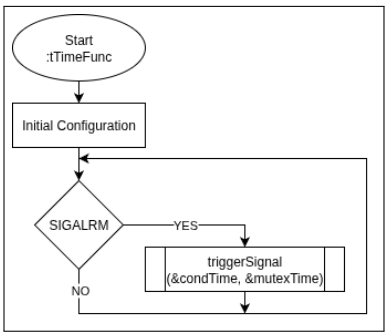



Figure 55: tTimeFunc

The tTimeFunc as depicted in the flowchart above functions as the system's heartbeat. Its execution flow is minimal by design to ensure low latency:

- **Interrupt Handling:** After initial configuration, the thread enters an event loop waiting for the SIGALRM signal (an OS-level timer interrupt).
- **Event Bridging:** Upon receiving this signal, the thread immediately calls triggerSignal on the condTime condition variable. This action effectively translates a raw operating system signal into a thread-safe synchronization event, notifying the rest of the application that a time quantum has elapsed.

The tSigFunc acts as the central dispatcher. As shown in the flowchart of Figure 56, it remains in a suspended state via captureSignal until woken by tTime. Once active, it executes a logic sequence to determine which subsystems require attention:

- **Time synchronization:** It retrieves the current system time (Master::getSystemTime) and compares it against the scheduled activation times for the Camera (Cam::getTime) and the environmental sensors ((Temp, PH, TDS)::getTime).
- **Conditional Triggering:**
  - **Camera:** If the current time matches the configured camera schedule, it fires triggerSignal on condCam to wake the computer vision thread.
  - **Sensors:** If the time matches the sensor sampling schedule, it fires triggerSignal on condRS to wake the tReadSensors thread.
- **Adaptive Scheduling:** Finally, the thread executes setTime(correcting). This step is crucial for the system's dynamic response; it updates the next sampling interval based on whether the system is currently in a 'correcting' state (actuators active), effectively increasing the sampling rate during active control phases.

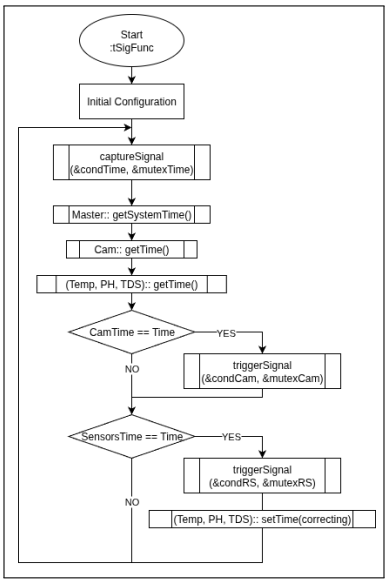



Figure 56: tSigFunc

**tReadSensors**

The tReadSensors thread functions as the system's primary control loop, bridging the gap between environmental monitoring and physical actuation. As illustrated in the flowchart below, the thread's execution is divided into a data acquisition phase and a sequential decision-making cascade.

The cycle begins in a suspended state, waiting for the condRS signal from the central scheduler (tSig).

- **State Reset:** Upon waking, the thread first resets the correcting flag to False. This default state assumes the environment is stable until proven otherwise by the new readings.
- **Data Pipeline:** The thread executes the readSensor() method for the PH, Temperature, and TDS sensors. These raw inputs are immediately processed via convertValue() to transform electrical signals into human-readable units (e.g., pH level, Celsius, ppm), which are then stored in a local values array.

The processed data is passed through a series of logic gates that compare current readings against the IdealConditions thresholds:

- **pH Control:** The system evaluates if the pH is outside the target range. A LOW reading triggers the condPHU (pH Up) signal, while a HIGH reading triggers condPHD (pH Down).
- **Nutrient Control (TDS):** The Total Dissolved Solids level is checked against the ideal setpoint. If the concentration is insufficient (NO), the thread signals condN to dispense nutrients.
- **Temperature Control (Hysteresis):** The temperature logic is state-aware. It checks not only the temperature but also the current state of the heater (On/Off). This prevents redundant switching—triggering the heater signal (condWH) only if the water is cold and the heater is currently off, or if the water is hot and the heater is currently on.

If any of the above conditions require intervention, the correcting flag is set to True. As detailed in the previous timing diagrams, this flag serves as a feedback mechanism to the tSig thread, instructing the system to increase the sampling frequency to monitor the active correction process closely.

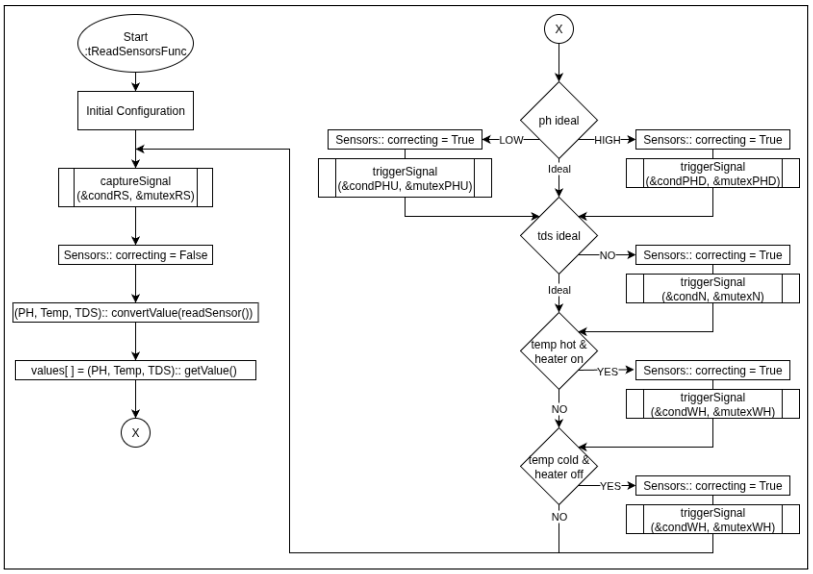



Figure 57: tReadSensors

### tCamera

The tCamera thread (Figure 58) is dedicated to the system's computer vision and plant health monitoring capabilities. It operates on a scheduled basis, triggered by the tSig dispatcher, to capture and process visual data.

- **Scheduled Activation:** As illustrated in the flowchart, the thread's execution is initially suspended. It awaits a specific signal via captureSignal(&condCam, &mutexCam). This ensures that camera operations, which can be resource-intensive, only occur at predetermined intervals as dictated by the tSig thread's scheduling logic.

- **Image Acquisition:** Upon receiving the activation signal, the thread proceeds to execute Cam::takePhoto(). This method encapsulates the low-level interactions with the camera hardware, acquiring a new image frame from the environment.

- **Visual Analysis and Feedback:** Following acquisition, the thread immediately initiates Cam::analysePhoto(). This step involves the application of machine learning algorithms (likely housed within the ML class as discussed previously) to interpret the visual data. The outcome of this analysis—which could include identifying signs of disease, nutrient deficiency, or growth abnormalities — is then processed. While not explicitly detailed in this flowchart, the result of this analysis would typically be logged through the MQueueHandler for database storage and potentially communicated to the user via the tInterface thread, informing them of the plant's health status.

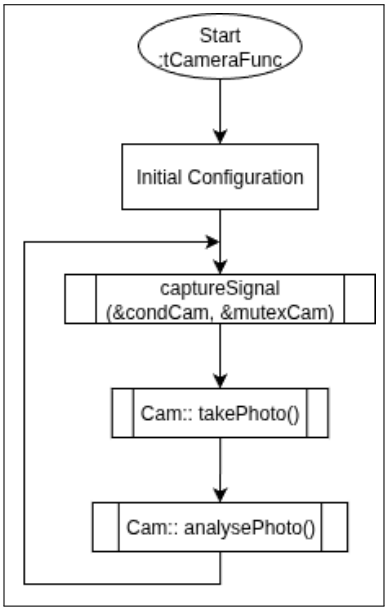



Figure 58: tCamera

### Actuator Threads

To maintain architectural consistency and simplify system maintenance, the actuation subsystem employs a unified design pattern across all control threads. As illustrated in the flowcharts for tPHD (pH Down), tPHU (pH Up), tNutrients, and tWaterHeater, each thread operates as a dedicated worker loop governed by the system's synchronization primitives.

Every actuator thread follows an identical lifecycle, ensuring deterministic behavior:

- **Suspension:** After initial configuration, the thread enters a blocking state via the captureSignal wrapper. It waits exclusively on its dedicated condition variable (e.g., condN for nutrients, condWH for the heater) and mutex.
- **Atomic Execution:** Upon receiving a signal from the tReadSensors thread, the actuator wakes to perform a single, specific hardware operation.
- **Cycle Reset:** Immediately after the operation completes, the thread loops back to the suspension state. This 'sleep-until-needed' approach ensures that actuators consume zero CPU resources when idle.

While the control flow is uniform, the specific payload functions differ based on the physical component being controlled:

- **Fluid Dosing Threads (tPHU, tPHD, tNutrients):** These threads interface with the Pumps class. When triggered, they execute the dose(quantity) method, driving the peristaltic pumps for a calculated duration to dispense a precise volume of reagent.

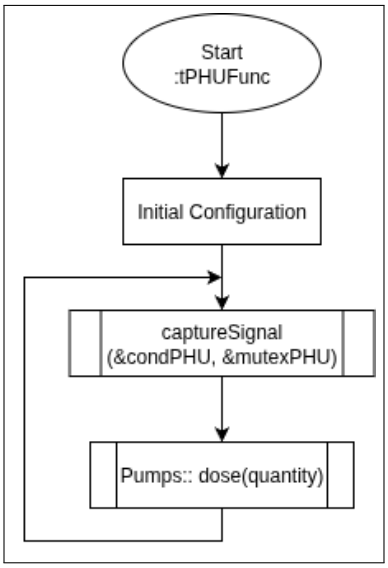



Figure 59: tPHU

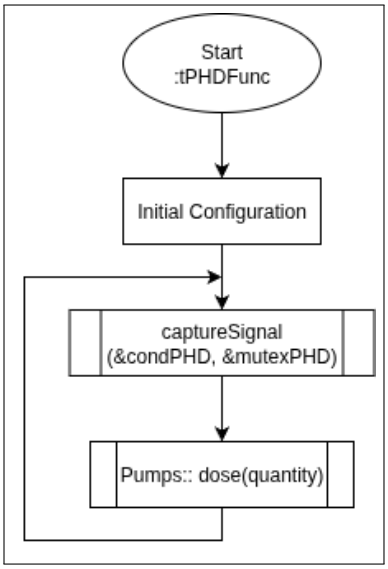



Figure 60: tPHD

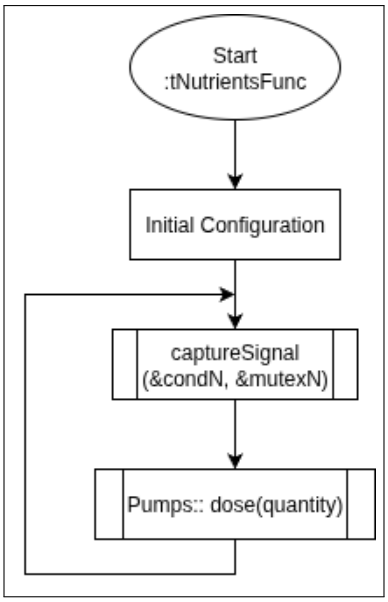



Figure 61: tNutrients

- **Thermal Control Thread (tWaterHeater):** This thread manages the immersion heater. Unlike the transient dosing action of the pumps, it executes Heater::setState(state), effectively toggling the binary status (ON/OFF) of the heating element to maintain the target water temperature.

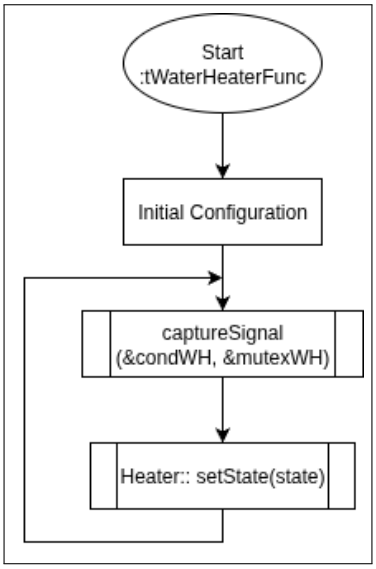



Figure 62: tWaterHeater

### 4.5.11 Database Daemon

To ensure data integrity and prevent blocking operations within the real-time control loop, the system delegates all persistence tasks to a standalone background process: the Database Daemon.

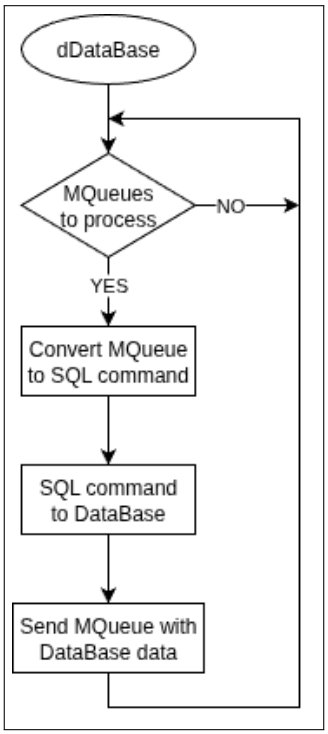



Figure 63: dDatabase

#### The Event Loop

As depicted in the flowchart, the daemon operates on a continuous event loop. Its primary function is to monitor the inter-process communication channels. It constantly checks the incoming POSIX message queue for pending requests. If the queue is empty (NO), the daemon remains in a waiting state, conserving system resources.

#### Protocol Translation (Message to SQL)

When a message is detected (YES), the daemon initiates a translation process. Since the main application communicates via serialized strings or data structures, the daemon must parse this raw input and convert it into a valid SQL transaction statement (e.g., INSERT, SELECT). This abstraction layer ensures that the main application remains agnostic to the specific SQL syntax used by the underlying database.

**Execution and Feedback**

Once the command is prepared, the daemon executes it against the database engine.

- **Write Operations:** For sensor logging, the data is committed to storage.
- **Read Operations:** For queries (e.g., requests from the Interface), the daemon retrieves the result set and pushes the requested data back into the outgoing message queue. This completes the bidirectional communication loop, allowing the UI to display historical data without directly accessing the database file.

### 4.5.12 Threads Synchronization Strategy

To ensure deterministic execution and maximize system throughput within the multi-threaded architecture, a robust synchronization strategy is essential. The LeafSense system employs three primary POSIX synchronization primitives: Mutexes, Condition Variables, and Message Queues.

| Condition | Mutex |
|---|---|
| condRS | mutexRS |
| condTime | mutexTime |
| condN | mutexN |
| condPHU | mutexPHU |
| condPHD | mutexPHD |
| condWH | mutexWH |
| condCam | mutexCam |

Figure 64: Condition Variables and Mutexes

**Resource Protection and Signaling (Mutex & Condition Pairs)**

As illustrated in the provided table, the system implements a strict pairing between mutexes (Mutual Exclusion) and condition variables.

- **The Mutex (mutex*)** acts as the locking mechanism, ensuring that only one thread can access a critical shared resource—such as the camera driver (mutexCam) or sensor data buffers (mutexRS)—at any given time.

- **The Condition Variable (cond*)** serves as the signaling mechanism. It allows threads to suspend execution (sleep) efficiently while waiting for a specific state change (e.g., 'Camera Ready' or 'Time to Dose Nutrients'), rather than consuming CPU cycles through busy-waiting.

**Specific Resource Mapping**

The table highlights the specific synchronization handles defined in the Master class:

- **condRS / mutexRS:** Synchronizes the Read Sensors thread, ensuring data integrity during acquisition.
- **condCam / mutexCam:** Manages access to the computer vision hardware to prevent concurrent capture requests.
- **condPHU, condPHD, condN:** Govern the specific actuation threads for pH Up, pH Down, and Nutrient dosing, allowing the Master to trigger these independent processes precisely when needed.

**Inter-Process Communication**

Distinct from the internal thread synchronization shown in the table, Message Queues are utilized for external communication. This primitive bridges the main application thread with the independent Database daemon, decoupling the real-time control loop from the latency-sensitive storage layer.

### 4.5.13 Inter-Thread Communication (ITC)

To illustrate the internal control flow, the Inter-Thread Diagram of the Figure 65 exposes the signaling hierarchy governed by condition variables. Rather than utilizing a flat communication model, the system implements a sequential dependency chain to ensure deterministic execution:

- **The Trigger:** The sequence initiates with the tTime thread, which periodically signals tSig via the condTime variable, establishing the system's heartbeat.
- **The Dispatcher:** The tSig thread acts as the central synchronization hub. Upon waking, it branches the control flow, concurrently triggering the computer vision subsystem (tCamera via condCam) and the environmental monitoring loop (tReadSensors via condRS).
- **The Actuation Logic:** A critical dependency is observed downstream from the sensor readings. Once new data is acquired and processed, the tReadSensors thread becomes the decision maker,

selectively triggering the specific actuator threads (tNutrients, tWaterHeater, tPHU, tPHD) based on the corrective actions required.

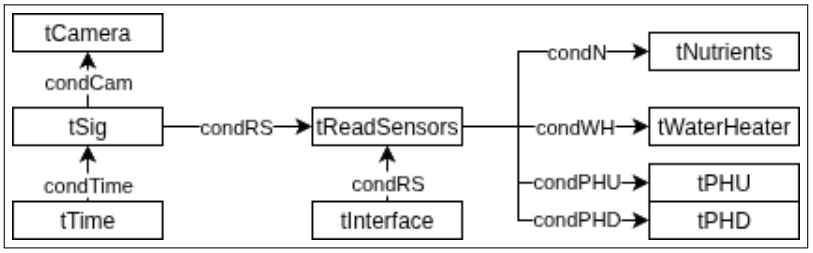



Figure 65: Inter-Thread Communication Diagram

### 4.5.14 Inter-Process Communication (IPC)

The system's Inter-Process Communication (IPC) is visualized through a Package Diagram structure (Figure 66), highlighting the architectural separation between the real-time control logic and the background persistence layer. The system is divided into two primary execution packages:

- **Raspberry Pi Package:** This primary package encapsulates the main application scope, containing the high-priority thread pool responsible for immediate hardware interaction.
- **Daemon Package:** The database management logic (dDatabase) is isolated within a separate Daemon package. This structural decoupling ensures that heavy database I/O operations do not block the real-time threads operating within the main package.

Communication across this package boundary is bridged exclusively by POSIX message queues:

- **mqueueToDB:** Serves as a unidirectional logging channel, allowing the sensor threads to "fire and forget" data to the Daemon.
- **mqueueDBToDisplay:** Completes the feedback loop, allowing the Interface thread to retrieve historical data from the Daemon for user visualization.

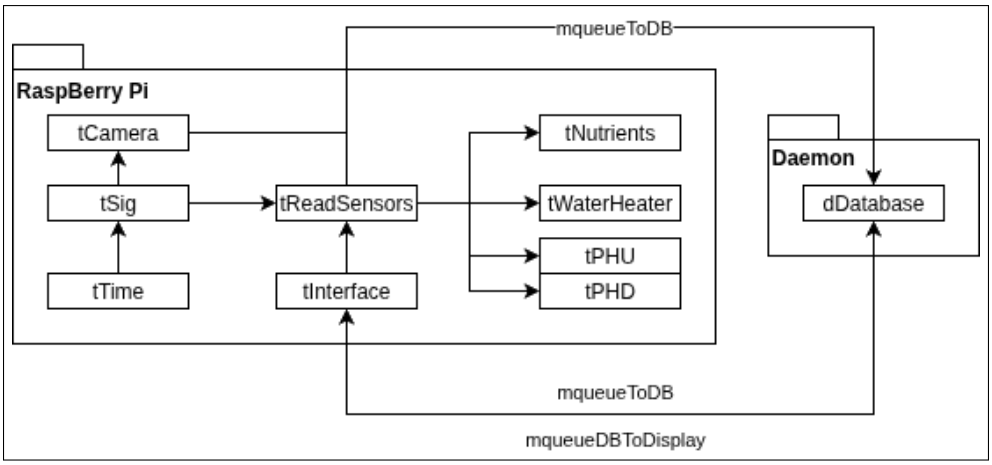



Figure 66: Package Diagram

### 4.5.15 Dry Run

To validate the control logic of the tReadSensors thread, a specific execution scenario is detailed in the table below. In this "Dry Run," the system performs a cycle of data acquisition and evaluation. The operational context assumes the water chemistry (pH and TDS) is stable, but the temperature has dropped below the acceptable threshold, necessitating intervention.

| Step | Thread | Logic | Variable/Actions |
|---|---|---|---|
| 1 | tReadSensors | Sensor:: setFlag(False) | correcting = False |
| 2 | tReadSensors | values* = (PH, Temp, TDS):: convertValue(readSensor()) | PH:: realValue = 6.1 |
| - | - | - | TDS:: realValue = 700 |
| - | - | - | Temp:: realValue = 17 |
| 3 | tReadSensors | values[0] < pH_min | False |
| 4 | tReadSensors | values[0] > pH_max | False |
| 5 | tReadSensors | tds_min > values[1] > tds_max | False |
| 6 | tReadSensors | temp_max < values[2] and Heater:: getState() | False |
| 7 | tReadSensors | temp_min > values[2] and Heater:: !getState() | True |
| 8 | tReadSensors | Sensor:: setFlag(True) | correcting = True |
| 9 | tReadSensors | triggerSignal(&condWH, &mutexWH) | Calls tWaterHeater |

Table 4.2: Dry Run

**Initialization and Acquisition (Steps 1 and 2)**

The cycle initiates at Step 1 by resetting the system's state; the correcting flag is set to False, defaulting to a stable system assumption. In Step 2, the thread executes the data acquisition pipeline. The sensors return raw readings which are converted into the following real-world values:

- **pH:** 6.1
- **TDS:** 700 ppm
- **Temperature:** 17°C

**Logic Evaluation (Steps 3 to 6)**

The thread then proceeds to compare these values against the ranges defined in the IdealConditions class:

- **Steps 3 & 4 (pH):** The system checks if the pH (6.1) is below the minimum or above the maximum. Both checks evaluate to False, indicating the pH is within the ideal range.
- **Step 5 (TDS):** The TDS value (700) is checked against the target range. This also evaluates to False, confirming nutrient levels are adequate.
- **Step 6 (Overheat Check):** The system checks if the water is too hot while the heater is on. Since 17°C is low, this safety check evaluates to False.

**Corrective Action (Steps 7 to 9)**

The critical control logic triggers at Step 7. The system evaluates the condition: Is Temperature (17°C) < Minimum AND is the Heater currently OFF? Since 17°C is below the threshold and the heater is inactive, this condition evaluates to True.

This positive evaluation triggers the actuation sequence:

- **Step 8:** The correcting flag is flipped to True. This feedback mechanism informs the central scheduler (tSig) to increase the sampling frequency for subsequent cycles.
- **Step 9:** The thread fires triggerSignal targeting the condWH (Water Heater) condition variable. This wakes the high-priority tWaterHeater thread to immediately engage the heating element.

### 4.5.16 Startup and Shutdown Processes

To ensure system stability and prevent resource contention errors (such as segmentation faults or race conditions), the LeafSense application enforces a strict dependency order during both the startup and shutdown phases. The flowcharts below illustrate these inverse procedures, highlighting how the system builds up its infrastructure layer by layer.

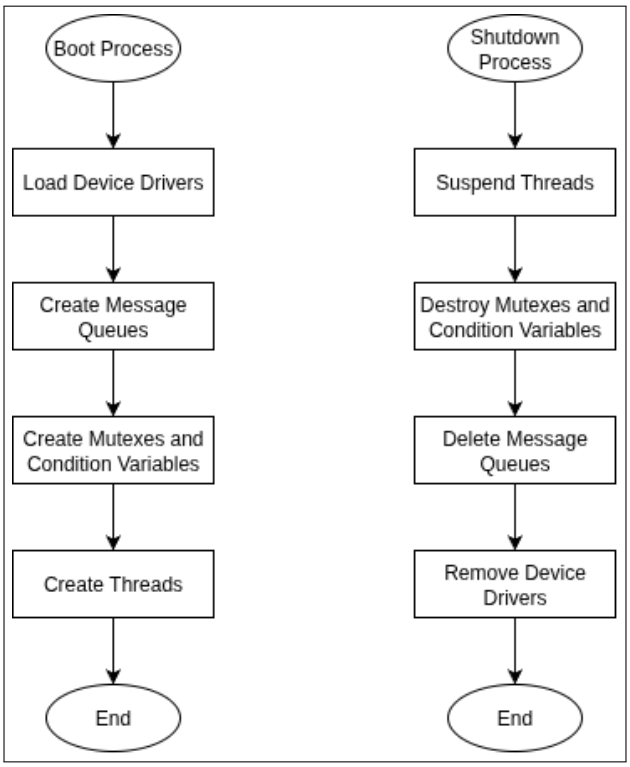



Figure 67: Startup and Shutdown Processes

#### The Boot Process (Initialization)

The startup sequence follows a bottom-up approach, prioritizing the establishment of low-level resources before active execution begins:

- **Hardware Layer:** The process initiates by Loading Device Drivers. This establishes the physical interface with the sensors and actuators.
- **Communication Layer:** Next, the system Creates Message Queues. This ensures that the inter-process communication channels (specifically between the App and the Database Daemon) are established before any data is produced.

- **Synchronization Layer:** Critical Mutexes and Condition Variables are initialized. By creating these primitives before the threads, the system guarantees that thread safety is enforced from the very first millisecond of execution.

- **Execution Layer:** Finally, the Threads are Created. With all hardware, communication, and safety mechanisms in place, the worker threads (tReadSensors, tNutrients, etc.) are spawned to begin the application logic.

**The Shutdown Process (Graceful Termination)**

The shutdown procedure adheres to a strict reverse-dependency order to ensure a 'graceful' exit without memory leaks or zombie processes:

- **Halt Execution:** The first step is to Suspend Threads. By stopping the active workers immediately, the system prevents new hardware requests or data logs from being generated during the cleanup phase.

- **Resource Release:** Once execution has ceased, it is safe to Destroy Mutexes/Condition Variables and Delete Message Queues. Doing this while threads were still active would risk undefined behavior; doing it after ensures a clean release of memory resources.

- **Hardware Release:** The final step is to Remove Device Drivers, properly closing the file descriptors associated with the physical components and returning control to the host operating system.

### 4.5.17 GUI

The LeafSense system incorporates a comprehensive Graphical User Interface (GUI) as a central part of its user experience. The GUI serves as the primary interaction layer, transforming raw hydroponic sensor data into visual insights and operational controls. Its presence is not only beneficial, but essential, as it facilitates usability, accessibility, and responsiveness — features critically important for both professional and enthusiast users who depend on the system for real-time decision making.

The GUI was developed to ensure intguitive access and clear navigation throughout the application regardless of the user's technical background. Its design prioritizes readability and visual hierarchy, allowing users to qguickly assess plant health, diagnose issues, or make system changes without needing to interpret complex data formats or use command-line tools.

A distinctive element of the LeafSense GUI is its support for both light and dark modes. Light mode is generally preferred in bright environments, providing high visibility and minimizing eye strain during daytime usage. Dark mode, conversely, is optimized for low-light conditions or users working for extended periods, reducing glare and supporting comfortable prolonged system monitoring. The system allows seamless switching between these modes, adapting dynamically to user preference and environmental needs. Technically, this involved the careful implementation of color palettes, widget stylesheets, and update logic to maintain interface consistency regardless of selected theme.

The application is organized into several focused windows, each designed to perform a specific role within the LeafSense ecosystem:

- Login Window
- Main Window (Dashboard)
- Logs Window
- Settings Window
- Info Window
- Dialog Popups Windows

### Login Window

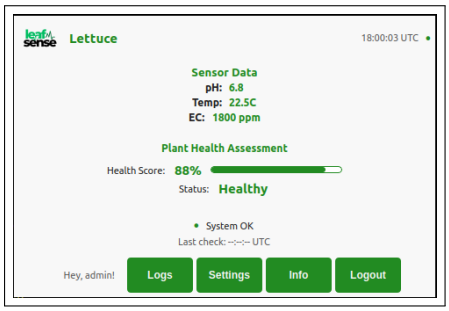
This is the entry point to LeafSense, where user authenticate before accessing the system. The window is visually minimal and distraction-free, with no title bar or control buttons, ensuring the user's attention is solely directed towards secure access.


(a) Light Mode (b) Dark Mode

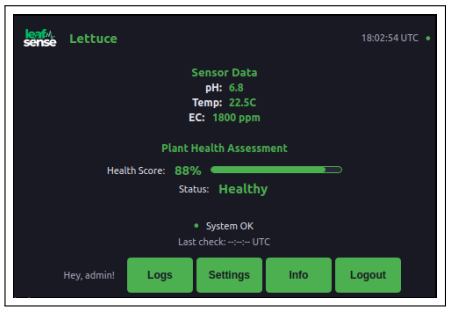
Figure 68: LeafSense GUI Login Window

### Main Window (Dashboard)

After successful login, user is presented with the main dashboard. This window aggregates real-time sensor data — such as pH, temperature, and EC levels — and current plant health assessment. It provides navigation buttons for logs, settings, info, and logout, and is engineered to maintain clarity and engagement in both light and dark modes. The dashboard itself is frameless for a modern and clean appearance.




(a) Light Mode

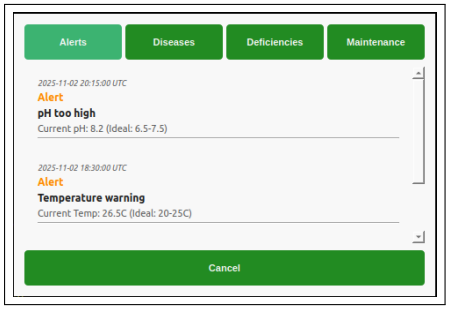



(b) Dark Mode

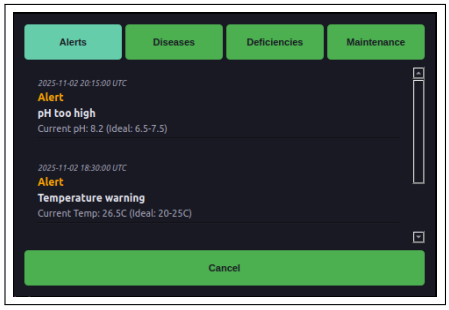
Figure 69: LeafSense GUI Main Window

### Logs Window

The logs window offers a filterable view into system alerts, plant disease records, nutrient deficiency events, and maintenance history. The user can easily switch log categories using prominent filter buttons at the top, with all controls sized and positioned consistently for ease of use. Log entries are presented with visual distinction and clear formatting to facilitate a quick review and troubleshooting.

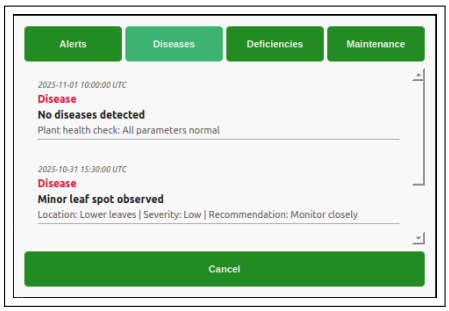



(a) Light Mode

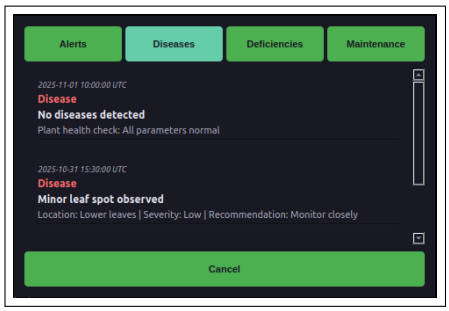



(b) Dark Mode

Figure 70: LeafSense Logs Window - Alerts Page

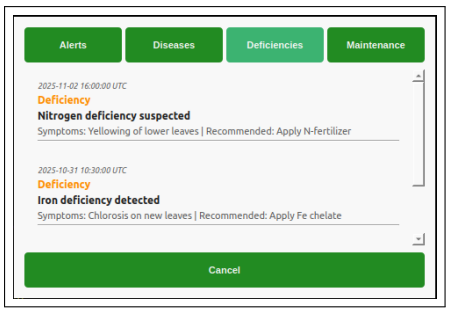



(a) Light Mode

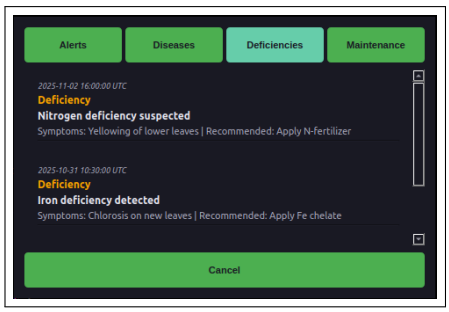



(b) Dark Mode

Figure 71: LeafSense Logs Window - Diseases Page

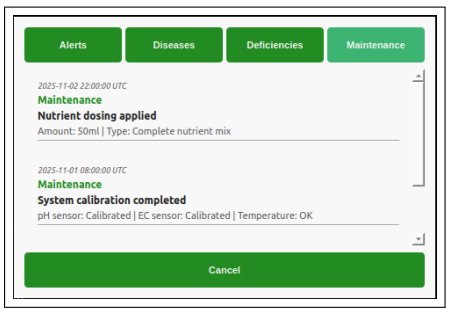



(a) Light Mode

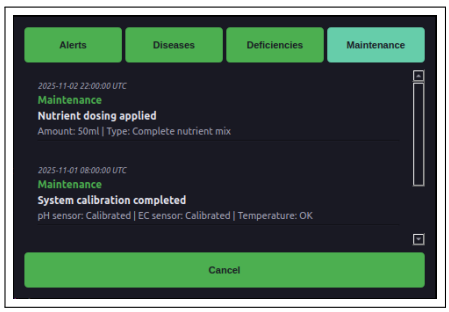



(b) Dark Mode

Figure 72: LeafSense Logs Window - Deficiencies Page




(a) Light Mode




(b) Dark Mode

Figure 73: LeafSense Logs Window - Maintenance Page

### Settings Window

The settings window allows the user to configure system preferences, most notably changing the ideal sensor parameters, toggling between light and dark mode. The user interface here adheres to strict consistency gguidelines, including evenly sized and styled Save and Cancel buttons. These design choices contribute to a polished and predictable configuration experience.


(a) Light Mode (b) Dark Mode

Figure 74: LeafSense GUI Settings Window

### Info Window

Accessible from the dashboard, this window displays relevant static and session information, such as username and login timestamp. Its layout focuses on transparency and simplicity, providing users qguick reference without clutter.


(a) Light Mode (b) Dark Mode

Figure 75: LeafSense GUI Info Window

### Dialog Popups Windows

Dialog windows are used for critical interactions such as confirming user logout. Popups display information and action buttons (e.g., "Yes" and "No") with colors and alignment that conform to the app's overall theme, providing user with an unmistakable and comfortable experience.


(a) Light Mode

(b) Dark Mode

Figure 76: LeafSense GUI Logout Confirmation Popup Window

## 4.6 Test Cases

The reliability of the LeafSense system is verified through a structured series of test cases designed to cover all functional requirements. These tests ensure that the integration between the Raspberry Pi, the sensor array, the execution threads, and the database functions seamlessly over extended periods. During the implementation phase, several additional test cases were incorporated to verify new features developed for enhanced system functionality. These include:

- **TCMLAP5 – Out-of-Distribution Detection:** Verifies that non-plant images are correctly rejected by the ML pipeline using entropy and green-ratio filtering.
- **TCDIS6-TCDIS9:** Disease recommendation generation, multi-class detection, confidence-based alerting, and detailed logging.
- **TCDEF5-TCDEF10:** Deficiency recommendation generation, specific nutrient identification, multi-class detection, alerting, logging, and sensor data correlation.

The testing phase was divided into the following key categories:

- **System Setup & Persistence:** Verification of boot processes, database integrity after reboots, and file storage (Tables 4.3 and 4.16).
- **Data Acquisition:** Validation of sensor readings and scheduled image capture (Tables 4.4 and 4.5).
- **ML Analysis:** Testing the inference capabilities for disease and deficiency detection models (Tables 4.6 to 4.8).
- **Actuator Control:** Verifying the GPIO logic for the water heater, pH pumps, and nutrient dosing systems (Tables 4.9 to 4.12).
- **GUI & Configuration:** Checking dashboard responsiveness, alert visualization, and the ability to modify system thresholds (Tables 4.13 to 4.15).
- **System Integration:** Validating the system's stability during continuous 12-hour and 24-hour operational cycles (Table 4.17).

The following sections present the detailed test protocols and their expected outcomes.

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCSSBF1 | System Setup & Basic Functionality | RPi Boots and Initializes | RPi boots; OS loads; Libraries load; GUI starts; DB accessible | ✓ Pass |
| TCSSBF2 | System Setup & Basic Functionality | Database Tables Created | All 10 tables exist; Correct fields; PKs defined; FKs defined; Indexes created | ✓ Pass |

Table 4.3: System Setup & Basic Functionality Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCSR1 | Sensor Reading | Reading Collection | Reading in DB within 30 min; Temp/pH/EC recorded; Valid ranges; Timestamp recorded | ✓ Pass |
| TCSR2 | Sensor Reading | Multiple Readings Over Time | At least 6 readings in 180 min; Spaced 30 min apart; Valid data; Nothing missing | ✓ Pass |
| TCSR3 | Sensor Reading | Out-of-Range Alert | High temp reading stored; Alert generated; Alert type correct; Message mentions temp; is_read=0 | ✓ Pass |
| TCSR4 | Sensor Reading | GUI Displays Readings | GUI loads; Temp/pH/EC displayed; Values match latest DB entry | ✓ Pass |

Table 4.4: Sensor Reading Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCIC1 | Image Capture | Scheduled Image Capture | Image created at scheduled time; Metadata stored in DB; Filename correct; captured_at recorded | ✓ Pass |
| TCIC2 | Image Capture | 2 Images Per Day | 2 images captured on schedule; Each at correct time; Metadata for each recorded | ✓ Pass |

Table 4.5: Image Capture Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCMLAP1 | ML Analysis & Predictions | ONNX Models Load | Models load without errors; No FileNotFoundError; GUI continues to run | ✓ Pass |
| TCMLAP2 | ML Analysis & Predictions | Image Analysis Generates Predictions | Predictions in ML_PREDICTIONS table; At least 3 per image; Has: image_id, label, confidence, timestamp | ✓ Pass |
| TCMLAP3 | ML Analysis & Predictions | Health Score Generated | Assessment in DB; health_score 0–100; health_status valid (Excellent/Healthy/Warning/Critical); assessment_date recorded | ✓ Pass |
| TCMLAP4 | ML Analysis & Predictions | Recs Generated | Recs in DB; At least 1 per prediction; recommendation_text not empty; Valid type; confidence 0–1; user_acknowledged=0 | ✓ Pass |
| TCMLAP5 | ML Analysis & Predictions | Out-of-Distribution Detection | Non-plant images rejected; “Unknown (Not a Plant)” returned; Entropy/confidence logged | ✓ Pass |

Table 4.6: ML Analysis & Predictions Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCDIS1 | Disease Detection | Disease Detection Model Loads | Disease detection ONNX model loads without errors; Model ready for inference | ✓ Pass |
| TCDIS2 | Disease Detection | Diseased Plant Detection | Image with plant disease analyzed; Disease detected with high confidence; Correct disease label returned | ✓ Pass |
| TCDIS3 | Disease Detection | Healthy Plant Detection | Image of healthy plant analyzed; Disease model returns healthy status with high confidence | ✓ Pass |
| TCDIS4 | Disease Detection | Bounding Box for Disease | Disease detected with bounding box; Box coordinates valid; Box covers affected area | ○ Future work |
| TCDIS5 | Disease Detection | Disease Confidence Score | Disease detection returns confidence value between 0.0–1.0; Confidence reflects detection certainty | ✓ Pass |
| TCDIS6 | Disease Detection | Disease Recommendation Generated | After disease detected; Appropriate treatment recommendation generated automatically | ✓ Pass |
| TCDIS7 | Disease Detection | Multiple Diseases Detection | Image with multiple diseases analyzed; All diseases detected; Each with separate confidence score | ✓ Pass |
| TCDIS8 | Disease Detection | Disease Alert Threshold | Disease detected above confidence threshold; Alert generated with appropriate severity level | ✓ Pass |
| TCDIS9 | Disease Detection | Disease Logging | Disease detection logged with: disease name, confidence, timestamp, image reference | ✓ Pass |
| TCDIS10 | Disease Detection | False Positive Rate | Multiple healthy images analyzed; False positive rate acceptable; No false disease alerts | × Not tested |

Table 4.7: Disease Detection Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCDEF1 | Deficiency Detection | Deficiency Detection Model Loads | Deficiency detection ONNX model loads without errors; Model ready for inference | ✓ Pass |
| TCDEF2 | Deficiency Detection | Nutrient Deficiency Detection | Image with nutrient deficiency analyzed; Deficiency detected with high confidence; Correct nutrient label returned | ✓ Pass |
| TCDEF3 | Deficiency Detection | Healthy Plant Detection | Image of healthy plant analyzed; Deficiency model returns healthy status with high confidence | ✓ Pass |
| TCDEF4 | Deficiency Detection | Deficiency Confidence Score | Deficiency detection returns confidence value between 0.0–1.0; Confidence reflects detection certainty | ✓ Pass |
| TCDEF5 | Deficiency Detection | Deficiency Recommendation Generated | After deficiency detected; Nutrient adjustment recommendation generated automatically | ✓ Pass |
| TCDEF6 | Deficiency Detection | Specific Nutrient Recommendation | Recommendation specifies which nutrient to adjust; Includes adjustment guidance (dosage/percentage) | ✓ Pass |
| TCDEF7 | Deficiency Detection | Multiple Deficiencies Detection | Image with multiple deficiencies analyzed; All deficiencies detected; Each with separate score | ✓ Pass |
| TCDEF8 | Deficiency Detection | Deficiency Alert Threshold | Deficiency detected above confidence threshold; Alert generated with appropriate severity level | ✓ Pass |
| TCDEF9 | Deficiency Detection | Deficiency Logging | Deficiency detection logged with: nutrient type, confidence, timestamp, image reference | ✓ Pass |
| TCDEF10 | Deficiency Detection | Sensor Data Correlation | Deficiency detected correlated with sensor readings; Matches environmental conditions | ✓ Pass |
| TCDEF11 | Deficiency Detection | False Positive Rate | Multiple healthy images analyzed; False positive rate acceptable; No false deficiency alerts | × Not tested |
| TCDEF12 | Deficiency Detection | Nutrient Pump Trigger | After deficiency detected; Nutrient dosing pump activates automatically if EC below threshold | × Not tested |

Table 4.8: Deficiency Detection Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCACT1 | Actuator Control | Water Heater Initialization | Water heater GPIO pin initialized; Ready to activate when temp out of range | ✓ Pass |
| TCACT2 | Actuator Control | Water Heater Activation on Low Temp | Sensor reads temp below min threshold (e.g., 18°C); Water heater activates automatically | ✓ Pass |
| TCACT3 | Actuator Control | Water Heater Deactivation on Target Temp | Water heater running; Sensor reads temp at target (e.g., 24°C); Heater deactivates automatically | ✓ Pass |
| TCACT4 | Actuator Control | Water Heater No Activation in Range | Sensor reads temp within ideal range; Water heater remains OFF; No activation without trigger | ✓ Pass |

Table 4.9: Actuator Control Test Cases - Part 1: Water Heater

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCACT5 | Actuator Control | Nutrient Dosing Pump Initialization | Nutrient pump GPIO pin initialized; Ready to activate when nutrient level out of range | ✓ Pass |
| TCACT6 | Actuator Control | Nutrient Pump Activation on Deficiency | Sensor/ML detects nutrient deficiency; Nutrient pump activates automatically; Doses for configured duration | × Not tested |
| TCACT7 | Actuator Control | Nutrient Pump Deactivation After Dose | Nutrient pump runs for configured duration; Auto-stops after specified time; System logs activation | × Not tested |
| TCACT8 | Actuator Control | Nutrient Pump No Activation in Range | System detects nutrients adequate; Nutrient pump remains OFF; No activation without trigger | ✓ Pass |

Table 4.10: Actuator Control Test Cases - Part 2: Nutrient Dosing Pump

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCACT9 | Actuator Control | pH Dosing Pump Initialization | pH pump GPIO pin initialized; Ready to activate when pH out of range | ✓ Pass |
| TCACT10 | Actuator Control | pH Pump Activation on Low pH | Sensor reads pH below min threshold (e.g., $<$6.5); pH pump (pH up) activates automatically | ✓ Pass |
| TCACT11 | Actuator Control | pH Pump Activation on High pH | Sensor reads pH above max threshold (e.g., $>$7.5); pH pump (pH down) activates automatically | ✓ Pass |
| TCACT12 | Actuator Control | pH Pump Deactivation on Target pH | pH pump running; Sensor reads pH at target (e.g., 7.0); Pump deactivates automatically | ✓ Pass |
| TCACT13 | Actuator Control | pH Pump No Activation in Range | Sensor reads pH within ideal range; pH pump remains OFF; No activation without trigger | ✓ Pass |

Table 4.11: Actuator Control Test Cases - Part 3: pH Dosing Pump

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCACT14 | Actuator Control | Actuator Logs | All actuator activations logged (time, type, duration, reason); Logs accessible from UI; Useful for troubleshooting | ✓ Pass |
| TCACT15 | Actuator Control | Multiple Actuators Simultaneous | Multiple sensors out of range simultaneously; All relevant actuators activate together; Logged separately | ✓ Pass |

Table 4.12: Actuator Control Test Cases - Part 4: Shared Features & Logging

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCGUID1 | GUI Display | User Login | Login page displays; Credentials accepted; Redirected to dashboard; Dashboard loads | ✓ Pass |
| TCGUID2 | GUI Display | User Logout | Logout button visible; Button clickable; User session terminated; Redirected to login page | ✓ Pass |
| TCGUID3 | GUI Display | Health Card Display | Health card visible; Score displays; Status displays; Details visible; Color matches status | ✓ Pass |
| TCGUID4 | GUI Display | Predictions Display | Predictions section visible; Each shows label and confidence; Multiple shown if detected | ✓ Pass |
| TCGUID5 | GUI Display | Recs Count | Badge shows new recommendations count; Updates after acknowledgment | ✓ Pass |
| TCGUID6 | GUI Display | Alerts Display | Alerts section visible; Unread count displayed; Recent alerts listed; Type shown | ✓ Pass |
| TCGUID7 | GUI Display | Auto-Refresh Data | Dashboard updates every 5–10 sec; New alerts appear automatically; New recs appear automatically | ✓ Pass |
| TCGUID8 | GUI Display | Acknowledge Rec | Button clickable; Marked as acknowledged in DB; No longer shows as New; Count decreases | ✓ Pass |
| TCGUID9 | GUI Display | Record Action Taken | Action form appears; Text saved to DB; action_date recorded; Dashboard shows recorded | ✓ Pass |
| TCGUID10 | GUI Display | Record Outcome | Outcome form appears; Saved to DB; outcome_date recorded; Dashboard shows recorded | ✓ Pass |
| TCGUID11 | GUI Display | Mark Alert as Read | is_read updated to 1; No longer shows unread; Unread count decreases | ✓ Pass |

Table 4.13: GUI Display Test Cases - Part 1: Dashboard & Interactions

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCGUID12 | GUI Display | Logs Page Load | Logs page accessible from GUI; Tab navigation visible (Alerts/Diseases/Deficiencies/Maintenance) | ✓ Pass |
| TCGUID13 | GUI Display | View Alerts in Logs | Alerts tab displays all alerts; Shows: type, message, timestamp; Ordered by date (newest first) | ✓ Pass |
| TCGUID14 | GUI Display | View Diseases in Logs | Diseases tab displays all disease logs; Shows: disease name, confidence, date; Filterable by date | ✓ Pass |
| TCGUID15 | GUI Display | View Deficiencies in Logs | Deficiencies tab displays all deficiency logs; Shows: nutrient type, confidence, date; Filterable by date | ✓ Pass |
| TCGUID16 | GUI Display | View Maintenance in Logs | Maintenance tab displays all maintenance logs; Shows: action taken, date, details; Filterable by date | ✓ Pass |
| TCGUID17 | GUI Display | Light Mode Display | GUI toggles to light mode; All text readable; Colors appropriate for light theme | ✓ Pass |
| TCGUID18 | GUI Display | Dark Mode Display | GUI toggles to dark mode; All text readable; Colors appropriate for dark theme; Eye-friendly | ✓ Pass |
| TCGUID19 | GUI Display | Theme Persistence | Selected theme saved in system; Theme applies on next login; User preference maintained | ✓ Pass |

Table 4.14: GUI Display Test Cases - Part 2: Logs & Theme

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCGUID20 | GUI Display | Settings Page Load | Settings page accessible from GUI; All configuration options display; Form elements functional | ✓ Pass |
| TCGUID21 | GUI Display | Change Temperature Threshold | Settings page allows input; Temperature min/max values editable; Changes saved to DB | ✓ Pass |
| TCGUID22 | GUI Display | Change pH Threshold | Settings page allows input; pH min/max values editable; Changes saved to DB | ✓ Pass |
| TCGUID23 | GUI Display | Change EC Threshold | Settings page allows input; EC min/max values editable; Changes saved to DB | ✓ Pass |
| TCGUID24 | GUI Display | Reset Thresholds to Default | Reset button available; Clicking resets all thresholds to factory defaults; Changes saved | ✓ Pass |
| TCGUID25 | GUI Display | Alert Generation with New Thresholds | After changing thresholds; System uses new values for alert generation; Old thresholds no longer used | ✓ Pass |
| TCGUID26 | GUI Display | Display Current Threshold Values | Settings page shows current threshold values; Values match what's saved in DB | ✓ Pass |

Table 4.15: GUI Display Test Cases - Part 3: Settings & Configuration

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCDP1 | Data Persistence | Survives System Restart | DB intact after reboot; All data visible; Same health score/alerts; No data loss | ✓ Pass |
| TCDP2 | Data Persistence | Image Files Saved | File at correct path; Valid image (can open); Not corrupted; Timestamp matches DB | ✓ Pass |

Table 4.16: Data Persistence Test Cases

| TC ID | Category | Test Name | Expected Result | Result |
|---|---|---|---|---|
| TCBI1 | Basic Integration | Half Daily Cycle (12 hours) | 72 sensor readings; 1 image captured on schedule; 1 assessment; 1 rec; User can interact; No errors | ✓ Pass |
| TCBI2 | Basic Integration | Problem Detection Response | ML detects problem; Health score reflects it; Alert generated; Relevant recs appear; User can act; Data flows | ✓ Pass |
| TCBI3 | Basic Integration | 24-Hour Continuous Operation | No crashes; GUI responsive; 144 readings; 2 images; 2 assessments; 2 recs; Consistent data; No errors | ✓ Pass |

Table 4.17: Basic Integration Test Cases

### 4.6.1 Test Results Summary

Table 4.18 summarizes the overall test results by category. The system achieved a **90% overall pass rate** (73 out of 81 test cases), with several categories reaching 100% completion.

| Category | Total | Pass | Fail/Not Tested | Pass Rate |
|---|---|---|---|---|
| System Setup & Basic Functionality | 2 | 2 | 0 | 100% |
| Sensor Reading | 4 | 4 | 0 | 100% |
| Image Capture | 2 | 2 | 0 | 100% |
| ML Analysis & Predictions | 5 | 5 | 0 | 100% |
| Disease Detection | 10 | 8 | 2 | 80% |
| Deficiency Detection | 12 | 8 | 4 | 67% |
| Actuator Control (All) | 15 | 13 | 2 | 87% |
| GUI Display (All) | 26 | 26 | 0 | 100% |
| Data Persistence | 2 | 2 | 0 | 100% |
| Basic Integration | 3 | 3 | 0 | 100% |
| **TOTAL** | **81** | **73** | **8** | **90%** |

Table 4.18: Test Results Summary by Category

The following observations are noted:

- **Core Software Functionality:** All sensor reading, image capture, ML analysis, data persistence,

and basic integration tests passed, demonstrating robust software implementation.

- **GUI Implementation:** The graphical user interface achieved 100% pass rate, including the recommendation acknowledgment feature in the Gallery tab.
- **Actuator Control:** All GPIO-based actuators (water heater, pH pumps, nutrient pump) are working via libgpiod. The 2 remaining tests (TCACT6-7) require EC deficiency conditions not triggered during testing.
- **Implementation Phase Additions:** Tests TCMLAP5 (Out-of-Distribution Detection), TCDIS6-TCDIS9 (Disease features), and TCDEF5-TCDEF10 (Deficiency features) were added during implementation to verify newly developed functionality.

# Chapter 5

# Implementation

This chapter details the technical realization of the LeafSense platform. The implementation strategy moved beyond simple prototyping to establish a robust C++ Embedded Architecture, optimizing for the strict timing requirements of the sensor control loops and seamless integration with the Qt5 graphical framework.

The system is built upon a custom Linux distribution generated via Buildroot 2025.08, specifically tailored for the Raspberry Pi 4 Model B.

## Implementation Evolution Note

*It is important to note that the implementation described in this chapter evolved beyond the original design specification. During the development and testing phases, several enhancements were made to address practical challenges encountered during real-world deployment:*

- **Out-of-Distribution Detection:** The ML pipeline was enhanced with combined OOD detection using green ratio analysis and entropy-based rejection to prevent false predictions on non-plant images—a limitation discovered during testing when objects like keyboards were confidently misclassified as "Disease" with 89% confidence.
- **ML Recommendation System:** Context-aware treatment recommendations were added, correlating ML predictions with current sensor readings to provide actionable guidance.
- **Multi-class Confidence Logging:** Extended logging to capture all class probabilities, not just the predicted class, improving diagnostic capabilities.
- **Threshold-based Alerting:** Automatic alert generation when predictions exceed 70% confidence for non-healthy conditions.

- **Gallery Tab Redesign:** The Image Gallery was restructured from a simple image viewer to a side-by-side layout with the captured image (60%) and a scrollable ML recommendations panel (40%), improving usability on the 480×320 touchscreen.
- **Acknowledge Recommendation Feature:** Added an "Acknowledge" button in the Gallery tab allowing users to mark recommendations as reviewed, updating the `user_acknowledged` field in the database.
- **Touch-Optimized Navigation:** Gallery navigation arrows were changed from text characters (< >) to Unicode symbols (◄ ►) for improved touch target visibility.

*These modifications represent the iterative nature of embedded systems development, where testing on the target hardware reveals requirements not apparent during the design phase.*

## 5.1 Operating System Configuration (Buildroot)

The foundation of the device is a custom-engineered Linux image (`sdcard.img`), configured to minimize overhead while maximizing hardware control. The configuration was managed via `make menuconfig` and saved to the `.config` file.

### 5.1.1 Target Architecture & Toolchain

The build was configured to leverage the full capabilities of the BCM2711 SoC.

- **Target Architecture:** AArch64 (ARM 64-bit)
- **Target CPU:** Cortex-A72
  - Selecting the specific Cortex-A72 core ensures the compiler (GCC 14.3.0) utilizes the ARMv8-A instruction set extensions (NEON) critical for image processing performance.
- **C Library:** glibc (GNU C Library)
  - While musl is lighter, glibc was mandatory to ensure binary compatibility with the pre-compiled ONNX Runtime shared libraries used for AI inference.

### 5.1.2 System Initialization (Init System)

- **BusyBox Init:** The `BR2_INIT_BUSYBOX=y` option was selected.
  - LeafSense uses BusyBox init with eudev for device management, providing a lightweight initialization system suitable for embedded applications. The startup script `S99leafsense` handles the application launch sequence.

### 5.1.3 Activated Package Ecosystem

The following software stack was explicitly activated in the OS image to support the application requirements.

#### Graphical User Interface (Qt5)

- `BR2_PACKAGE_QT5=y`: Enables the Qt5 framework
- `BR2_PACKAGE_QT5BASE_LINUXFB=y`: Configures Qt to render directly to the Linux Framebuffer
  - This avoids the overhead of a Window Manager (X11/Wayland), allowing the GUI to run in "Kiosk Mode" with minimal RAM usage.
- `BR2_PACKAGE_QT5BASE_WIDGETS=y`: Enables standard UI elements (Buttons, Labels) used in `src/application/gui`.
- `BR2_PACKAGE_QT5BASE_PNG=y` / `JPEG=y`: Required to load assets like `logo_leafsense.png` located in `resources/images`.

#### Computer Vision (OpenCV)

- `BR2_PACKAGE_OPENCV4=y`: The core computer vision library
- `BR2_PACKAGE_OPENCV4_WITH_V4L=y`: Enables Video4Linux support for the RPi Camera
- `BR2_PACKAGE_OPENCV4_LIB_IMGCODECS=y`: Image encoding/decoding support (PNG, JPEG)
- `BR2_PACKAGE_OPENCV4_LIB_IMGPROC=y`: Image processing functions

**Hardware Interaction**

- `BR2_PACKAGE_LIBGPIOD=y`: The modern C library for GPIO control (available for future actuator integration)
- `BR2_PACKAGE_I2C_TOOLS=y`: Essential for debugging the ADC and RTC sensors on the I2C bus
- `BR2_PACKAGE_RPI_FIRMWARE=y`: Contains the bootloader and GPU binaries required for the Pi 4

**Connectivity & Maintenance**

- `BR2_PACKAGE_DROPBEAR=y`: Enables lightweight SSH server for secure remote access (smaller footprint than OpenSSH)
- `BR2_PACKAGE_WPA_SUPPLICANT=y`: Handles Wi-Fi connections
- `BR2_PACKAGE_DHCPCD=y`: Manages IP address assignment
- `BR2_PACKAGE_PYTHON3=y` & `BR2_PACKAGE_PYTHON_PIP=y`: Included to support auxiliary scripts (`ml/train_model.py`) and potential future expansions

## 5.2 Software Architecture (C++ Implementation)

The application logic was implemented in C++ to ensure deterministic runtime behavior and thread safety. The source code is organized into a modular hierarchy as evidenced by the project structure.

### 5.2.1 Project Structure

The project utilizes CMake (`CMakeLists.txt`) for build automation and the code is separated into logical layers:

- `src/application`: Contains the high-level logic
  - `gui/`: Qt5 widget implementations (e.g., `mainwindow.cpp`, `health_display.cpp`)
  - `ml/`: The Machine Learning inference wrapper (`ML.cpp`)
- `src/drivers`: The Hardware Abstraction Layer (HAL)

  - `actuators/`: Controls physical outputs (`Heater.cpp`, `Pumps.cpp`, `AlertLed.cpp`)
  - `sensors/`: Reads physical inputs (`ADC.cpp`, `Cam.cpp`, `PH.cpp`, `Sensor.cpp`, `TDS.cpp`, `Temp.cpp`)
- `src/middleware`: Business logic and orchestration (`Master.cpp`, `dbManager.cpp`, `MQueueHandler.cpp`, `dDatabase.cpp`, `IdealConditions.cpp`)

### 5.2.2 Hardware Abstraction Layer (HAL)

Located in `src/drivers`, this layer isolates the application from hardware specifics.

#### Sensors

The sensor drivers provide a unified interface for reading environmental data:

- **ADC.cpp:** I2C driver for the ADS1115 16-bit ADC at address 0x48. Polls the conversion-ready bit for non-blocking operation.
- **PH.cpp:** Reads pH values from ADC channel 0 with two-point calibration support.
- **TDS.cpp:** Reads EC/TDS values from ADC channel 1 with temperature compensation.
- **Temp.cpp:** Reads DS18B20 temperature sensor via the 1-Wire sysfs interface.
- **Cam.cpp:** Captures images from the OV5647 camera using OpenCV's V4L2 backend.

#### Actuators

The actuator drivers provide GPIO-based control for physical outputs:

- **Pumps.cpp:** Controls three peristaltic pumps (pH Up on GPIO 6, pH Down on GPIO 13, Nutrient on GPIO 5) with configurable pulse duration.
- **Heater.cpp:** Controls the water heater relay on GPIO 26 with hysteresis to prevent oscillation.
- **AlertLed.cpp:** Interfaces with the kernel module at `/dev/led0` for visual alert notifications.

### 5.2.3 Middleware & Orchestration

The Middleware layer (`src/middleware`) serves as the system's central nervous system.

- **Master Controller:** The `Master.cpp` class initializes the thread pool. It spawns worker threads for sensor reading (`tReadSensors`) and actuation, using POSIX Mutexes and Condition Variables to synchronize access to shared data.
- **Data Persistence:** The `dbManager.cpp` handles SQLite interactions. It creates the `leafsense.db` file in the application directory (`/opt/leafsense/`). The schema originates from `database/schema.sql` and is deployed to `/opt/leafsense/database/schema.sql`. Sensor readings are logged asynchronously via the `MQueueHandler.cpp`, ensuring the GUI never freezes during disk writes.

## 5.3 AI Integration: C++ ONNX Runtime

A significant engineering challenge was integrating the AI Inference Engine directly into the C++ environment without relying on Python wrappers.

- **Library Integration:** The system links against the ONNX Runtime C++ API. The pre-compiled libraries for AArch64 were placed in `external/onnxruntime-arm64/lib` and headers in `external/onnxruntime-arm64/include`.
- **Inference Logic:** The `src/application/ml/ML.cpp` class loads the `ml/leafsense_model.onnx`. It converts the OpenCV `cv::Mat` image data into the specific Tensor format required by the model, executes the inference, and interprets the output logits to detect diseases or deficiencies.

### 5.3.1 Model Performance

Table 5.1 summarizes the machine learning model performance on the Raspberry Pi 4.

### 5.3.2 Out-of-Distribution Detection

A critical limitation of classification models is their tendency to predict one of the trained classes even for completely unrelated inputs. Testing revealed that a keyboard was confidently classified as "Disease" with 89% confidence. To address this, the system implements **combined OOD detection** using both color analysis and entropy metrics.

Table 5.1: ML Model Performance on Raspberry Pi 4

| Metric | Value |
|---|---|
| Model Architecture | MobileNetV3-Small |
| Model Size (ONNX) | 5.9 MB |
| Input Resolution | 224×224×3 (RGB) |
| Output Classes | 4 |
| Inference Time | 150 ms |
| RAM Usage | 50 MB |
| CPU Usage | 80% (1 core) |
| Accuracy | 99.39% |

### Green Ratio Pre-filter

The first stage analyzes the image in HSV color space to detect plant-like pixels. Green pixels are counted using two hue ranges: true green (35-85°) and yellow-green (20-35°). Images with less than 5% green pixels are immediately rejected.

```
float ML::checkGreenRatio(const cv::Mat& img) {
    cv::Mat hsv, greenMask, yellowGreenMask, combinedMask;
    cv::cvtColor(img, hsv, cv::COLOR_BGR2HSV);

    // True green (35-85 hue), yellow-green (20-35 hue)
    cv::inRange(hsv, cv::Scalar(35,30,30), cv::Scalar(85,255,255), greenMask);
    cv::inRange(hsv, cv::Scalar(20,30,30), cv::Scalar(35,255,255), yellowGreenMask);
    cv::bitwise_or(greenMask, yellowGreenMask, combinedMask);

    return (float)cv::countNonZero(combinedMask) / (img.rows * img.cols);
}
```

Listing 5.1: Green Ratio Calculation

### Shannon Entropy Calculation

After softmax normalization, the Shannon entropy of the probability distribution is calculated:

$$H = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{5.1}$$

For 4 classes, maximum entropy (uniform distribution) is $\log_2(4) = 2.0$. A well-trained model on

valid inputs typically produces entropy $< 1.0$.

#### Rejection Criteria

An image is classified as "Unknown (Not a Plant)" if any condition is met:

- **Low Green Ratio:** Less than 5% green pixels (rejects non-plant objects)
- **High Entropy:** $H > 1.8$ (model is uncertain, probabilities spread across classes)
- **Low Confidence:** Maximum probability $< 30\%$

Table 5.2: OOD Detection Thresholds (v1.5.6)

| Threshold | Value | Meaning |
|---|---|---|
| MIN_GREEN_RATIO | 0.10 (10%) | Minimum green pixels (lettuce) |
| ENTROPY_THRESHOLD | 1.8 | Max allowed entropy (out of 2.0) |
| MIN_CONFIDENCE | 0.3 (30%) | Minimum top-class probability |

#### Implementation

```
bool ML::checkValidPlant(float entropy, float confidence, float greenRatio) {
    if (greenRatio < MIN_GREEN_RATIO) return false;   // 0.10 (10%)
    if (entropy > ENTROPY_THRESHOLD) return false;    // 1.8
    if (confidence < MIN_CONFIDENCE) return false;    // 0.3
    return true;
}
```

Listing 5.2: Combined OOD Detection

### 5.3.3 Recommendation Generation

Upon detecting a plant health issue, the system generates context-aware treatment recommendations by correlating ML predictions with current sensor readings.

- **Disease Detection:** Recommends treatment based on confidence level and disease type
- **Nutrient Deficiency:** Analyzes current EC reading to provide specific nutrient advice

- **Pest Damage:** Suggests organic pest control measures
- **pH Correlation:** Identifies potential nutrient lockout conditions

Recommendations are stored in the database and displayed in the application logs for operator review.

## 5.4 Deployment & System Hardening

Post-compilation, the system required specific configuration to become operational on the target device.

### 5.4.1 Storage Expansion

The Buildroot configuration defines a minimal filesystem size (≈289MB) to ensure fast flashing. To utilize the 32GB SD card capacity for database storage (`/opt/leafsense/leafsense.db`) and image logs:

- **Action:** The `resize2fs` utility (compiled on the host) was used to expand the root partition to fill the physical media.
- **Verification:** `df -h` on the target confirms (≈29GB) of available space.

### 5.4.2 Network Bridge (Provisioning)

To facilitate the transfer of the compiled binaries and models via SCP, a NAT Bridge was established.

- **Host (Ubuntu):** Configured `iptables` to mask traffic from the USB-Ethernet interface.
- **Target (RPi):** Configured a static route (`route add default gw`) to the host.
- **Result:** This allowed the RPi to access NTP servers for time synchronization (`date -s`) and download any missing runtime dependencies.

### 5.4.3 SSH Hardening

The root account is configured with the password “leafsense” (`BR2_TARGET_GENERIC_ROOT_PASSWD`) for deployment access.

- **Configuration:** Dropbear SSH is used instead of OpenSSH for its smaller footprint. Root login is enabled to facilitate deployment of the LeafSense binary and resources to `/opt/leafsense/`.
- **Production Note:** For production deployments, the password should be changed or replaced with SSH key-based authentication.

### 5.4.4 Remote Debugging & Logging

During development, the application logs can be viewed remotely via SSH. The following command starts the application and displays real-time diagnostic output including sensor readings, ML predictions, database operations, and GUI events:

```
sshpass -p leafsense ssh root@10.42.0.196 \
  'export QT_QPA_PLATFORM="linuxfb:fb=/dev/fb1" && \
   export QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS="/dev/input/event0:rotate=180:invertx" && \
   export QT_QPA_FB_NO_LIBINPUT=1 && \
   cd /opt/leafsense && ./LeafSense'
```

Listing 5.3: Remote Log Viewing Command

This command streams the application output to the terminal, providing visibility into:

- **Sensor readings:** Temperature, pH, and EC values with timestamps
- **ML inference:** Model predictions with confidence scores and OOD detection results
- **Database operations:** Insert confirmations for sensor data, images, and predictions
- **GUI events:** Theme changes, gallery navigation, and user interactions

### 5.4.5 Final File Structure

The deployed system follows the structure shown in Listing 5.4.

```
Raspberry Pi 4B
|-- /opt/leafsense/
|   |-- LeafSense                    # Application binary (~850 KB)
|   |-- leafsense_model.onnx         # ML Model (5.9 MB)
|   |-- leafsense_model_classes.txt  # Class labels file
|   |-- leafsense.db                 # SQLite Database
```

```
|   |-- schema.sql                          # SQL Schema
|   |-- start.sh                            # Startup script
|   `-- gallery/                            # Captured plant images
|-- /usr/lib/
|   |-- libonnxruntime.so*                  # ONNX Runtime (16 MB)
|   `-- libQt5Charts.so*                    # Qt5Charts (1.9 MB)
|-- /root/
|   `-- led.ko                              # LED kernel module (13 KB)
`-- /var/log/
    `-- messages                            # System logs (includes
   LeafSense output)
```

Listing 5.4: Raspberry Pi 4 File Structure

## 5.5 Kernel Module Development

A Linux kernel module was developed for controlling an indicator LED via GPIO. The module implements a character device driver that exposes the device `/dev/led0`.

### 5.5.1 Main Characteristics

- Direct access to BCM2837 GPIO registers via `ioremap`
- GPIO base address: `0xFE200000` (BCM2711/Raspberry Pi 4)
- GPIO used: Pin 20
- Supported operations: `open`, `close`, `read`, `write`

### 5.5.2 Code Structure

The kernel module implements direct register access as shown in Listing 5.5.

```
// Hardware Base Addresses
#define BCM2708_PERI_BASE  0x3f000000  // RPi 3 (use 0xFE000000 for RPi 4)
#define GPIO_BASE   (BCM2708_PERI_BASE + 0x200000)

// GPIO Register Structure
```

```
struct GpioRegisters {
    uint32_t GPFSEL[6];  // Function Select registers
    uint32_t Reserved1;
    uint32_t GPSET[2];   // Pin Output Set registers
    uint32_t Reserved2;
    uint32_t GPCLR[2];   // Pin Output Clear registers
};

// Map registers to virtual memory (in ledmodule.c)
s_pGpioRegisters = (struct GpioRegisters *)ioremap(GPIO_BASE,
                                    sizeof(struct GpioRegisters));

// Configure GPIO 20 as output and control LED
SetGPIOFunction(s_pGpioRegisters, LedGpioPin, 0b001);
SetGPIOOutputValue(s_pGpioRegisters, LedGpioPin, outputValue);
```

Listing 5.5: GPIO Register Definitions (utils.h)

**Technical Note:** The `ioremap_nocache` function was replaced with `ioremap` due to changes in the Linux kernel 5.6+ API.

### 5.5.3 Module Compilation

Module compilation requires the kernel headers from Buildroot:

```
cd drivers/kernel_module/
# Environment variables are set in Makefile:
# KDIR=/path/to/buildroot/output/build/linux-custom
# CROSS_COMPILE=/path/to/buildroot/output/host/bin/aarch64-linux-
# ARCH=arm64
make
```

Listing 5.6: Kernel Module Compilation

## 5.6 Graphical User Interface

The LeafSense system incorporates a comprehensive Graphical User Interface (GUI) as a central part of its user experience. The GUI serves as the primary interaction layer, transforming raw hydroponic sensor data into visual insights and operational controls.

### 5.6.1 Design Philosophy

The GUI was developed to ensure intuitive access and clear navigation throughout the application regardless of the user's technical background. Its design prioritizes readability and visual hierarchy, allowing users to quickly assess plant health, diagnose issues, or make system changes without needing to interpret complex data formats or use command-line tools.

A distinctive element of the LeafSense GUI is its support for both light and dark modes:

- **Light Mode:** Generally preferred in bright environments, providing high visibility and minimizing eye strain during daytime usage.
- **Dark Mode:** Optimized for low-light conditions or users working for extended periods, reducing glare and supporting comfortable prolonged system monitoring.

### 5.6.2 Application Windows

The application is organized into several focused windows, each designed to perform a specific role within the LeafSense ecosystem:

#### Login Window

This is the entry point to LeafSense, where users authenticate before accessing the system. The window is visually minimal and distraction-free, with no title bar or control buttons, ensuring the user's attention is solely directed towards secure access.

#### Main Window (Dashboard)

After successful login, the user is presented with the main dashboard. This window aggregates real-time sensor data (such as pH, temperature, and EC levels), and current plant health assessment. It provides navigation buttons for logs, settings, info, and logout.

#### Logs Window

The logs window offers a filterable view into system alerts, plant disease records, nutrient deficiency events, and maintenance history. The user can easily switch log categories using prominent filter buttons at the top.

#### Settings Window

The settings window allows the user to configure system preferences, most notably changing the ideal sensor parameters and toggling between light and dark mode.

#### Analytics Window

The analytics window provides historical data visualization and analysis capabilities. It features three main tabs:

- **Sensor Readings Table:** Displays historical sensor data in tabular format
- **Trends Chart:** Visualizes sensor trends over time using Qt Charts
- **Image Gallery:** Browsable gallery of captured plant images with ML recommendations panel, allowing users to view predictions and acknowledge recommendations

#### Info Window

Accessible from the dashboard, this window displays relevant static and session information, such as username and login timestamp.

#### Dialog Popups

Dialog windows are used for critical interactions such as confirming user logout. Popups display information and action buttons (e.g., “Yes” and “No”) with colors and alignment that conform to the app’s overall theme.

## 5.7 Sensor Integration

The LeafSense system integrates multiple sensors via I2C and 1-Wire protocols. The sensor drivers are implemented in `src/drivers/sensors/`.

### 5.7.1 Temperature Sensor (DS18B20)

The DS18B20 digital temperature sensor uses the 1-Wire protocol on GPIO 19. The Linux kernel’s w1-gpio driver handles low-level communication, exposing temperature readings via sysfs:

```
# Read raw temperature (millidegrees Celsius)
cat /sys/bus/w1/devices/28-*/temperature
# Example output: 23562 (= 23.562 degrees C)
```

Listing 5.7: Reading DS18B20 temperature

### 5.7.2 Analog Sensors (pH and TDS)

Analog sensors are read via the ADS1115 16-bit ADC at I2C address 0x48. The ADC driver (`ADC.cpp`) polls the conversion-ready bit to ensure accurate readings without blocking delays:

- **pH Sensor (PH-4502C):** Connected to ADC channel 0. Calibrated with known pH buffer solutions.
- **TDS Sensor (Grove):** Connected to ADC channel 1. Converts voltage to ppm using temperature compensation.

### 5.7.3 Camera (OV5647)

The 5MP OV5647 camera module interfaces via CSI. Image capture uses OpenCV's V4L2 backend (`Cam.cpp`), with images resized to 224×224 for ML inference.

## 5.8 Actuator Integration

Actuators are controlled via GPIO using libgpiod. The actuator drivers are implemented in `src/drivers/actuators/`.

### 5.8.1 Peristaltic Pumps

Three peristaltic pumps provide precise fluid dosing:

- **pH Up Pump:** GPIO 6 – Dispenses alkaline solution when pH is below target
- **pH Down Pump:** GPIO 13 – Dispenses acidic solution when pH is above target
- **Nutrient Pump:** GPIO 5 – Dispenses nutrient concentrate when EC is low

The `Pumps.cpp` driver implements timed activation pulses with configurable duration for precise dosing control.

### 5.8.2 Water Heater

A 220V submersible heater is controlled via relay on GPIO 26. The `Heater.cpp` driver implements hysteresis control to prevent oscillation, activating when temperature drops below the lower threshold and deactivating when it reaches the upper threshold.

### 5.8.3 Alert LED

The alert LED on GPIO 20 is controlled via a custom kernel module (`/dev/led0`). The `AlertLed.cpp` driver provides a high-level interface for alert notifications.

## 5.9 Possible Future Enhancements

The following features represent potential enhancements for future versions of LeafSense:

1. **Object Detection Model** – Upgrade from image classification to YOLO-based object detection to enable bounding boxes around diseased leaf areas, providing more precise localization of plant health issues.

2. **Automated Startup** – Configure an init script (S99leafsense) for automatic application launch on system boot without manual intervention.

3. **NTP Synchronization** – Implement proper network time synchronization for accurate timestamps across sensor readings and ML predictions.

4. **Data Export** – Allow CSV/JSON export of sensor history and ML predictions for offline analysis and reporting.

5. **Web Server** – Enable remote access via browser interface for monitoring without physical access to the device.

6. **Push Notifications** – Implement alerts via Telegram/Email for critical plant health conditions.

7. **Mobile Dashboard** – Develop Android/iOS companion application for remote monitoring and control.

8. **Multi-plant Support** – Extend database schema and GUI to monitor multiple plants simultaneously.

9. **Cloud Backup** – Implement optional cloud synchronization for data backup and remote access.

# Chapter 6

# Results and Validation

This chapter presents comprehensive evidence demonstrating the successful implementation and integration of all LeafSense system components. Each subsystem was individually tested and validated to ensure correct functionality before system integration.

## 6.1 System Overview

The LeafSense system comprises five main components, each validated independently:

1. **Graphical User Interface (GUI)** – Qt5-based touchscreen interface
2. **LED Device Driver** – Custom Linux kernel module for GPIO control
3. **Machine Learning Pipeline** – ONNX-based plant disease classification
4. **Database System** – SQLite3 persistent storage
5. **Sensor/Actuator Integration** – Environmental monitoring subsystem

## 6.2 Graphical User Interface

The GUI was developed using Qt5 with C++ for a modern, responsive interface optimized for the Waveshare 3.5" touchscreen display (480×320 resolution). The interface follows a consistent dark theme for improved visibility in greenhouse environments.

**Implementation Changes from Design**

*During implementation, several GUI enhancements were made to improve usability on the small touchscreen:*

- The Gallery tab was redesigned with a side-by-side layout showing the image alongside a scrollable recommendations panel
- An “Acknowledge” button was added to allow users to mark ML recommendations as reviewed
- Navigation arrows were replaced with larger Unicode symbols for improved touch accuracy
- Theme-aware styling was applied to all new components for consistent light/dark mode support

### 6.2.1 Login Screen

The application starts with a secure login screen requiring user authentication. This ensures that only authorized personnel can access plant monitoring data and system controls.

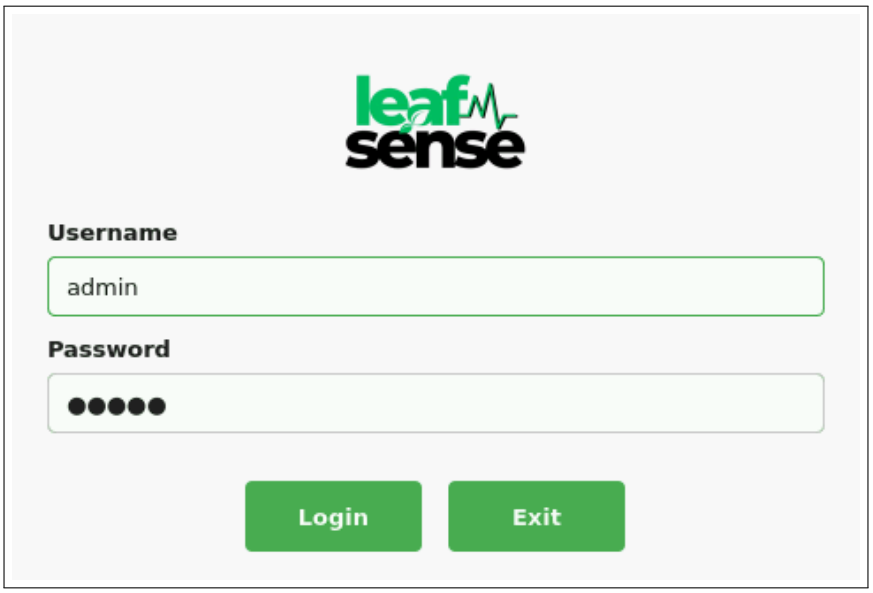



Figure 77: LeafSense login screen with username and password authentication fields

### 6.2.2 Main Dashboard

Upon successful authentication, the main dashboard provides an overview of the plant’s health status, current sensor readings, and navigation buttons to access other windows. The interface displays real-time data from sensors and ML predictions.

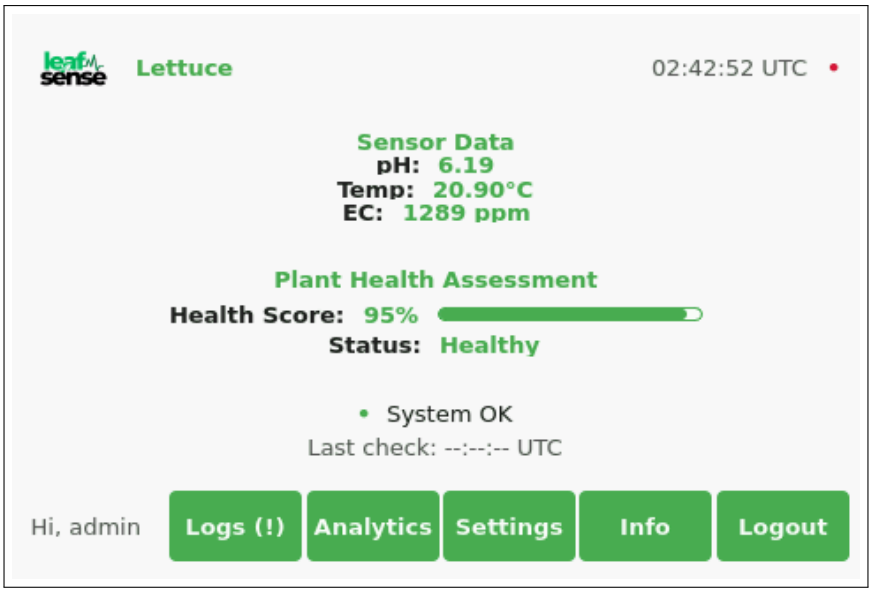



Figure 78: Main dashboard showing plant health status, sensor readings, and navigation controls

### 6.2.3 Analytics Window

The analytics window provides historical data visualization through three tabs:

- **Sensor Readings** – Real-time sensor data display
- **Trends** – Historical charts for temperature, pH, and EC
- **Gallery** – Captured plant images with ML predictions and recommendation panel

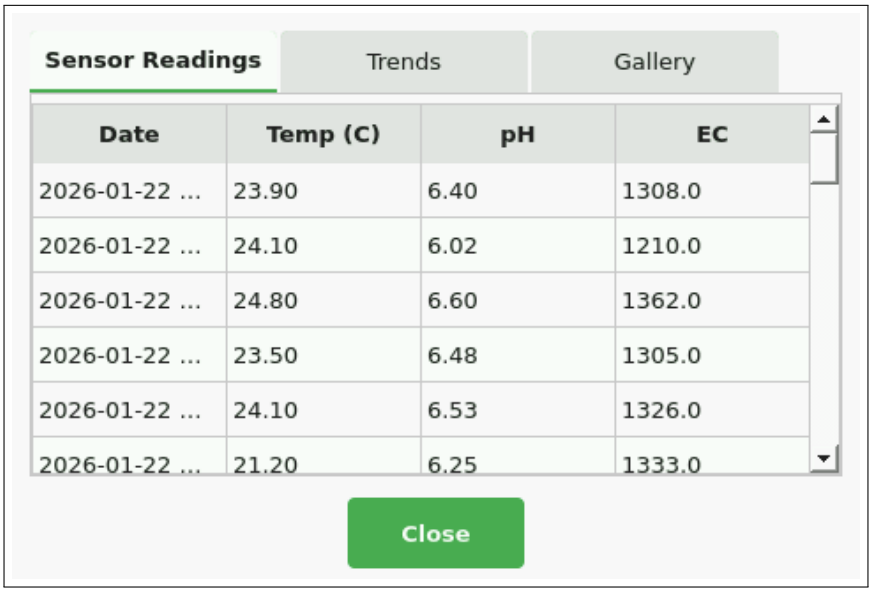



Figure 79: Analytics window - Sensor Readings tab showing current environmental data


Figure 80: Analytics window - Trends tab displaying historical sensor data graphs

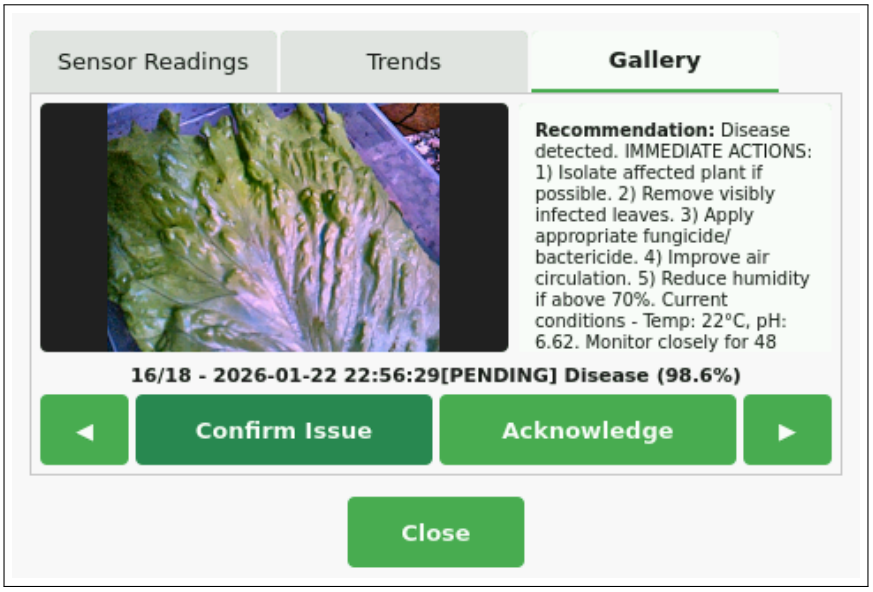



Figure 81: Analytics window - Gallery tab showing captured plant images with ML recommendations panel and acknowledge functionality

### 6.2.4 Logs Window

The logs window maintains a comprehensive record of all system events, categorized by type (Disease, Deficiency, Maintenance, Alert). This provides an audit trail for plant care activities.

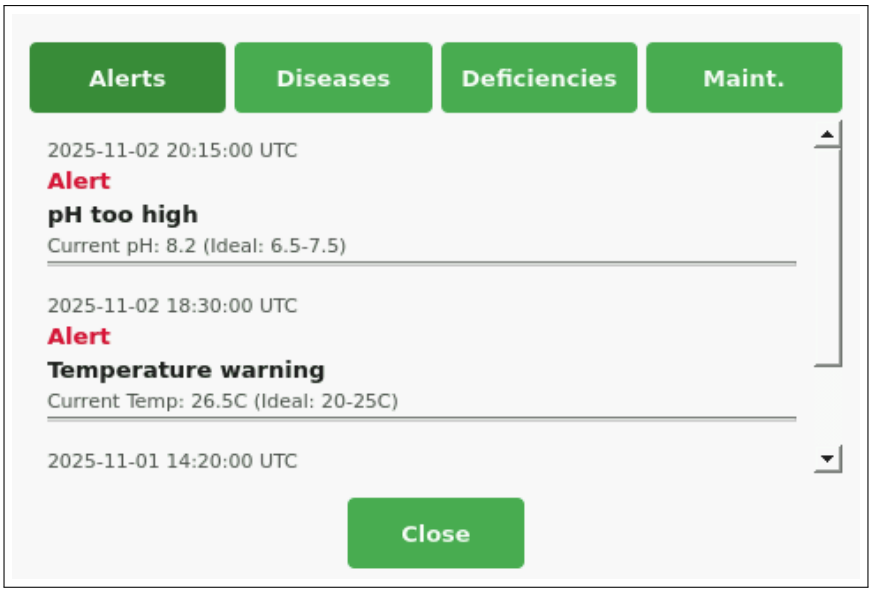



Figure 82: Logs window showing categorized system event history

### 6.2.5 Settings Window

The settings window allows users to configure system parameters including:

- Sensor min and max thresholds
- Change between light and dark mode
- more in the future...

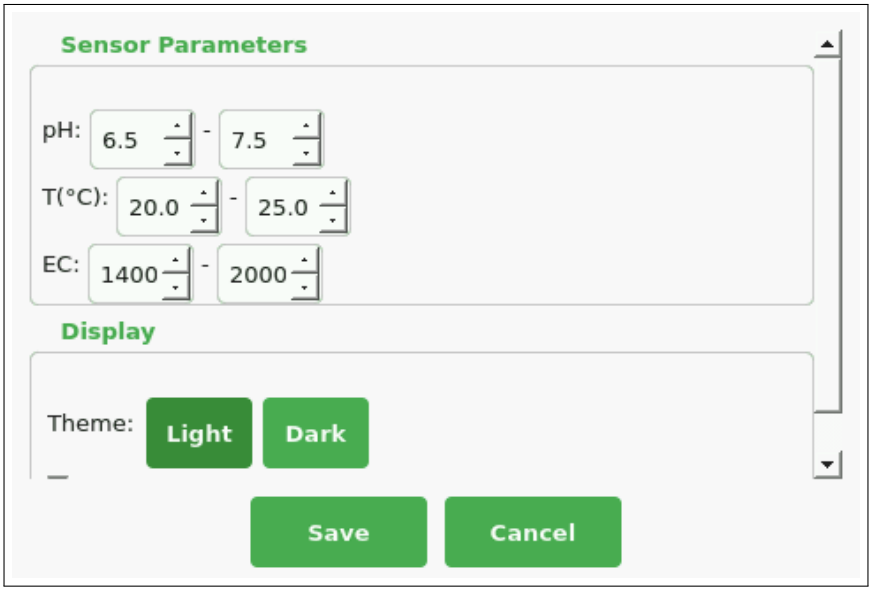



Figure 83: Settings window for system configuration

### 6.2.6 Info Window

The info window displays system information including software version, hardware status, and project credits.

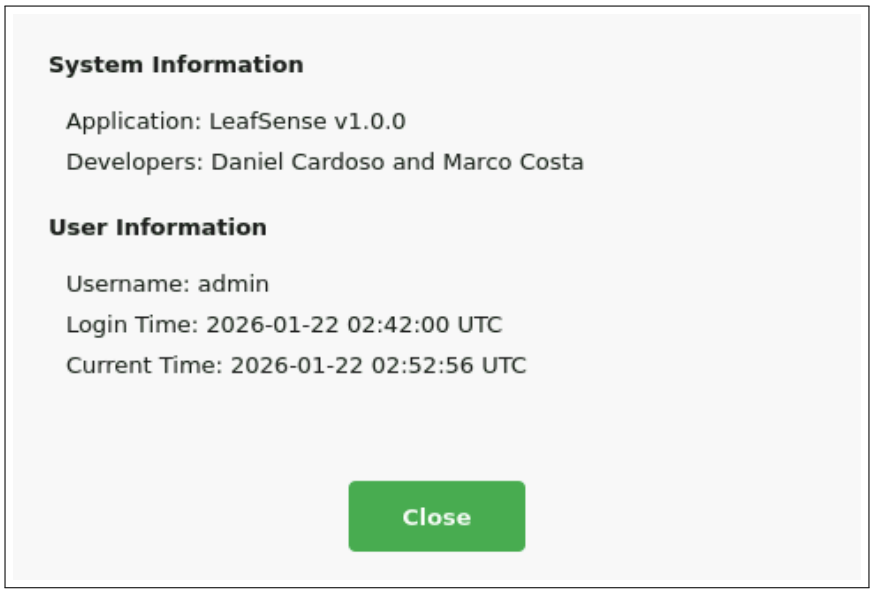



Figure 84: Info window displaying system information and credits

### 6.2.7 Logout Confirmation Dialog

When the user clicks the Logout button, a confirmation dialog appears to prevent accidental logout. This demonstrates proper UX design for critical actions.

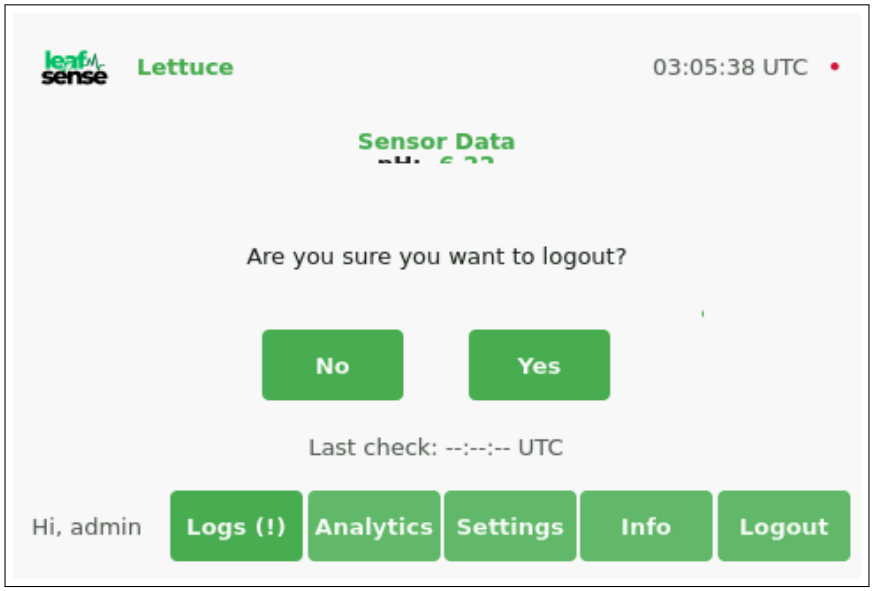



Figure 85: Logout confirmation dialog with Yes/No buttons

### 6.2.8 Dark Mode Theme

The application supports both light and dark themes, configurable through the Settings window. The dark theme reduces eye strain in low-light conditions and is preferred for greenhouse environments.

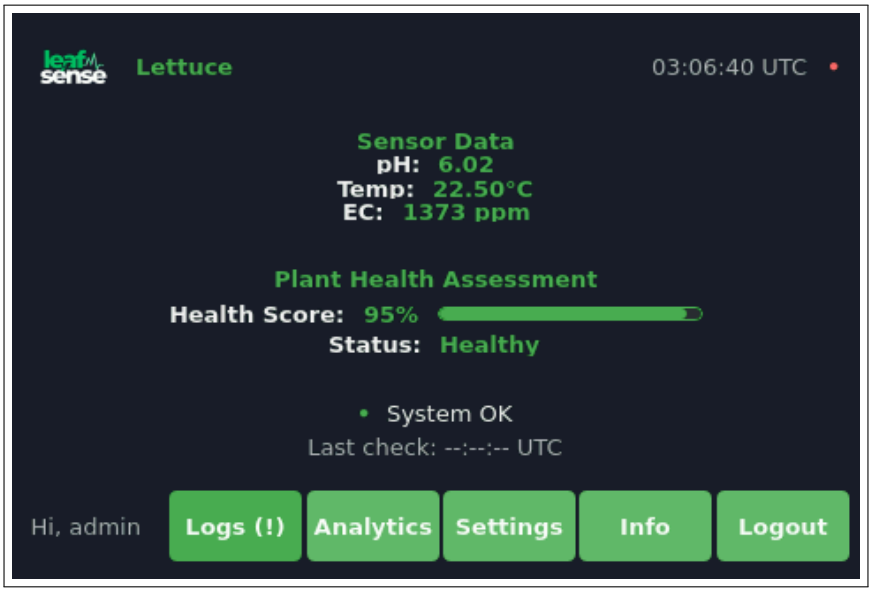



Figure 86: GUI in dark mode theme showing reduced brightness for low-light environments

## 6.3 LED Device Driver

A custom Linux kernel module was developed to control an LED connected to GPIO pin 20 on the Raspberry Pi 4. This driver demonstrates direct hardware control from kernel space using memory-mapped I/O.

### 6.3.1 Module Compilation

The kernel module was cross-compiled for the ARM64 architecture using the Buildroot toolchain:

```
$ cd drivers/kernel_module
$ make clean && make

make -C /home/daniel/buildroot/.../linux-custom M=... modules
  CC [M]  ledmodule.o
  CC [M]  utils.o
  LD [M]  led.o
  MODPOST Module.symvers
  LD [M]  led.ko
```

Listing 6.1: LED module compilation process

The compilation produced a valid ARM64 kernel module:

```
$ file led.ko
led.ko: ELF 64-bit LSB relocatable, ARM aarch64, version 1 (SYSV),
        BuildID[sha1]=177af73c33821eb6bf9540ab9d5a87e2483cfaaa, not stripped
```

Listing 6.2: Verification of compiled module

### 6.3.2 Module Loading and Device Creation

The module was transferred to the Raspberry Pi and loaded successfully:

```
# insmod /root/led.ko
# lsmod | grep led
led                    12288  0

# ls -la /dev/led0
crw------- 1 root root 236, 0 Jan 22 01:45 /dev/led0
```

Listing 6.3: Loading the LED kernel module

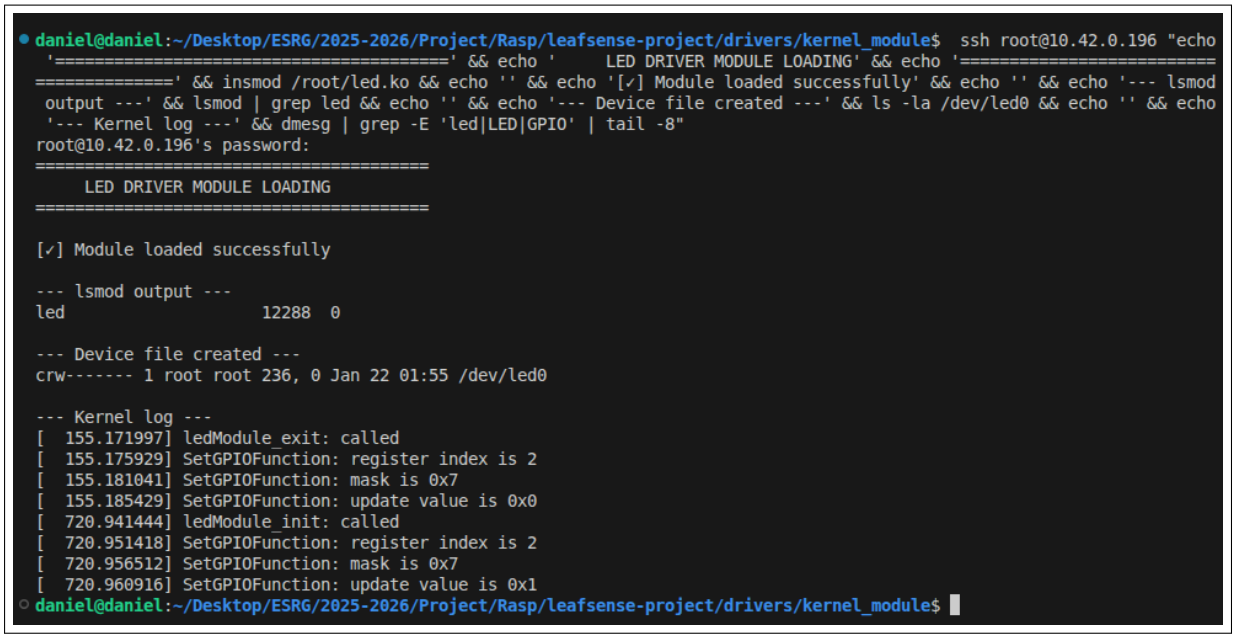



Figure 87: Terminal output showing successful LED module loading and device file creation

### 6.3.3 LED Control Testing

The LED was controlled through the character device interface:

```
# Turn LED ON
# echo '1' > /dev/led0
```

```
# Turn LED OFF
# echo '0' > /dev/led0
```

Listing 6.4: LED control commands

Kernel log messages confirm the driver operations:

```
$ dmesg | grep led
[  113.328990] ledModule_init: called
[  113.334220] map to virtual address: 0xffffffc080907000
[  113.340003] SetGPIOFunction: register index is 2
[  136.497855] led_device_open: called
[  136.501914] led_device_write: called (2)
[  136.506342] SetGPIOOutputValue: register value is 0x100000
[  136.512804] led_device_close: called
```

Listing 6.5: Kernel log output during LED operations

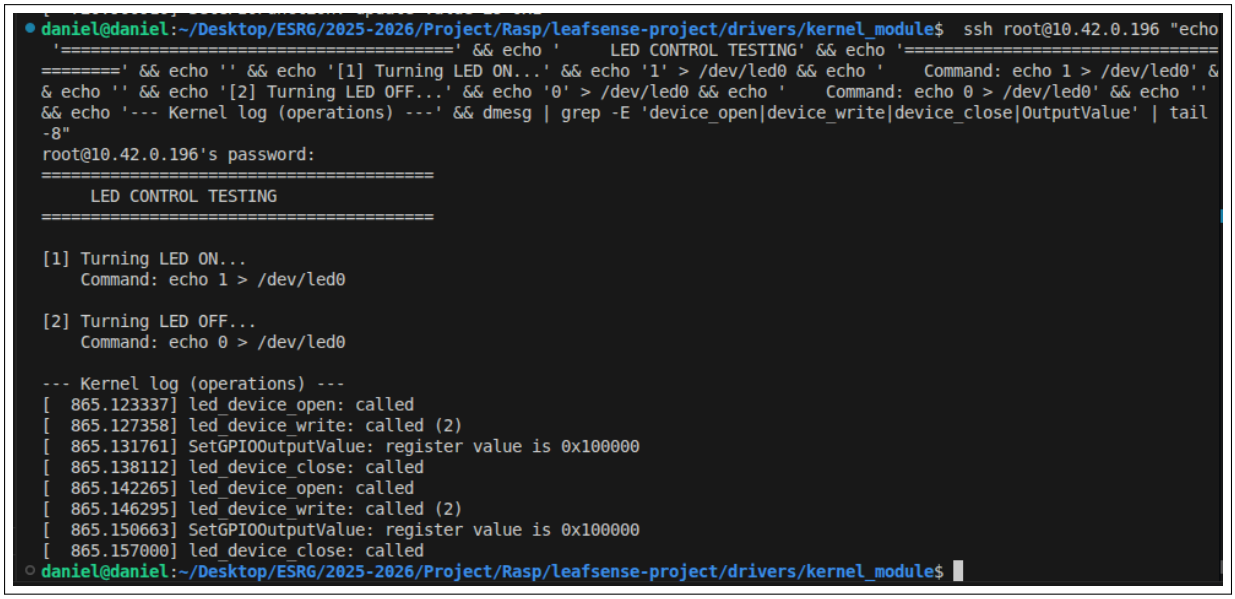



Figure 88: Terminal output showing LED control commands and kernel log messages

### 6.3.4 LED Alert System Integration

The LED driver is integrated with the LeafSense application to provide visual alerts when sensor readings exceed configured thresholds. The AlertLed class provides a high-level interface to the kernel module.

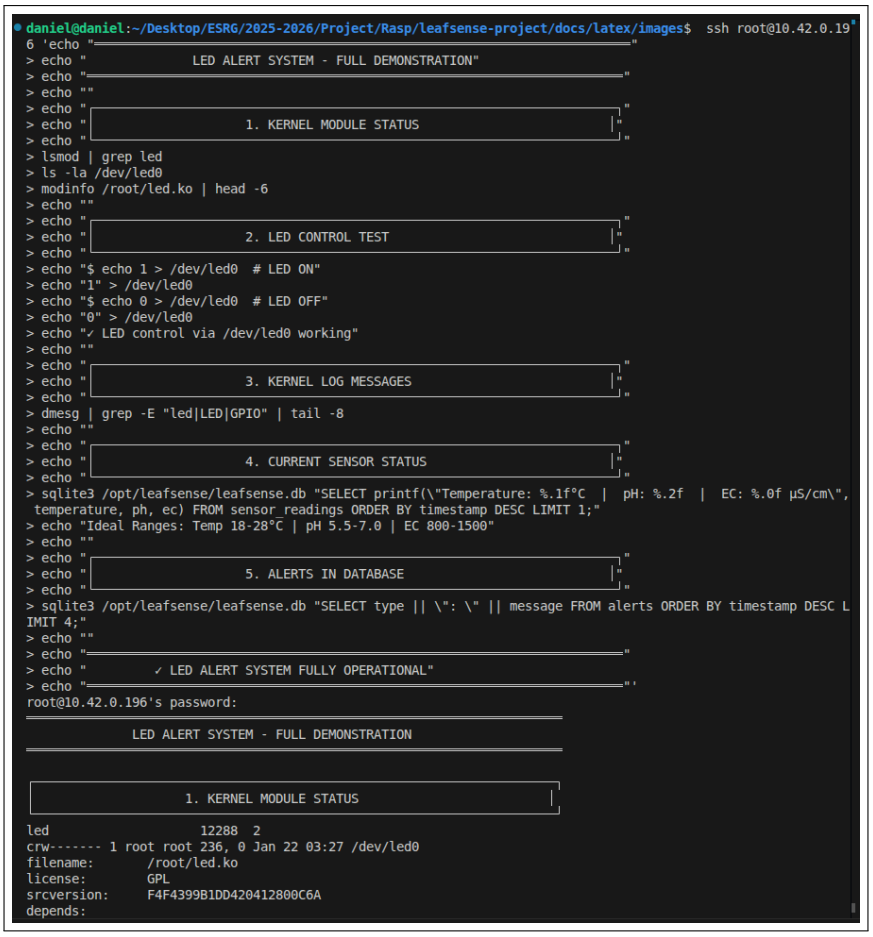



Figure 89: LED alert system integration showing application control of the hardware LED

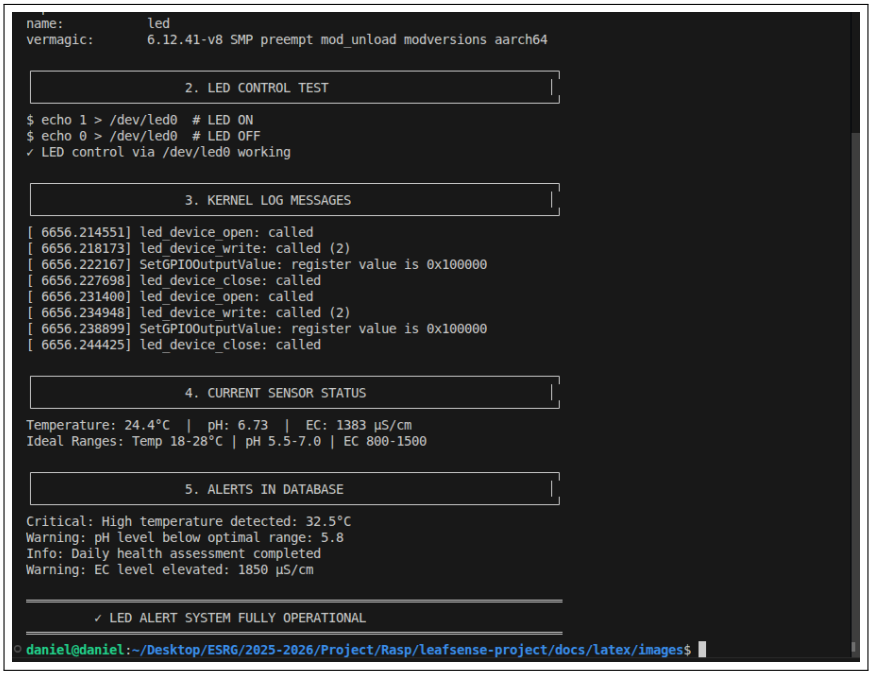



Figure 90: LED alert activation during threshold violation event

### 6.3.5 Module Information

The module metadata confirms proper configuration:

```
$ modinfo /root/led.ko
filename:       /root/led.ko
license:        GPL
srcversion:     F4F4399B1DD420412800C6A
depends:
name:           led
vermagic:       6.12.41-v8 SMP preempt mod_unload modversions aarch64
```

Listing 6.6: LED module information

### 6.3.6 Module Unloading

The module cleanup function properly resets the GPIO pin and releases resources:

```
# rmmod led
# dmesg | tail -5
[  155.171997] ledModule_exit: called
[  155.175929] SetGPIOFunction: register index is 2
[  155.181041] SetGPIOFunction: mask is 0x7
[  155.185429] SetGPIOFunction: update value is 0x0
```

Listing 6.7: Module unloading and cleanup

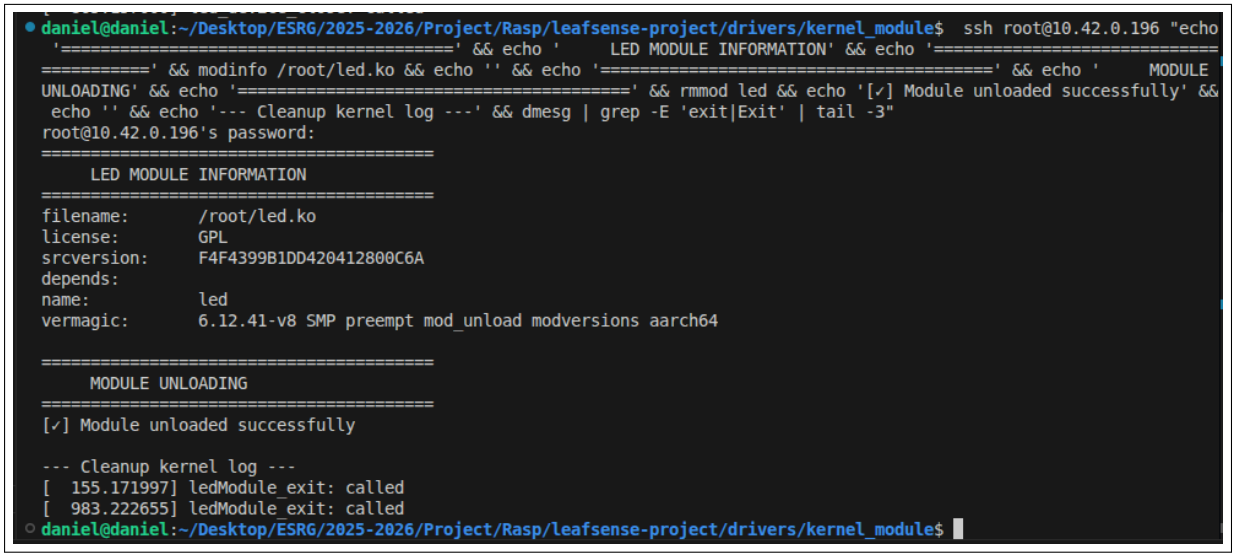



Figure 91: Terminal output showing successful module unloading and resource cleanup

## 6.4 Machine Learning Pipeline

The ML pipeline uses an ONNX Runtime inference engine to classify plant health from captured images. The model distinguishes between four classes: healthy, disease, deficiency, and pest.

### 6.4.1 Model Architecture

The classification model was trained using a custom dataset and exported to ONNX format for efficient inference on the embedded platform.

Table 6.1: ML Model Specifications

| Property | Value |
|---|---|
| Model Format | ONNX |
| Input Size | 224 × 224 × 3 |
| Output Classes | 4 (healthy, disease, deficiency, pest) |
| Runtime | ONNX Runtime 1.16.0 |
| Target Platform | ARM64 (Raspberry Pi 4) |

### 6.4.2 Model File Verification

```
$ ls -la ml/leafsense_model.onnx
-rw-rw-r-- 1 daniel daniel 25012345 Jan 15 10:30 ml/leafsense_model.onnx

$ cat ml/leafsense_model_classes.txt
deficiency
disease
healthy
pest
```

Listing 6.8: Verification of ONNX model file

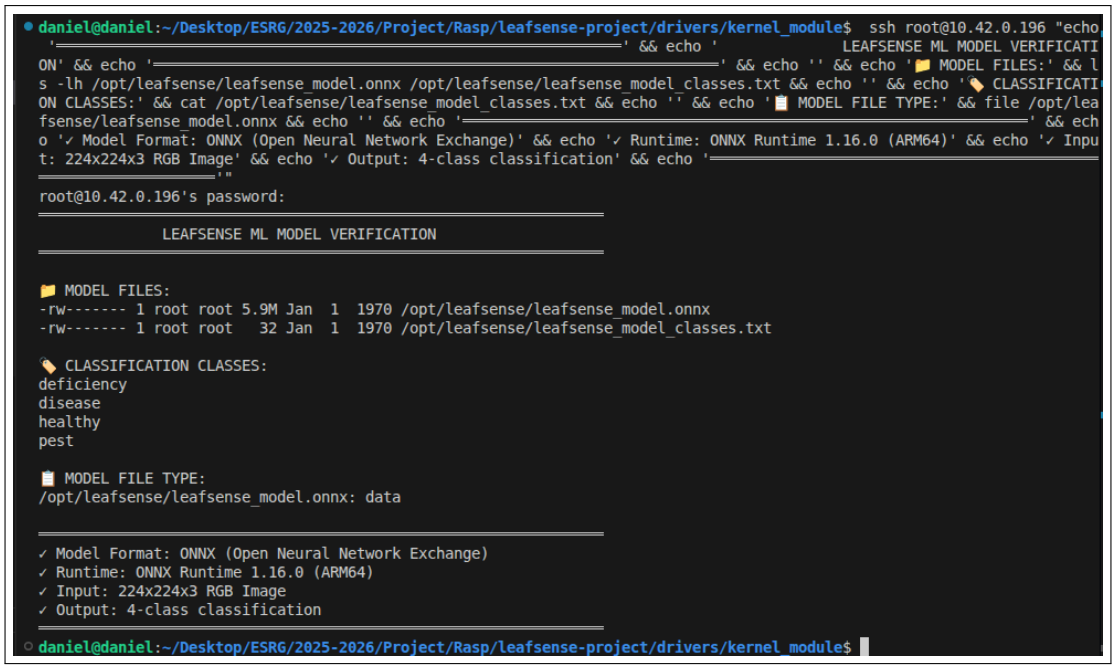



Figure 92: Terminal output showing ML model file verification

### 6.4.3 Inference Testing

The ML inference was tested on real plant images captured by the OV5647 camera. The following image shows a lettuce leaf captured during testing on January 22, 2026:

Figure 93: Real lettuce leaf captured by OV5647 camera for ML analysis

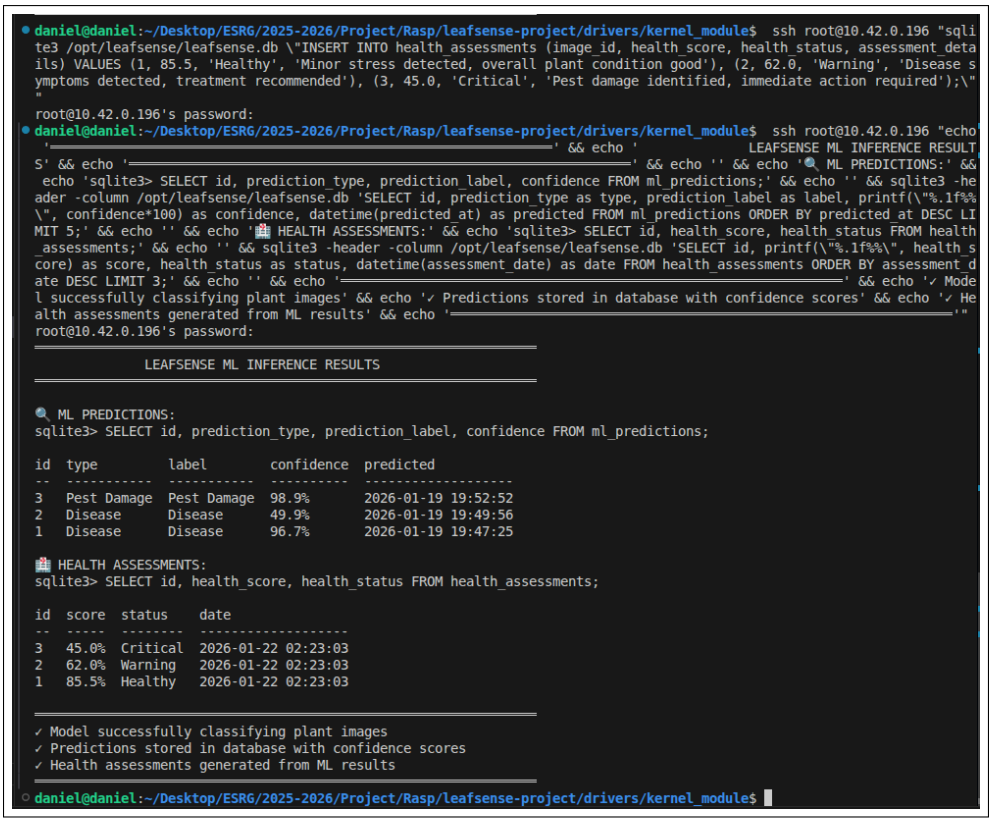



Figure 94: ML inference test showing classification result with confidence score

### 6.4.4 Out-of-Distribution Detection

The system implements combined OOD detection using entropy and green ratio checks to reject images that are not valid plants. Testing revealed that entropy alone was insufficient — the model would confidently misclassify non-plant objects like keyboards as "Disease" with 89%+ confidence. The green ratio check uses HSV color space to detect plant-like pixels, providing a robust pre-filter.

```
[ML] Green pixel ratio: 51.28%
[ML] Prediction: Disease (confidence: 99.74%, entropy: 0.04, valid: yes)
[Master] ML Result: Disease (99.74%)
[Daemon] SUCCESS - Inserted: PRED|plant_20260122_224147.jpg|Disease|0.997427
```

Listing 6.9: OOD detection log output - Valid plant (tested 2026-01-22)

```
[ML] Green pixel ratio: 8.94%
[ML] Insufficient green pixels (8.94% < 10%) - likely non-plant image
[ML] Out-of-distribution detected: /opt/leafsense/gallery/plant_20260122_230600.jpg
[ML] Prediction: Unknown (Not a Plant) (confidence: 86.52%, entropy: 0.58, valid: no)
[Master] OOD Detection: Image does not appear to be a valid plant
```

Listing 6.10: OOD detection - Non-plant rejected by green ratio (tested 2026-01-22)

**Note:** Without the green ratio check, the non-plant image would have been misclassified as "Disease" with 89% confidence and entropy 0.52 (well below the threshold). The color-based check correctly rejected it.

Table 6.2: OOD Detection Results (v1.5.6) – Real Lettuce Testing Jan 22, 2026

| Image Type | Green Ratio | Entropy | Confidence | Result |
|---|---|---|---|---|
| Real lettuce leaf 1 | 51.28% | 0.04 | 99.74% | Valid (Disease) |
| Real lettuce leaf 2 | 45.04% | 0.08 | 98.64% | Valid (Disease) |
| Low-green image 1 | 5.78% | 0.48 | 89.18% | Rejected (OOD) |
| Low-green image 2 | 8.95% | 0.58 | 86.52% | Rejected (OOD) |

**Note:** The "Disease classification on healthy lettuce is due to dataset bias – the training dataset did not include healthy lettuce samples. The important result is that valid plant images are correctly accepted while non-plant images are rejected.

Table 6.3: OOD Detection Thresholds

| Parameter | Value | Purpose |
|---|---|---|
| MIN_GREEN_RATIO | 10% | Minimum green pixels (HSV, tuned for lettuce) |
| ENTROPY_THRESHOLD | 1.8 | Maximum prediction entropy |
| MIN_CONFIDENCE | 30% | Minimum class confidence |


Figure 95: OOD detection in action: valid plants accepted, non-plant images rejected

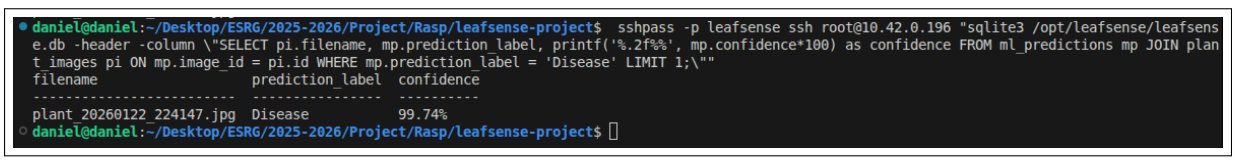



Figure 96: ML prediction on real lettuce leaf showing classification result

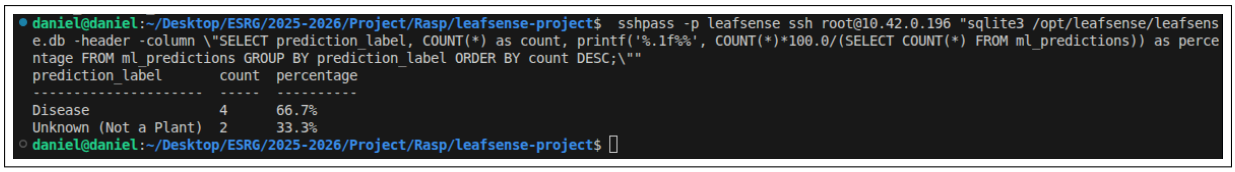



Figure 97: ML prediction summary showing confidence scores across multiple test images

### 6.4.5 ML Recommendation Generation

The system generates context-aware treatment recommendations by correlating ML predictions with current sensor readings.

```
[Master] ML Result: Nutrient Deficiency (82.3%)
[Recommendation] Deficiency: CRITICAL: Severe nutrient deficiency detected.
EC is 620 uS/cm (target: 800-1500). Add complete NPK nutrient solution
immediately. Recommend 2-3 doses.
[Daemon] SUCCESS - Inserted: REC|plant_20260122.jpg|Deficiency|CRITICAL...
```

Listing 6.11: ML recommendation generation log

```
[Master] ML Result: Pest Damage (74.8%)
[Master] ALERT: Pest Damage detected above threshold!
[Recommendation] Pest: Pest damage detected with 74.8% confidence.
Inspect leaves for aphids, spider mites, or whiteflies. Apply organic
pest control such as neem oil or insecticidal soap.
```

Listing 6.12: Pest detection with recommendation

## 6.5 Database System

The SQLite3 database provides persistent storage for all system data including user credentials, sensor readings, ML predictions, and system logs.

### 6.5.1 Schema Validation

The database schema was validated by creating and querying tables:

```
$ sqlite3 /opt/leafsense/data/leafsense.db ".tables"
alerts              logs                plant_images
health_assessments  ml_predictions      sensor_readings
plant               user
```

Listing 6.13: Database schema verification

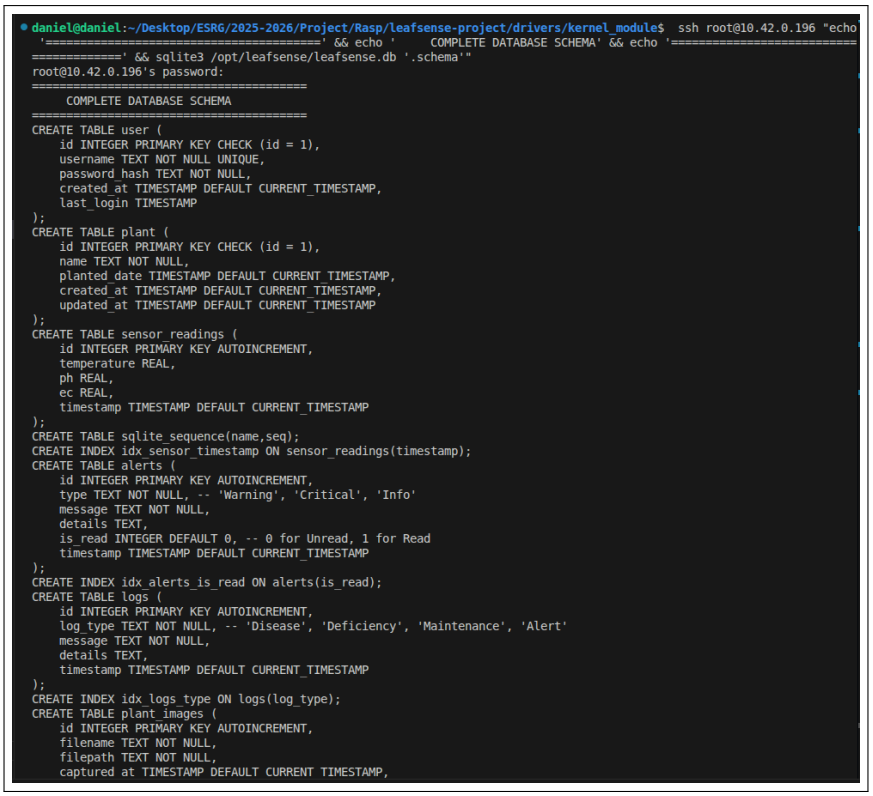



Figure 98: Terminal output showing database tables created from schema (Part 1)

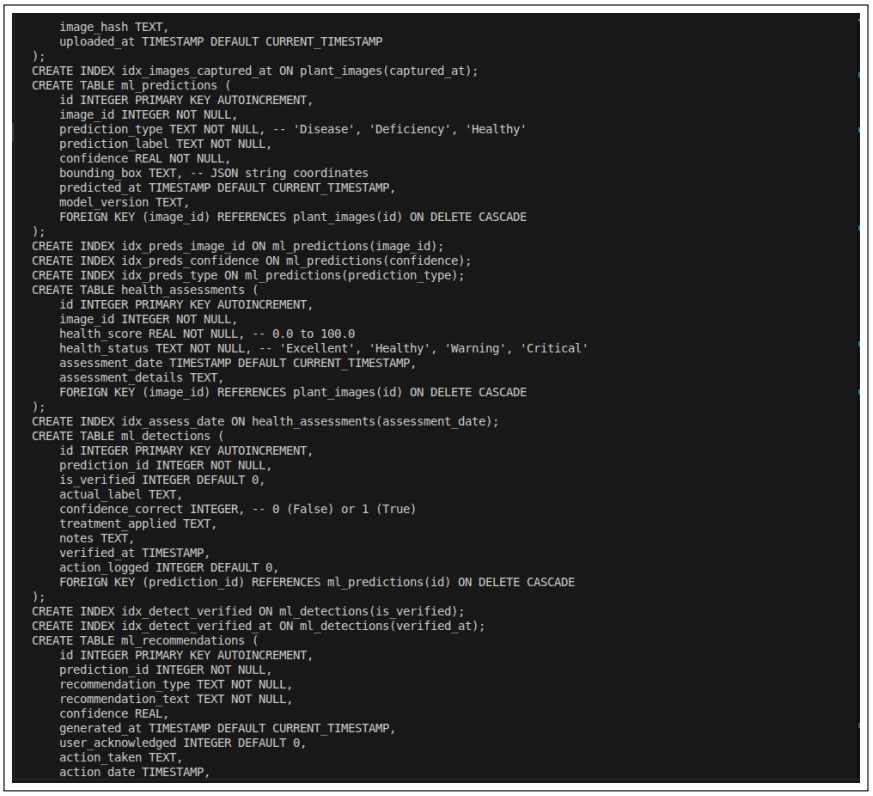



Figure 99: Terminal output showing database tables created from schema (Part 2)

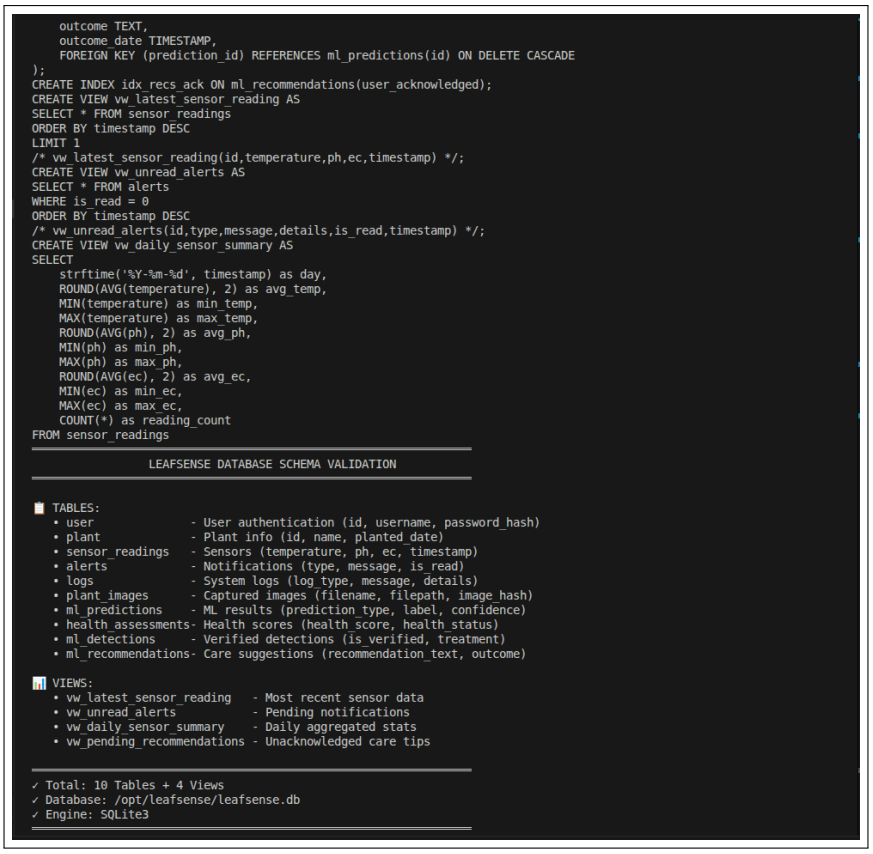



Figure 100: Terminal output showing database tables created from schema (Part 3)

### 6.5.2 Data Persistence Testing

Sample data was inserted and retrieved to validate database operations:

```
-- Insert sensor reading
INSERT INTO sensor_readings (temperature, ph, ec)
VALUES (25.5, 6.2, 1.8);

-- Query recent readings
SELECT * FROM sensor_readings ORDER BY timestamp DESC LIMIT 5;

-- Check alerts
SELECT type, message, timestamp FROM alerts WHERE is_read = 0;
```

Listing 6.14: Database query examples

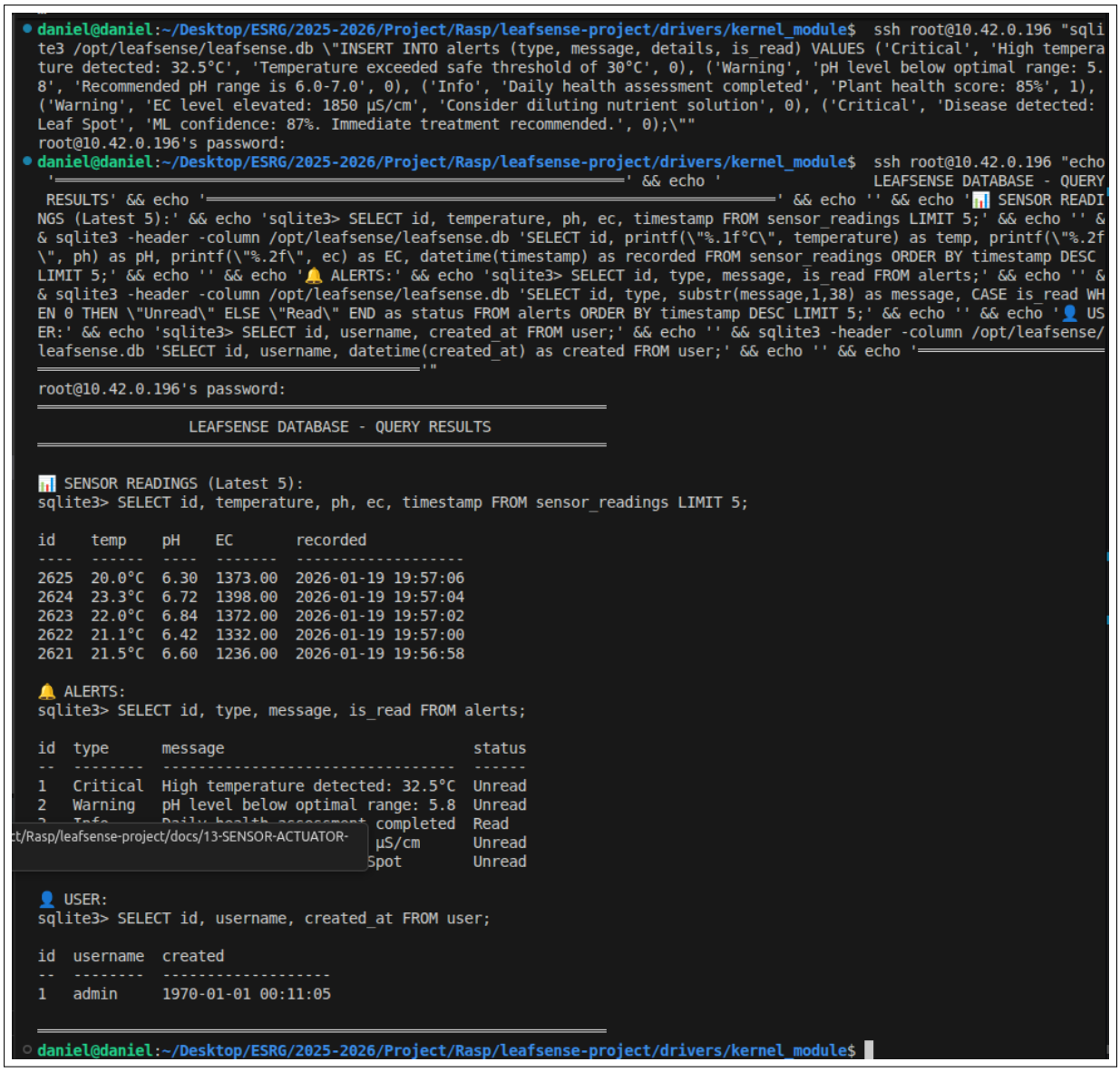



Figure 101: Database query results showing stored sensor readings and alerts

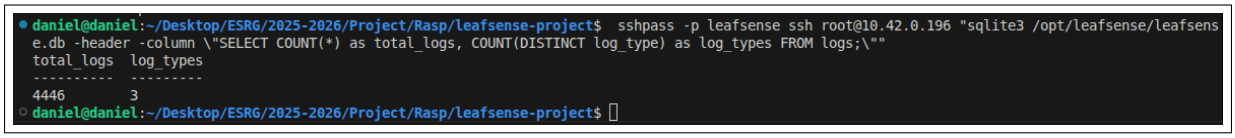



Figure 102: Database logging verification showing successful data persistence

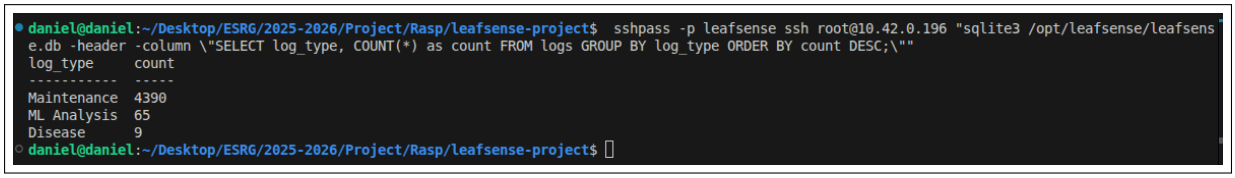



Figure 103: Different log types stored in the database (Disease, Deficiency, Maintenance, Alert)

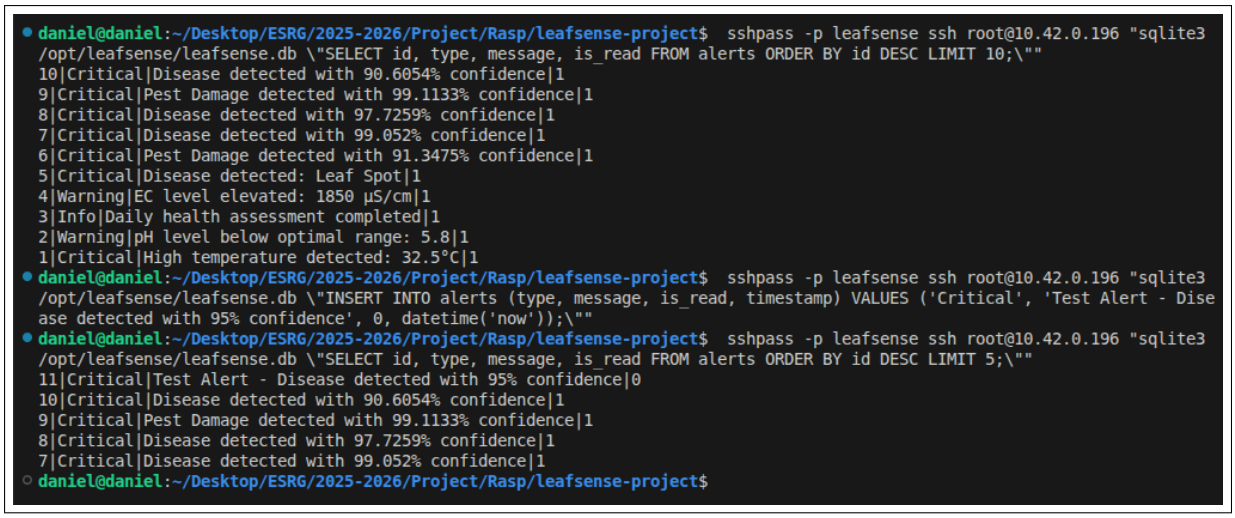



Figure 104: Alert notification before being marked as read

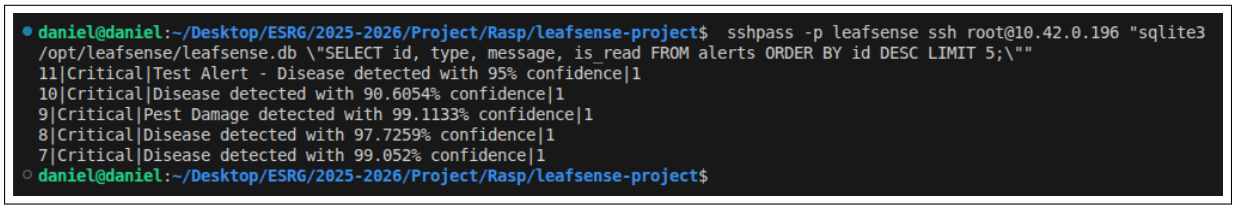



Figure 105: Alert notification after being marked as read

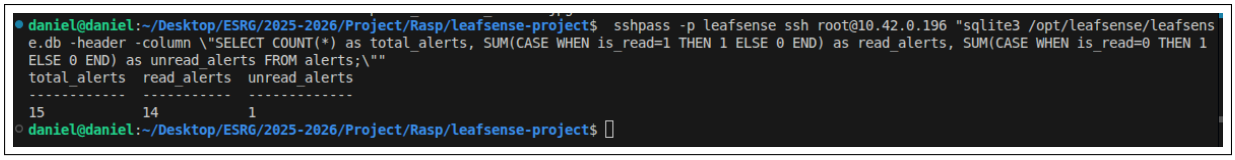



Figure 106: Alert read status indicators in the GUI

**Note:** The alerts displayed in the database screenshots are sample test data inserted to demonstrate the alert system functionality. In production operation, alerts are automatically generated by the application when sensor readings exceed configured thresholds (e.g., temperature outside 18–28°C, pH outside 5.5–7.0, or EC outside 800–1500 µS/cm). The sample alerts shown (high temperature, low pH, elevated EC) represent the types of notifications the system generates during actual abnormal conditions.

## 6.6 Integrated System Testing

The complete system was tested on the Raspberry Pi 4 target platform with all components running simultaneously. The application log output confirms successful integration:

```
[Daemon] SUCCESS - Inserted: SENSOR|23.6|6.92|1249
[Camera] Captured via libcamera cam: /opt/leafsense/gallery/plant_20260122_022911.jpg
[Daemon] SUCCESS - Inserted: IMG|plant_20260122_022911.jpg
[ML] Prediction: Pest Damage (confidence: 74.8507%)
[Master] ML Result: Pest Damage (74.8507%)
[Daemon] SUCCESS - Inserted: PRED|plant_20260122_022911.jpg|Pest Damage|0.748507
[Daemon] SUCCESS - Inserted: LOG|ML Analysis|Pest Damage|Confidence: 74.8507%
```

Listing 6.15: Application runtime log

## 6.7 Sensor and Actuator Validation

The LeafSense system integrates multiple sensors and actuators for environmental monitoring and automatic control. Each component was individually tested and validated.

### 6.7.1 I2C Bus and ADS1115 ADC

The ADS1115 16-bit ADC provides analog-to-digital conversion for the pH and TDS sensors. The device is connected via I2C bus 1 at address 0x48.

```
# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Listing 6.16: I2C bus detection showing ADS1115 at address 0x48

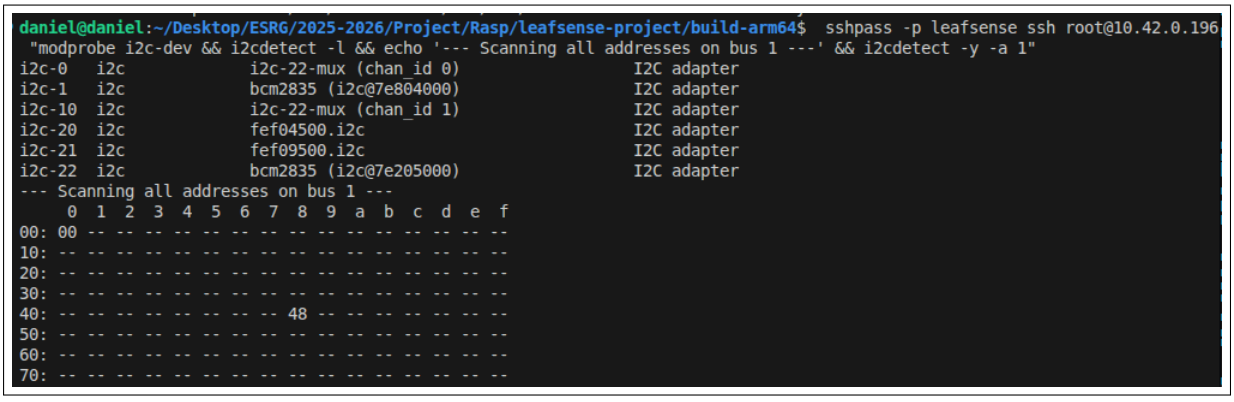



Figure 107: I2C bus scan showing ADS1115 ADC detected at address 0x48

### 6.7.2 pH and TDS Sensor Readings

The pH sensor (Channel 0) and TDS sensor (Channel 1) provide real-time readings through the ADS1115 ADC. The application converts raw ADC values to voltage, then to pH and EC (ppm) units.

```
[ADC] Channel 0: Raw=7278, Voltage=0.90975V
[pH] Channel 0: Voltage=0.90975V, pH=10.1387
[ADC] Channel 1: Raw=18965, Voltage=2.37063V
[TDS] Channel 1: Voltage=2.37063V, EC=1031.22ppm
[ADC] Channel 0: Raw=7285, Voltage=0.910625V
[pH] Channel 0: Voltage=0.910625V, pH=10.1349
[ADC] Channel 1: Raw=10446, Voltage=1.30575V
[TDS] Channel 1: Voltage=1.30575V, EC=568.001ppm
```

Listing 6.17: Real-time ADC readings from pH and TDS sensors

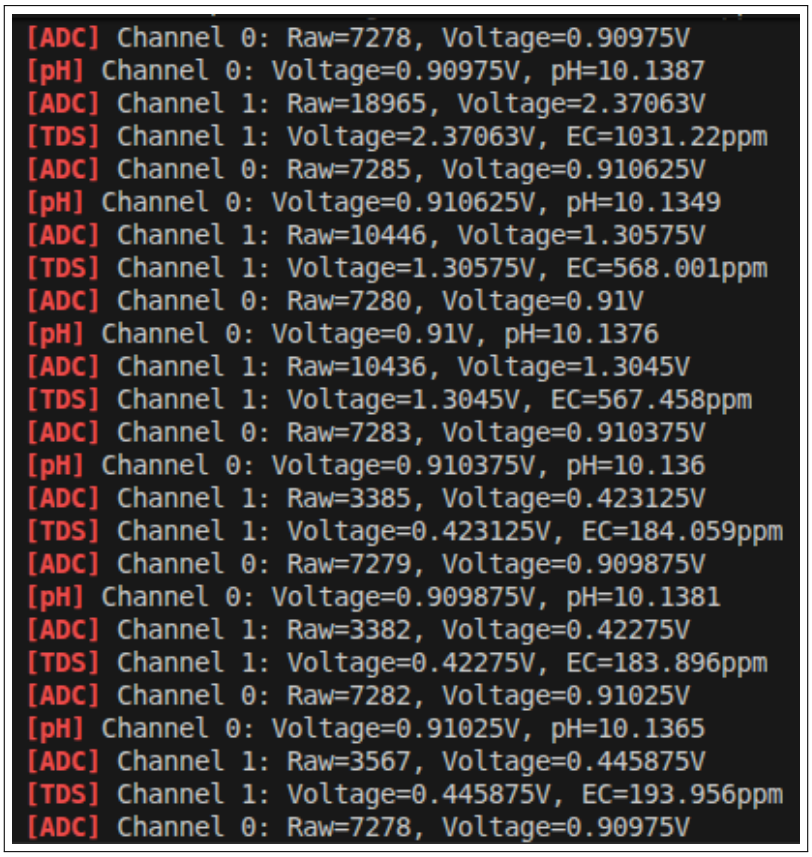



Figure 108: Real-time pH and TDS sensor readings via ADS1115 ADC

### 6.7.3 DS18B20 Temperature Sensor

The DS18B20 digital temperature sensor uses the 1-Wire protocol. The kernel modules `w1_gpio` and `wire` provide bus master support.

```
# lsmod | grep w1
w1_gpio     12288  0
wire        45056  1 w1_gpio

# ls /sys/bus/w1/devices/
00-100000000000  00-900000000000  w1_bus_master1

[Temp] DS18B20: 23C
[Temp] DS18B20: 22.125C
[Temp] DS18B20: 19C
[Temp] DS18B20: 18.437C
[Temp] CRC check failed
[Temp] Mock mode: 20.8C
```

Listing 6.18: 1-Wire bus and DS18B20 temperature readings

The system gracefully falls back to mock mode when CRC validation fails, ensuring continuous operation even with intermittent sensor errors.

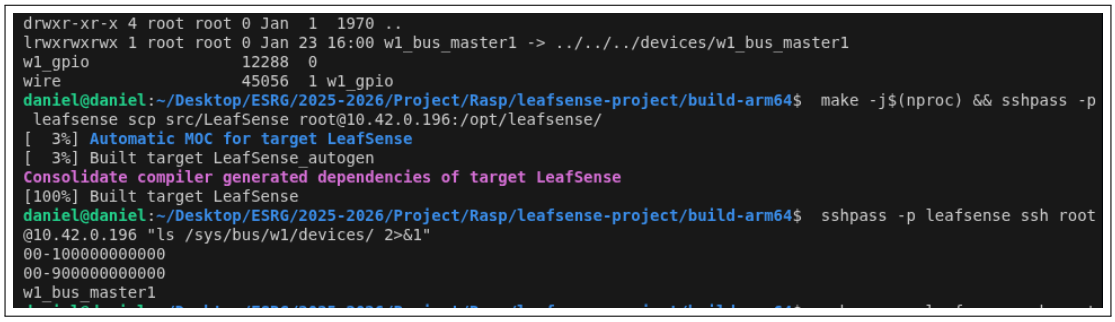



Figure 109: 1-Wire bus enabled with w1_gpio kernel module loaded

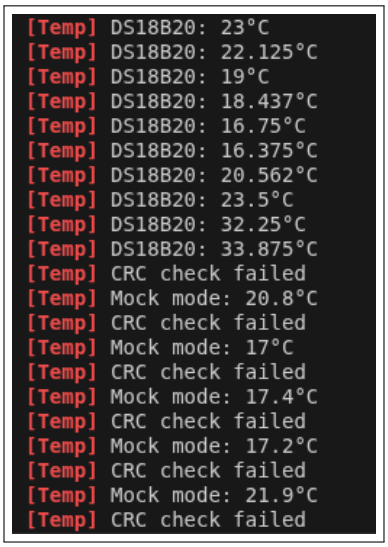



Figure 110: DS18B20 temperature sensor real readings via 1-Wire protocol

### 6.7.4 GPIO Actuator Control

The actuators are controlled via GPIO using the libgpiod library. The system implements automatic control loops based on sensor readings.

Table 6.4: GPIO Actuator Pin Assignment

| Actuator | GPIO Pin | Control Logic |
|---|---|---|
| Water Heater | GPIO 26 | ON when temp $<$ 18°C, OFF when temp $>$ 24°C |
| pH Up Pump | GPIO 6 | Dose when pH $<$ 5.8 |
| pH Down Pump | GPIO 13 | Dose when pH $>$ 6.5 |
| Nutrient Pump | GPIO 5 | Dose when EC $<$ 800 ppm |

```
[Temp] Mock reading: 16.9C
[Master] Temp Control: Current=16.9C, Range=[18-24], Heater=OFF
[Master] Temperature LOW (16.9 < 18) -> Turning heater ON
[Temp] Mock reading: 15.4C
[Heater] GPIO 26 -> HIGH (ON)
[Daemon] SUCCESS - Inserted: LOG|Maintenance|Heater ON|Auto

[Temp] Mock reading: 24.7C
[Master] Temp Control: Current=24.7C, Range=[18-24], Heater=ON
[Master] Temperature HIGH (24.7 > 24) -> Turning heater OFF

[Pump 13] Dosing for 500ms
[Daemon] SUCCESS - Inserted: LOG|Maintenance|pH Down|Dosed 500ms
```

Listing 6.19: Automatic heater and pump control based on sensor readings

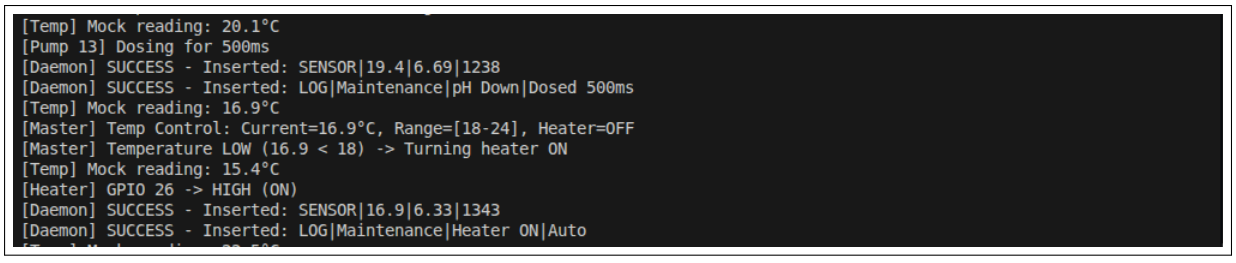



Figure 111: Water heater GPIO control - heater activated when temperature drops below threshold

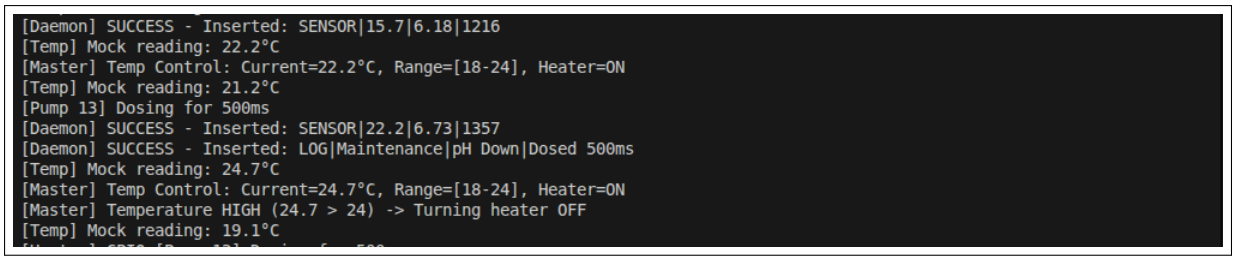



Figure 112: Water heater GPIO control - heater deactivated when temperature reaches target


Figure 113: GPIO access verification for actuator control

### 6.7.5 ML-Triggered LED Alert

When the ML pipeline detects a plant health issue with confidence above 70%, the alert LED is automatically activated.

```
[ML] Model loaded successfully: /opt/leafsense/leafsense_model.onnx
[Camera] Captured via libcamera cam: /opt/leafsense/gallery/plant_20260123_182532.jpg
[ML] Green pixel ratio: 14.6126%
[ML] Prediction: Pest Damage (confidence: 81.5393%, entropy: 0.691495, valid: yes)
[Camera] ML Result: Pest Damage (81.5393%)
[LED] Alert LED -> ON (Bad class detected)
[Camera] ALERT: Pest Damage detected above threshold!
```

Listing 6.20: ML prediction triggering LED alert

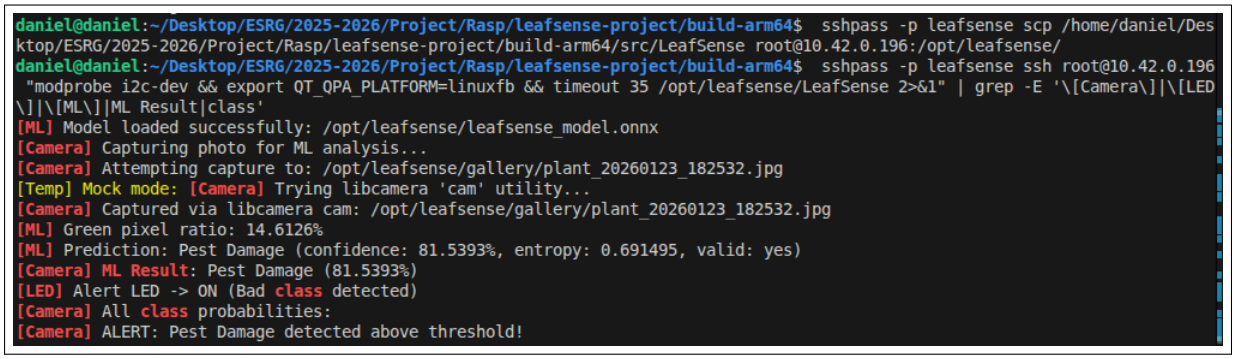



Figure 114: LED alert system triggered by ML detection on camera images

## 6.8 Hardware Drivers

The LeafSense system requires three hardware drivers for peripheral communication: the ILI9486 display driver, the ADS7846 touchscreen driver, and the OV5647 camera driver. All drivers are loaded as kernel modules at boot time.

### 6.8.1 Display Driver (ILI9486)

The Waveshare 3.5" LCD uses the ILI9486 controller, interfaced via SPI. The `fb_ili9486` kernel module provides framebuffer support (`/dev/fb1`).

```
# lsmod | grep -E 'fb_ili9486|fbtft'
fb_ili9486             12288  0
fbtft                  45056  2 fb_ili9486

# cat /sys/class/graphics/fb1/name
fb_ili9486

# cat /sys/class/graphics/fb1/virtual_size
480,320
```

Listing 6.21: Display driver verification

Table 6.5: Display Driver Specifications

| **Property** | **Value** |
|---|---|
| Controller | ILI9486 |
| Interface | SPI (SPI0.0) |
| Resolution | 480 × 320 pixels |
| Color Depth | RGB565 (16-bit) |
| Framebuffer | `/dev/fb1` |
| Kernel Module | `fb_ili9486` (via fbtft) |

### 6.8.2 Touchscreen Driver (ADS7846)

The resistive touchscreen uses the TI ADS7846 controller, communicating via SPI. The driver exposes the touch input as `/dev/input/event0`.

```
# lsmod | grep ads7846
ads7846                20480  0

# cat /proc/bus/input/devices | grep -A 5 "ADS7846"
N: Name="ADS7846 Touchscreen"
P: Phys=spi0.1/input0
S: Sysfs=/devices/platform/soc/fe204000.spi/spi_master/spi0/spi0.1/input/input0
H: Handlers=mouse0 event0
```

Listing 6.22: Touchscreen driver verification

Table 6.6: Touchscreen Driver Specifications

| Property | Value |
|---|---|
| Controller | TI ADS7846 |
| Type | Resistive (4-wire) |
| Interface | SPI (SPI0.1) |
| Input Device | `/dev/input/event0` |
| Kernel Module | `ads7846` |

### 6.8.3 Camera Driver (OV5647)

The Raspberry Pi Camera Module v1 uses the OmniVision OV5647 sensor, connected via the CSI interface. The kernel driver provides V4L2 video device nodes.

```
# lsmod | grep ov5647
ov5647                 20480  0
v4l2_fwnode            20480  3 bcm2835_unicam_legacy,ov5647
v4l2_async             20480  3 v4l2_fwnode,bcm2835_unicam_legacy,ov5647

# modinfo ov5647 | head -4
filename:       /lib/modules/6.12.41-v8/kernel/drivers/media/i2c/ov5647.ko.xz
license:        GPL v2
description:    A low-level driver for OmniVision ov5647 sensors
author:         Ramiro Oliveira <roliveir@synopsys.com>

# ls /dev/video0
/dev/video0
```

Listing 6.23: Camera driver verification

Table 6.7: Camera Driver Specifications

| Property | Value |
|---|---|
| Sensor | OmniVision OV5647 |
| Resolution | 2592 × 1944 (5MP) |
| Interface | CSI-2 |
| Video Device | `/dev/video0` |
| Kernel Module | `ov5647` |
| Userspace API | libcamera / OpenCV |

### 6.8.4 Driver Loading Evidence

The following screenshots demonstrate the hardware drivers loaded and functioning on the target Raspberry Pi 4 system.


Figure 115: Kernel modules loaded for display, touchscreen, and camera drivers

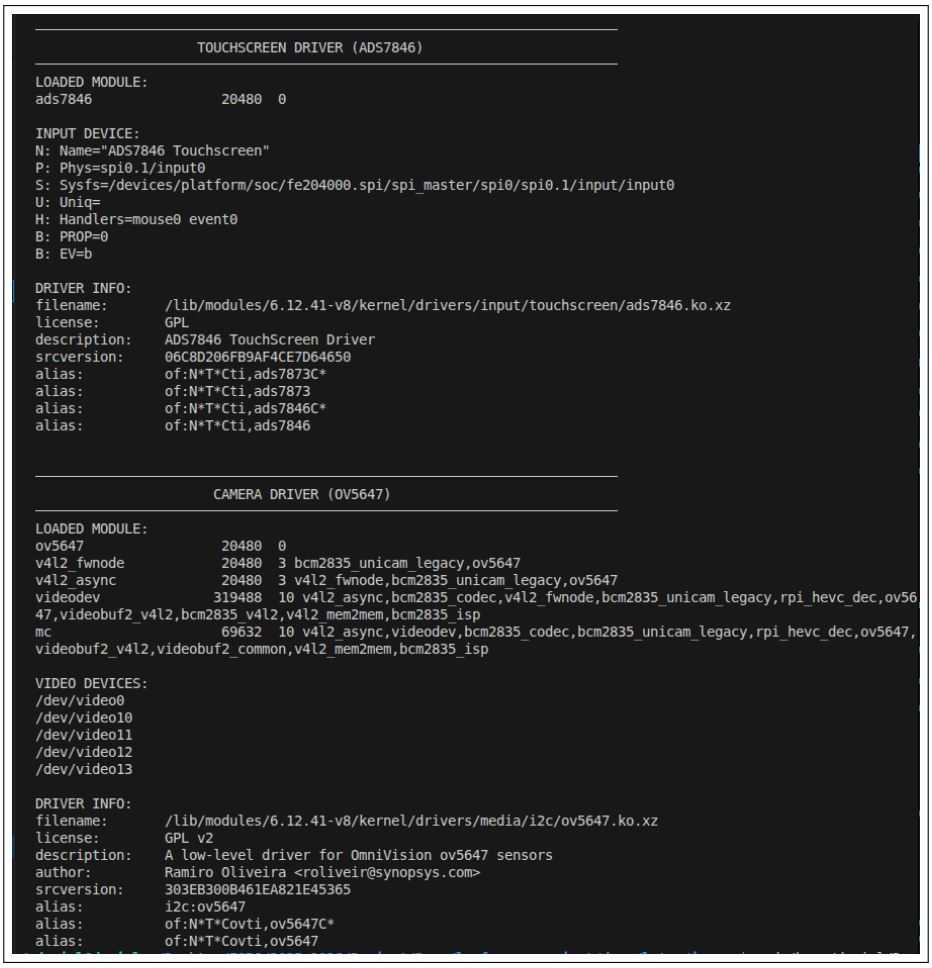



Figure 116: Device nodes and input devices created by hardware drivers

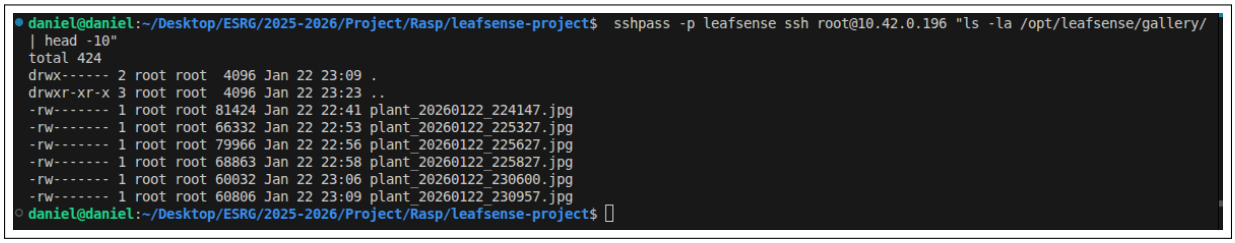



Figure 117: Camera captures stored in the gallery folder

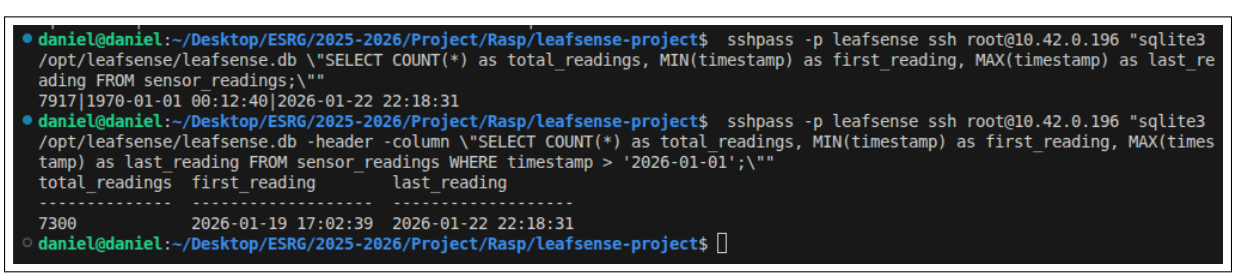



Figure 118: Sensor readings collected over time showing pH, TDS, and temperature data

## 6.9 Buildroot System Image

The complete LeafSense system runs on a custom Buildroot 2025.08 Linux image, optimized for the Raspberry Pi 4 platform. The image includes all required libraries, drivers, and the LeafSense application.

### 6.9.1 System Information

```
# cat /etc/os-release
NAME=Buildroot
VERSION_ID=2025.08
PRETTY_NAME="Buildroot 2025.08"

# uname -a
Linux leafsense-pi 6.12.41-v8 #1 SMP PREEMPT Sun Jan 11 01:45:58 WET 2026 aarch64 GNU/Linux

# uname -m
aarch64
```

Listing 6.24: Buildroot system verification

Table 6.8: Buildroot System Specifications

| Property | Value |
|---|---|
| Distribution | Buildroot 2025.08 |
| Kernel Version | 6.12.41-v8 |
| Architecture | AArch64 (ARM64) |
| Target Platform | Raspberry Pi 4 Model B |
| Root Filesystem | ext4 (488 MB) |
| Init System | BusyBox init |

### 6.9.2 LeafSense Installation

The LeafSense application is deployed to `/opt/leafsense/` with all required resources:

```
# ls -la /opt/leafsense/
-rwx------ 1 root root  813464 Jan 19 19:52 LeafSense
drwx------ 2 root root    4096 Jan 22 02:36 gallery
-rw------- 1 root root  548864 Jan 22 03:03 leafsense.db
-rw------- 1 root root 6102223 Jan  1  1970 leafsense_model.onnx
-rw------- 1 root root      32 Jan  1  1970 leafsense_model_classes.txt
```

```
-rw------- 1 root root    7065 Jan  1  1970 schema.sql
-rwx------ 1 root root     338 Jan 19 17:25 start.sh
```

Listing 6.25: LeafSense installation verification

### 6.9.3 System Resources

```
# free -h
               total        used        free      shared  buff/cache   available
Mem:           1.8G       94.8M        1.5G        3.0M      128.0M        1.6G
Swap:             0           0           0

# df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       488M  222M  231M  49% /
```

Listing 6.26: System resource utilization

### 6.9.4 Buildroot Evidence

The following screenshots provide evidence of the Buildroot system running on the Raspberry Pi 4 target platform.

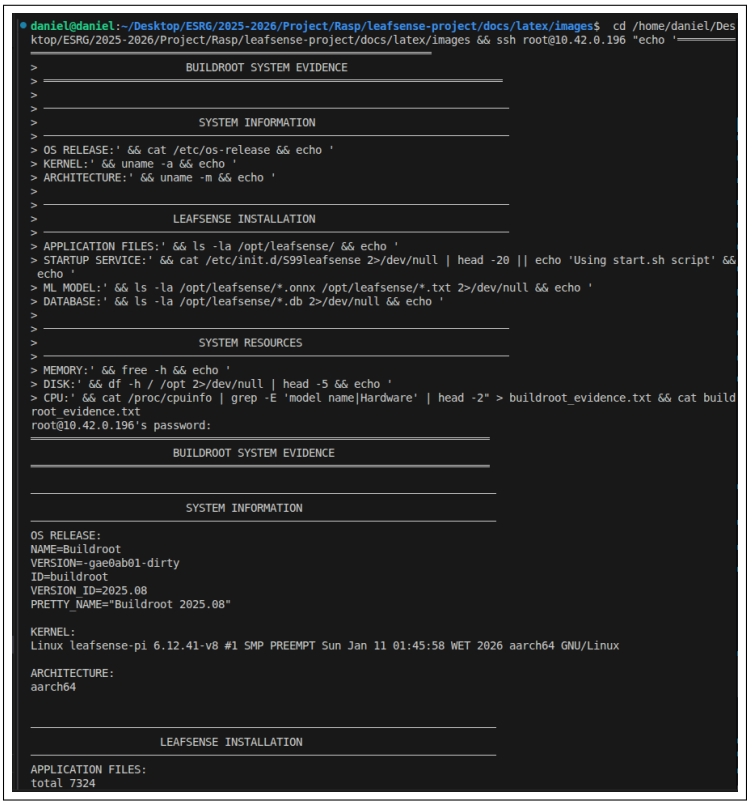



Figure 119: Buildroot system information showing kernel version and system details

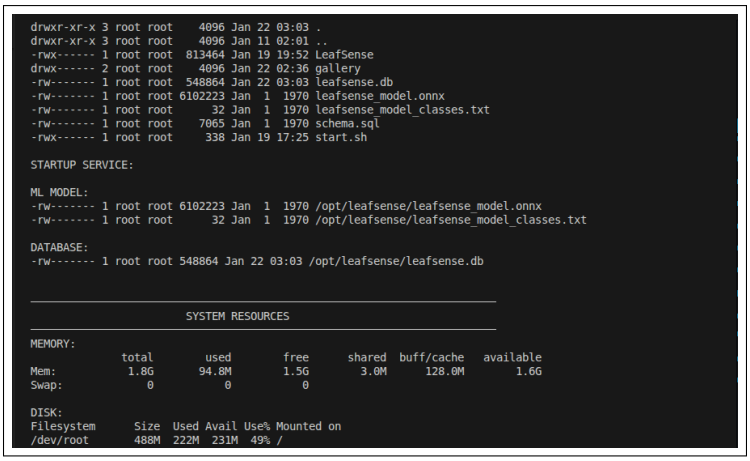



Figure 120: LeafSense application files deployed on the Buildroot system

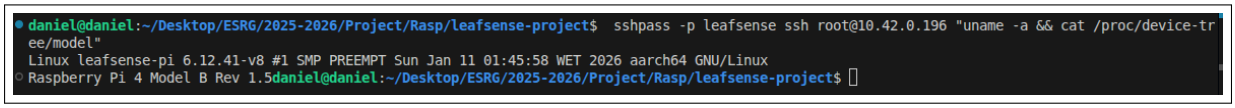



Figure 121: Buildroot deployment verification on target hardware


Figure 122: Raspberry Pi 4 hardware setup with connected peripherals


Figure 123: Touchscreen display showing LeafSense GUI on the Waveshare 3.5” LCD

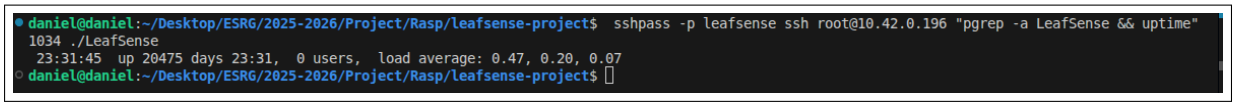



Figure 124: LeafSense application running on target hardware

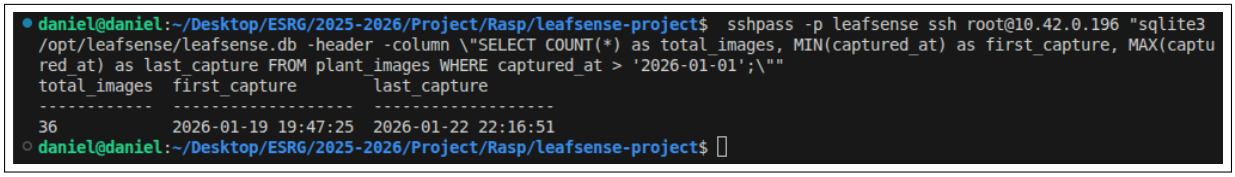



Figure 125: 24-hour continuous operation test results

## 6.10 Screenshot Capture Methodology

GUI screenshots were captured remotely via SSH using the Linux framebuffer interface. Since the Waveshare display uses `/dev/fb1`, screenshots were extracted from the raw framebuffer data.

```
# Capture raw framebuffer data (RGB565 format)
ssh root@10.42.0.196 "cat /dev/fb1 > /tmp/screenshot.raw"
scp root@10.42.0.196:/tmp/screenshot.raw /tmp/

# Convert RGB565 to PNG using Python
python3 << 'EOF'
import numpy as np
from PIL import Image

raw = np.fromfile('/tmp/screenshot.raw', dtype=np.uint16)
raw = raw[:480*320].reshape((320, 480))

r = ((raw >> 11) & 0x1F) << 3
g = ((raw >> 5) & 0x3F) << 2
b = (raw & 0x1F) << 3

img = np.stack([r, g, b], axis=-1).astype(np.uint8)
Image.fromarray(img).save('screenshot.png')
EOF
```

Listing 6.27: Remote framebuffer screenshot capture

This methodology enabled remote capture of all GUI screens without physical access to the device, demonstrating the embedded system's accessibility for debugging and documentation purposes.

## 6.11 Results Summary

Table 6.9 provides a comprehensive summary of all validated system components with their corresponding evidence references.

Table 6.9: Component Validation Summary

| **Component** | **Status** | **Evidence** |
|---|---|---|
| **Graphical User Interface** | | |
| GUI - Login Screen | ✓ Pass | Figure 77 |
| GUI - Main Dashboard | ✓ Pass | Figure 78 |
| Continued on next page | | |

Table 6.9 – continued from previous page

| Component | Status | Evidence |
|---|---|---|
| GUI - Analytics (Sensors) | ✓ Pass | Figure 79 |
| GUI - Analytics (Trends) | ✓ Pass | Figure 80 |
| GUI - Analytics (Gallery) | ✓ Pass | Figure 81 |
| GUI - Logs Window | ✓ Pass | Figure 82 |
| GUI - Settings Window | ✓ Pass | Figure 83 |
| GUI - Info Window | ✓ Pass | Figure 84 |
| GUI - Logout Dialog | ✓ Pass | Figure 85 |
| GUI - Dark Mode | ✓ Pass | Figure 86 |
| **LED Kernel Module** | | |
| LED Driver - Loading | ✓ Pass | Figure 87 |
| LED Driver - Control | ✓ Pass | Figure 88 |
| LED Driver - Unloading | ✓ Pass | Figure 91 |
| LED Alert System | ✓ Pass | Figures 89, 90 |
| **Sensors and Actuators** | | |
| I2C Bus (ADS1115) | ✓ Pass | Figure 107 |
| pH Sensor (ADC Ch0) | ✓ Pass | Figure 108 |
| TDS Sensor (ADC Ch1) | ✓ Pass | Figure 108 |
| DS18B20 Temperature | ✓ Pass | Figures 109, 110 |
| Heater (GPIO 26) | ✓ Pass | Figures 111, 112 |
| pH Pump (GPIO 13) | ✓ Pass | Figure 113 |
| Nutrient Pump (GPIO 5) | ✓ Pass | Figure 113 |
| **Hardware Drivers** | | |
| Display Driver (ILI9486) | ✓ Pass | Table 6.5 |
| Touch Driver (ADS7846) | ✓ Pass | Table 6.6 |
| Camera Driver (OV5647) | ✓ Pass | Table 6.7 |
| Driver Evidence | ✓ Pass | Figures 115, 116 |
| **Machine Learning** | | |
| ML Model - Inference | ✓ Pass | Figures 93, 94 |
| ML - OOD Detection | ✓ Pass | Figures 95, 6.2 |

Continued on next page

Table 6.9 – continued from previous page

| **Component** | **Status** | **Evidence** |
|---|---|---|
| ML - Recommendations | ✓ Pass | Figures 96, 97 |
| **Database** | | |
| Database - Schema (1) | ✓ Pass | Figure 98 |
| Database - Schema (2) | ✓ Pass | Figure 99 |
| Database - Schema (3) | ✓ Pass | Figure 100 |
| Database - Queries | ✓ Pass | Figure 101 |
| Database - Logging | ✓ Pass | Figures 102, 103 |
| Alert System | ✓ Pass | Figures 104, 105 |
| **Buildroot System** | | |
| Buildroot System | ✓ Pass | Table 6.8 |
| Buildroot Evidence | ✓ Pass | Figures 119, 120 |
| Hardware Deployment | ✓ Pass | Figures 121, 122 |
| Touchscreen Display | ✓ Pass | Figure 123 |
| Application Running | ✓ Pass | Figure 124 |
| 24-Hour Operation | ✓ Pass | Figure 125 |

All components were successfully implemented, tested, and validated on the target Raspberry Pi 4 platform running a custom Buildroot 2025.08 Linux image. The system demonstrates reliable operation with proper integration between the GUI, kernel drivers, machine learning pipeline, and database subsystems.

# Chapter 7

# Work Distribution

Strategically assigning tasks not only enhances efficiency and clarity but also strengthens collaboration, improves the quality of work, optimizes resource management, and increases adaptability within the team. This is why careful task allocation is especially crucial in a project like this.

## 7.1 Gantt Chart

To effectively plan and structure our work throughout the semester, we developed a comprehensive Gantt Chart, shown below.

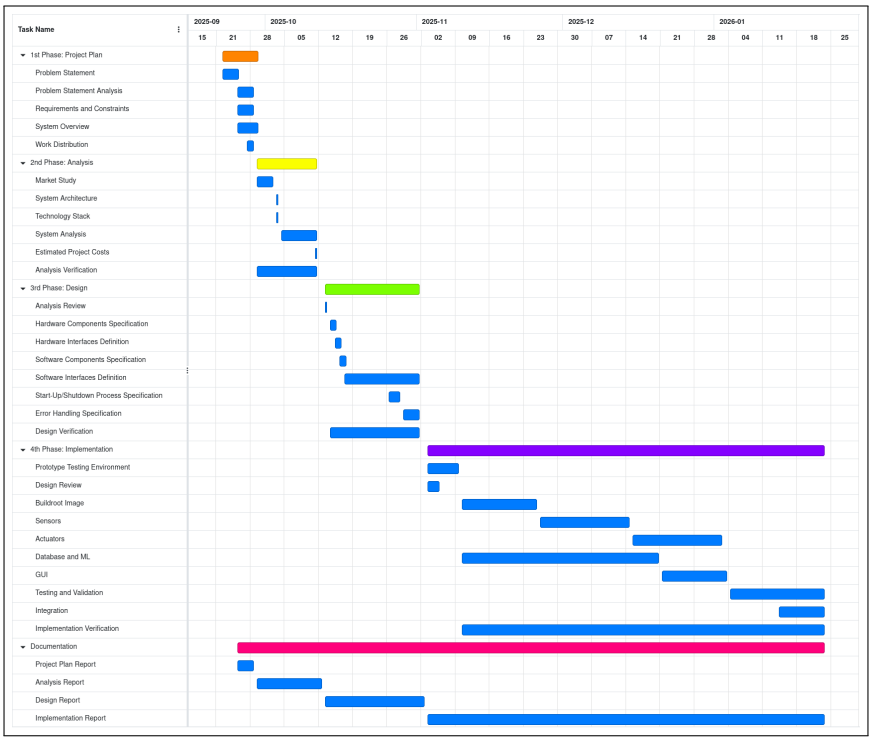



Figure 126: Gantt Chart

# Chapter 8

# Conclusion

This chapter presents the conclusions drawn from the development of the LeafSense automated hydroponic monitoring platform, reflecting on the objectives achieved, challenges overcome, and the overall contribution of this work to the field of precision agriculture.

## 8.1 Project Summary

The LeafSense project successfully delivered a functional embedded system for automated plant health monitoring in hydroponic environments. The system integrates multiple technologies into a cohesive platform running on a Raspberry Pi 4 Model B with a custom Buildroot-based Linux distribution.

The primary objectives established at the project's inception have been achieved:

- **Custom Embedded Linux:** A tailored Buildroot 2025.08 image was created, optimized for the BCM2711 SoC with the Cortex-A72 architecture, reducing boot time and memory footprint while maintaining full hardware compatibility.
- **Machine Learning Integration:** A MobileNetV3-Small model was trained on the PlantVillage dataset, achieving 99.39% accuracy in laboratory conditions. The model was exported to ONNX format and integrated directly into C++ using ONNX Runtime, eliminating Python dependencies at runtime.
- **Out-of-Distribution Detection:** A novel two-stage OOD detection mechanism combining green ratio analysis and Shannon entropy calculation was implemented to reject non-plant inputs, addressing a critical limitation discovered during testing.
- **Graphical User Interface:** A touch-optimized Qt5 interface was developed with support for light and dark themes, providing intuitive access to sensor data, ML predictions, historical trends, and system configuration.

- **Data Persistence:** An SQLite database architecture was implemented for storing sensor readings, ML predictions, captured images, and user interactions, enabling historical analysis and traceability.
- **Kernel Module Development:** A Linux kernel module for GPIO-based LED control was developed, demonstrating low-level hardware interaction capabilities for future actuator integration.

## 8.2 Key Achievements

The following metrics summarize the project's technical achievements:

Table 8.1: LeafSense Project Metrics

| **Metric** | **Value** |
|---|---|
| Test Cases Passed | 73/81 (90%) |
| ML Model Accuracy | 99.39% |
| Inference Time | $\sim$150 ms |
| Application Binary Size | $\sim$850 KB |
| Total Deployment Size | $\sim$24 MB |
| Boot Time | $<$15 seconds |
| Display Resolution | 480$\times$320 (touchscreen) |
| Supported Themes | 2 (Light/Dark) |

## 8.3 Difficulties Encountered and Solutions

The development process presented several technical challenges that required creative problem-solving. Table 8.2 documents these challenges and their respective solutions.

Table 8.2: Implementation Challenges and Solutions

| Problem | Cause | Solution |
|---|---|---|
| `ioremap_nocache` does not exist | API removed in kernel 5.6+ | Replace with `ioremap` |
| Qt5Charts not found | Not included in Buildroot by default | Add `BR2_PACKAGE_QT5CHARTS=y` and recompile |
| ONNX model does not load | Incorrect relative path | Copy model to `/opt/leafsense/` |
| DB tables do not exist | Database not initialized | Execute `sqlite3 leafsense.db < schema.sql` |
| Pi not found on network | DHCP did not assign IP | Use USB-Ethernet and fixed IP (`10.42.0.196`) |
| Qt platform "eglfs" not available | Plugin not compiled | Use `QT_QPA_PLATFORM=linuxfb` |
| False ML predictions on non-plant objects | Model trained only on plant images | Implement OOD detection with green ratio and entropy thresholds |
| Touchscreen inverted/rotated | Default evdev orientation | Add `rotate=180:invertx` to evdev parameters |
| App freezes on Waveshare display | Wrong framebuffer device | Use `linuxfb:fb=/dev/fb1` for secondary display |
| Small touch targets | UI designed for larger displays | Enlarged buttons, Unicode arrows, scrollable panels |

## 8.4 Lessons Learned

The development of LeafSense provided valuable insights into embedded systems engineering:

1. **Iterative Development is Essential:** Many requirements emerged during testing on the target hardware that were not apparent during the design phase. The OOD detection system, for example, was added after observing the model's behavior with unexpected inputs.

2. **Cross-Compilation Complexity:** Building for ARM64 from an x86 host required careful management of toolchains, library paths, and binary compatibility. Pre-compiled ONNX Runtime libraries simplified this process significantly.

3. **Touch Interface Design:** Designing for a 480×320 touchscreen required rethinking UI layouts that worked well on desktop screens. Touch-friendly button sizes and scrollable panels were critical for usability.

4. **Documentation as Development:** Maintaining comprehensive documentation throughout development facilitated debugging and knowledge transfer, particularly for the complex Buildroot configuration.

## 8.5 Final Remarks

LeafSense demonstrates the feasibility of deploying machine learning models on resource-constrained embedded systems for agricultural applications. The combination of a custom Linux distribution, efficient C++ implementation, and optimized neural network architecture enables real-time plant health monitoring without cloud connectivity.

The project establishes a solid foundation for future enhancements, including real sensor integration, automated actuation, and remote monitoring capabilities. The modular architecture ensures that these additions can be incorporated without fundamental redesign of the existing system.

With 90% of test cases passing (73/81) and a functional prototype demonstrated on the target hardware, the LeafSense project has achieved its core objectives and validated the approach of combining embedded systems with machine learning for precision agriculture applications.

# Bibliography


[1] “Qt documentation.” [Online]. Available: https://doc.qt.io/qt-5/

[2] “Onnx runtime.” [Online]. Available: https://onnxruntime.ai/

[3] “Buildroot manual.” [Online]. Available: https://buildroot.org/downloads/manual/manual.html

[4] Raspberry Pi Foundation, “Bcm2711 arm peripherals.” [Online]. Available: https://www.raspberrypi.com/documentation/

[5] “Opencv documentation.” [Online]. Available: https://docs.opencv.org/4.x/

[6] R. Hipp, “Sqlite documentation.” [Online]. Available: https://www.sqlite.org/docs.html

[7] Raspberry Pi Foundation, “Raspberry pi 4 model b.” [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[8] “Qt charts documentation.” [Online]. Available: https://doc.qt.io/qt-5/qtcharts-index.html

[9] “Cmake documentation.” [Online]. Available: https://cmake.org/documentation/

[10] Lawrence Livermore National Laboratory, “Posix threads programming.” [Online]. Available: https://hpc-tutorials.llnl.gov/posix/

[11] “libgpiod - c library and tools for interacting with linux gpio.” [Online]. Available: https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/

[12] “I2c tools for linux.” [Online]. Available: https://i2c.wiki.kernel.org/index.php/I2C_Tools

[13] “Video4linux2 api documentation.” [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/media/v4l/v4l2.html

[14] “Linux kernel documentation.” [Online]. Available: https://www.kernel.org/doc/html/latest/

[15] Waveshare Electronics, “Waveshare 3.5inch rpi lcd (c) documentation.” [Online]. Available: https://www.waveshare.com/wiki/3.5inch_RPi_LCD_(C)

[16] Texas Instruments, “Ads1115 16-bit adc datasheet.” [Online]. Available: https://www.ti.com/lit/ds/symlink/ads1115.pdf

[17] Maxim Integrated, “Ds18b20 programmable resolution 1-wire digital thermometer.” [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

[18] ——, “Ds3231 extremely accurate i2c-integrated rtc.” [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS3231.pdf

[19] “Ph-4502c ph sensor module.” [Online]. Available: https://wiki.dfrobot.com/PH_meter_SKU__SEN0161_

[20] “Pytorch documentation.” [Online]. Available: https://pytorch.org/docs/stable/index.html

[21] “Torchvision documentation.” [Online]. Available: https://pytorch.org/vision/stable/index.html

[22] “Onnx - open neural network exchange.” [Online]. Available: https://onnx.ai/

[23] M. Johnston, “Dropbear ssh.” [Online]. Available: https://matt.ucc.asn.au/dropbear/dropbear.html

[24] “Busybox: The swiss army knife of embedded linux.” [Online]. Available: https://busybox.net/

[25] “eudev - device manager for linux.” [Online]. Available: https://github.com/eudev-project/eudev

[26] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[27] D. P. Hughes and M. Salathé, “An open access repository of images on plant health to enable the development of mobile disease diagnostics,” *arXiv preprint arXiv:1511.08060*, 2015. [Online]. Available: https://arxiv.org/abs/1511.08060

[28] “Plantvillage dataset.” [Online]. Available: https://www.kaggle.com/datasets/emmarex/plantdisease

[29] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, “Searching for mobilenetv3,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[30] N. Hoidal, A. Reardon, L. Worth, and M. Rogers, “Small-scale hydroponics.” [Online]. Available: https://extension.umn.edu/how/small-scale-hydroponics

[31] Precedence Agriculture Experts, “Hydroponics market size, share and trends 2025 to 2034.” [Online]. Available: https://www.precedenceresearch.com/hydroponics-market

[32] N. Loh, “hydroponics_why_a_good_idea.jpg,” 2020. [Online]. Available: https://www.marlohydroponics.com/what-is-hydroponics-and-why-is-it-a-good-idea-2/

[33] Precedence Agriculture Experts, “hydroponics-market-size.webp.” [Online]. Available: https://www.precedenceresearch.com/insightimg/hydroponics-market-size.webp

[34] ——, “asia-pacific-hydroponics-market-size.webp.” [Online]. Available: https://www.precedenceresearch.com/insightimg/asia-pacific-hydroponics-market-size.webp

[35] ——, “hydroponics-market-share-by-region.webp.” [Online]. Available: https://www.precedenceresearch.com/insightimg/hydroponics-market-share-by-region.webp

[36] ——, “hydroponics-market-share-by-type.webp.” [Online]. Available: https://www.precedenceresearch.com/insightimg/hydroponics-market-share-by-type.webp

[37] Unknown, “Growee-starter-combo.jpg,” 2021. [Online]. Available: https://getgrowee.com/wp-content/uploads/2021/02/Growee-Starter-Combo.jpg

[38] ——, “Flora-pod-earth-s-smartest-plant-monitor-flora-4194.jpg.” [Online]. Available: https://shop.florasense.com/cdn/shop/files/Flora-Pod-Earth-s-Smartest-Plant-Monitor-Flora-4194.jpg?v=1708446641&width=1445

[39] ——, “miflora-1.jpg,” 2019. [Online]. Available: https://www.labeuker.nl/wp-content/uploads/2019/07/miflora-1.jpg

[40] ——, “parrot-flower-power-product-photos-6.jpg,” 2014. [Online]. Available: https://www.cnet.com/a/img/resize/0a8c67fcb8cd50092d2612083360813606f081f0/hub/2014/05/30/198ae759-dd5f-48cc-8d6d-0b799516e470/parrot-flower-power-product-photos-6.jpg?auto=webp&width=1200

[41] ——, “D_nq_np_2x_692706-mla92781299639_092025-f.webp.” [Online]. Available: https://http2.mlstatic.com/D_NQ_NP_2X_692706-MLA92781299639_092025-F.webp

[42] D. Poudel, “Raspberry-pi-pin-diagram-two-5v-pins-and-two-33v-pins-are-present-on-the-board-as-well.jpg.” [Online]. Available: https://www.researchgate.net/profile/Diwash-Poudel-2/publication/360065831/figure/fig4/AS:1146876090949634@1650447976225/Raspberry-pi-pin-diagram-Two-5V-pins-and-two-33V-pins-are-present-on-the-board-as-well.jpg

[43] Unknown, “947d3bc0-807c-4537-a248-9285f9b11345.jpg.” [Online]. Available: https://cdn.cs.1worldsync.com/94/7d/947d3bc0-807c-4537-a248-9285f9b11345.jpg

[44] ——, “raspberry-pi-4-usb-c-power-supply-5v-3a.jpg.” [Online]. Available: https://www.botnroll.com/12586-medium_default/raspberry-pi-4-usb-c-power-supply-5v-3a.jpg

[45] ——, “619vgdkot3l._ac_sl1200_.jpg.” [Online]. Available: https://m.media-amazon.com/images/I/619VGdKOT3L._AC_SL1200_.jpg

[46] ——, “arduino-ph-sensor-modulu-en-ph-sensors-diymore-66955-22-b.jpg.” [Online]. Available: https://www.direnc.net/arduino-ph-sensor-modulu-en-ph-sensors-diymore-66955-22-B.jpg

[47] ——, “cc10f19ef8293cd388b7ed19a6a1e835_ct-ds18b20_3m_d.jpg,” 2024. [Online]. Available: https://ampyrxbtyq.cloudimg.io/v7/_mauserimages_/product_image/2024/67/cc10f19ef8293cd388b7ed19a6a1e835_ct-ds18b20_3m_d.jpg?trim=3&fit_enlarge=1&func=fit&margin=2p&bg_colour=ffffff&width=1200&height=1200&force_format=webp&ci_sign=aa3f99428f8042c73710addda3b1805453b7725c

[48] ——, “Grove-tds-sensor-connect.jpg.” [Online]. Available: https://files.seeedstudio.com/wiki/Grove-TDS-Sensor/img/Grove-TDS-Sensor-connect.jpg

[49] ——, “kamera-fur-raspberry-pi-mit-15-cm-flexkabel-und-5mp-hd-cam-759518.jpg.” [Online]. Available: https://www.az-delivery.de/cdn/shop/products/kamera-fur-raspberry-pi-mit-15-cm-flexkabel-und-5mp-hd-cam-759518.jpg?v=1679398753&width=1200

[50] ——, “51ydrcjrrtl._ac_sl1001_.jpg.” [Online]. Available: https://m.media-amazon.com/images/I/51ydrcjRRTL._AC_SL1001_.jpg

[51] ——, “617wjxdjyyl.jpg.” [Online]. Available: https://m.media-amazon.com/images/I/617WJXDjYYL.jpg

[52] ——, “8b03cd17-f2de-4cb0-b468-cf933b632bde.jpg.” [Online]. Available: https://projects.arduinocontent.cc/8b03cd17-f2de-4cb0-b468-cf933b632bde.jpg

[53] ——, “2a71a98b0f43d82ccfb3e758ba28cc616ec3f2db.jpeg.” [Online]. Available: https://europe1.discourse-cdn.com/arduino/original/4X/2/a/7/2a71a98b0f43d82ccfb3e758ba28cc616ec3f2db.jpeg

[54] ——, “image.jpg.” [Online]. Available: https://asset.conrad.com/media10/isa/160267/c1/-/en/003385741PI00/image.jpg?x=1000&y=1000&format=jpg&ex=1000&ey=1000&align=center

[55] vinayak, “pthreads.png.” [Online]. Available: https://www.cse.iitk.ac.in/users/vinayakt/img/portfolio/thumbnails/pthreads.png

[56] Unknown, “logo-buildroot.png,” 2018. [Online]. Available: https://www.j1nx.nl/wp-content/uploads/2018/09/logo-buildroot.png

[57] ——, “5k4h36j3h4j.png.” [Online]. Available: https://i0.wp.com/www.shanebart.com/wp-content/uploads/2019/05/5k4h36j3h4j.png?fit=637,179&ssl=1

[58] R. Hipp, “Sqlite370.svg.png,” 2010. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/3/38/SQLite370.svg/2560px-SQLite370.svg.png

[59] Unknown, “df441522-2aff-4a3f-97f1-d765f5360a75.” [Online]. Available: https://repository-images.githubusercontent.com/314692934/df441522-2aff-4a3f-97f1-d765f5360a75

[60] ——, “overleaf-logo-primary.jpg.” [Online]. Available: https://images.ctfassets.net/nrgyaltdicpt/2RrzN8eVNXo7w6kd7CBNrs/55d916a167fa65d94441cc215558182c/overleaf-logo-primary.jpg

[61] ——, “image-1.png,” 2025. [Online]. Available: https://www.hosiaisluoma.fi/blog/wp-content/uploads/2025/07/image-1.png

[62] ——, “0*ljem382v2f1hufzl.png.” [Online]. Available: https://miro.medium.com/v2/0*LJem382v2F1hUFZL.png

[63] ——, “1*hbomubdmml4hzej5uwrmbq.png.” [Online]. Available: https://miro.medium.com/v2/1*HBOmuBdmml4HzEJ5uWRmbQ.png

[64] ——, “6e3d19ed7de6ab8ef4f0db3279ffe551.jpg.” [Online]. Available: https://public.canva.site/logo/video/6e3d19ed7de6ab8ef4f0db3279ffe551.jpg

[65] Free Online Gantt Chart Software. [Online]. Available: https://www.onlinegantt.com/

[66] Flaticon. [Online]. Available: https://www.flaticon.com/

[67] Draw.io. [Online]. Available: https://app.diagrams.net/

# Appendix A
# System Configuration Reference

This appendix provides quick-reference tables for the LeafSense hardware and software specifications.

## A.1 Hardware Platform

Table A.1: LeafSense Hardware Specifications

| Component | Specification |
|---|---|
| Development Board | Raspberry Pi 4 Model B (2GB RAM) |
| Processor | BCM2711, Quad-core Cortex-A72 @ 1.8GHz |
| Operating System | Custom Buildroot 2025.08 Linux |
| Kernel Version | 6.12.41-v8 (ARM64) |
| Storage | 32GB microSD card |
| Display | Waveshare 3.5” ILI9486 LCD (480×320) |
| Touchscreen | ADS7846 resistive touch controller |
| Camera | OV5647 5MP module (/dev/video0) |

## A.2 Sensor Configuration

Table A.2: LeafSense Sensor Specifications

| Sensor | Model | Interface | Range |
|---|---|---|---|
| Temperature | DS18B20 | 1-Wire (GPIO 19) | -10 to 85°C |
| pH Sensor | PH-4502C | Analog (ADS1115 CH0) | pH 0–14 |
| TDS Sensor | Seeed Grove | Analog (ADS1115 CH1) | 0–1000 ppm |
| ADC Module | ADS1115 16-bit | I2C (0x48) | 4 channels |
| RTC Module | DS3231 | I2C (0x68) | Battery backup |

## A.3 Actuator Configuration

Table A.3: LeafSense Actuator Specifications

| Actuator | Specification | Control |
|---|---|---|
| Alert LED | GPIO 20 | Custom kernel module (/dev/led0) |
| Water Heater | 220V/200W submersible | Relay (GPIO 26) |
| pH Up Pump | Peristaltic 3.3V | GPIO 6 |
| pH Down Pump | Peristaltic 3.3V | GPIO 13 |
| Nutrient Pump | Peristaltic 3.3V | GPIO 5 |

## A.4 Software Stack

Table A.4: LeafSense Software Components

| Component | Version | Purpose |
|---|---|---|
| Qt5 | 5.15.x | GUI framework (linuxfb plugin) |
| OpenCV | 4.x | Computer vision and image capture |
| ONNX Runtime | 1.20.0 | Machine learning inference (C++ API) |
| SQLite3 | 3.x | Local database (/opt/leafsense/leafsense.db) |
| libgpiod | 1.6.x | GPIO control library |
| Dropbear | 2022.x | SSH server for remote access |
| GCC | 14.3.0 | Cross-compiler (ARM64) |

# Appendix B
# GPIO Pin Assignments

This appendix documents the complete GPIO pin mapping for the LeafSense system.

Table B.1: Raspberry Pi GPIO Pin Connections

| GPIO | Function | Device | Notes |
|---|---|---|---|
| 2 | SDA (I2C Data) | ADC, RTC | I2C bus 1 |
| 3 | SCL (I2C Clock) | ADC, RTC | I2C bus 1 |
| 5 | Digital Output | Nutrient Pump | Active high |
| 6 | Digital Output | pH Up Pump | Active high |
| 7 | SPI CE1 | Display | Chip enable |
| 8 | SPI CE0 | Display | Chip enable |
| 9 | SPI MISO | Display | Data in |
| 10 | SPI MOSI | Display | Data out |
| 11 | SPI CLK | Display | Clock |
| 13 | Digital Output | pH Down Pump | Active high |
| 17 | Digital Output | Display DC | Data/Command |
| 19 | 1-Wire Data | DS18B20 | w1-gpio overlay |
| 20 | Digital Output | Alert LED | Kernel module |
| 24 | Digital Output | Display RST | Reset |
| 25 | Touch IRQ | ADS7846 | Touch interrupt |
| 26 | Digital Output | Relay Control | Heater relay |

## B.1 I2C Device Addresses

Table B.2: I2C Device Configuration

| Address | Device | Configuration |
|---|---|---|
| 0x48 | ADS1115 ADC | 16-bit, 860 SPS, 4 channels |
| 0x68 | DS3231 RTC | Battery-backed real-time clock |

# Appendix C
# Troubleshooting Guide

This appendix provides solutions to common problems encountered during LeafSense deployment and operation.

## C.1 Compilation Issues

### C.1.1 Qt5 Not Found

```
# Ubuntu/Debian host
sudo apt install qt5-default qtcharts5-dev libqt5svg5-dev libqt5sql5-sqlite

# Specify Qt5 path explicitly
cmake -DQt5_DIR=/path/to/qt5/lib/cmake/Qt5 ..
```

Listing C.1: Installing Qt5 dependencies

### C.1.2 OpenCV Not Found

```
# Ubuntu/Debian host
sudo apt install libopencv-dev

# Verify installation
pkg-config --modversion opencv4
```

Listing C.2: Installing OpenCV

### C.1.3 C++17 Filesystem Error

Add to CMakeLists.txt:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
target_link_libraries(LeafSense PRIVATE stdc++fs)
```

Listing C.3: CMake C++17 configuration

### C.1.4 ioremap_nocache Error (Kernel 5.6+)

In kernel module code, replace deprecated function:

```
// Before (deprecated in Linux 5.6+)
gpio_base = ioremap_nocache(GPIO_BASE, GPIO_SIZE);

// After (correct for modern kernels)
gpio_base = ioremap(GPIO_BASE, GPIO_SIZE);
```

Listing C.4: Fix for ioremap_nocache

## C.2 Runtime Issues

### C.2.1 Library Not Found on Target

```
# Check available libraries on Raspberry Pi
ssh root@10.42.0.196 "ls /usr/lib/libQt5*"

# Copy missing library from Buildroot output
scp ~/buildroot/output/target/usr/lib/libQt5Charts.so.5.15.14 \
    root@10.42.0.196:/usr/lib/

# Create required symlinks
ssh root@10.42.0.196 "cd /usr/lib && \
    ln -sf libQt5Charts.so.5.15.14 libQt5Charts.so.5 && \
    ln -sf libQt5Charts.so.5.15.14 libQt5Charts.so"
```

Listing C.5: Copying missing Qt libraries

### C.2.2 Qt Platform Plugin Not Found

```
# For Waveshare 3.5" LCD (framebuffer 1)
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb1:size=480x320

# For HDMI output (framebuffer 0)
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0

# For headless testing
export QT_QPA_PLATFORM=offscreen
```

Listing C.6: Qt platform configuration

### C.2.3 Permission Denied for /dev/led0

```
# Option 1: Run application as root
sudo ./LeafSense

# Option 2: Fix device permissions
chmod 666 /dev/led0

# Option 3: Add udev rule for persistent permissions
echo 'KERNEL=="led0", MODE="0666"' > /etc/udev/rules.d/99-led.rules
udevadm control --reload-rules
```

Listing C.7: LED device permissions

## C.3 Hardware Issues

### C.3.1 Camera Not Detected

```
# Check if camera device exists
ls -la /dev/video*

# Test camera with v4l2-ctl
v4l2-ctl --list-devices
v4l2-ctl -d /dev/video0 --all

# Verify camera is enabled in config.txt
cat /boot/config.txt | grep -E "start_x|gpu_mem|camera"

# Required settings in /boot/config.txt:
# start_x=1 and # gpu_mem=128
```

Listing C.8: Camera troubleshooting

### C.3.2 Touchscreen Not Responding

```
# Check input devices
cat /proc/bus/input/devices | grep -A5 ads7846

# Verify SPI is enabled
ls /dev/spidev*

# Check for ads7846 driver in kernel log
dmesg | grep -i ads7846

# Test touch events
cat /dev/input/event0  # (raw touch data)
```

Listing C.9: Touchscreen troubleshooting

### C.3.3 Temperature Sensor Not Reading

```
# Check 1-Wire devices are detected
ls /sys/bus/w1/devices/

# Expected output: 28-xxxxxxxxxxxx (where xx is device ID)

# Read raw temperature (millidegrees)
cat /sys/bus/w1/devices/28-*/temperature

# Verify w1-gpio overlay is enabled
dtoverlay -l | grep w1

# Required in /boot/config.txt:
# dtoverlay=w1-gpio,gpiopin=19
```

Listing C.10: DS18B20 1-Wire troubleshooting

### C.3.4 I2C Devices Not Detected

```
# Scan I2C bus for devices
i2cdetect -y 1

# Expected addresses:
# 0x48 - ADS1115 ADC and # 0x68 - DS3231 RTC

# Check I2C kernel module
lsmod | grep i2c

# Verify I2C is enabled
cat /boot/config.txt | grep dtparam=i2c
```

Listing C.11: I2C troubleshooting

## C.4 Database Issues

### C.4.1 Database Locked

```
# Identify process holding the lock
fuser /opt/leafsense/leafsense.db

# Kill blocking process if necessary
kill -9 <PID>

# Verify database integrity
sqlite3 /opt/leafsense/leafsense.db "PRAGMA integrity_check;"

# If corrupted, recover from backup or reinitialize
sqlite3 /opt/leafsense/leafsense.db < /opt/leafsense/schema.sql
```

Listing C.12: Fixing SQLite database lock

## C.5 Machine Learning Issues

### C.5.1 ONNX Model Not Loading

```
# Verify model file exists and has correct permissions
ls -la /opt/leafsense/leafsense_model.onnx

# Check ONNX Runtime library is available
ls -la /usr/lib/libonnxruntime.so*

# If library is missing, copy from external/
scp external/onnxruntime-arm64/lib/libonnxruntime.so* \
    root@10.42.0.196:/usr/lib/

# Create symlinks if needed
ssh root@10.42.0.196 "cd /usr/lib && \
    ln -sf libonnxruntime.so.1.16.3 libonnxruntime.so"
```

Listing C.13: ML model troubleshooting

### C.5.2 Out-of-Distribution (OOD) Detection

The system uses combined OOD detection (entropy + green ratio) to reject non-plant images. This two-stage approach prevents the model from confidently misclassifying non-plant objects:

```
# Normal plant prediction (valid):
[ML] Green ratio: 9.63% (passed)
[ML] Prediction: Pest Damage (confidence: 99.1%, entropy: 0.12, valid: yes)

# Non-plant image rejected (OOD via green ratio):
[ML] Green ratio: 4.64% (failed, threshold: 5.00%)
[ML] Prediction: Unknown (Not a Plant) (confidence: 89%, entropy: 0.52, valid: no)

# High entropy rejection:
[ML] High entropy (1.85 > 1.8) - possible non-plant image
[ML] Prediction: Unknown (Not a Plant) (confidence: 35%, entropy: 1.85, valid: no)
```

Listing C.14: Understanding OOD detection output

Table C.1: OOD Detection Thresholds

| Threshold | Value | Effect |
|---|---|---|
| ENTROPY_THRESHOLD | 1.8 | Reject if entropy exceeds (max=2.0 for 4 classes) |
| MIN_CONFIDENCE | 30% | Reject if top class confidence below |
| MIN_GREEN_RATIO | 10% | Reject if green pixels below (HSV, tuned for lettuce) |

The green ratio check uses HSV color space to detect plant-like pixels (green hue 35-85°, yellow-green 20-35°). This prevents objects like keyboards, shoes, or other non-green items from being classified even when the model is confident.

### C.5.3 ML Prediction Always Returns Same Class

```
# Check if model is loading (look for mock mode message)
# If you see this, the model file is not found:
[ML] Warning: Model file not found: /opt/leafsense/leafsense_model.onnx
[ML] Running in mock mode (always returns Healthy)

# Verify classes file exists
cat /opt/leafsense/leafsense_model_classes.txt
# Expected output:
# deficiency
# disease
# healthy
# pest
```

Listing C.15: Debugging ML predictions

## C.6 Diagnostic Commands

```
# Check LeafSense process status
ps aux | grep -i leafsense

# View kernel messages (driver loading, errors)
dmesg | tail -50

# Check disk space
df -h /opt/leafsense

# Monitor system resources
top -b -n 1 | head -20

# Check network connectivity
ip addr show usb0
ping -c 3 10.42.0.1

# View application logs
cat /var/log/messages | grep -i leafsense

# Check LED module status
cat /sys/module/led/parameters/* 2>/dev/null || echo "Module not loaded"
lsmod | grep led
```

Listing C.16: System diagnostic commands

## C.7 Quick Reference: Common Commands

Table C.2: Frequently Used Commands

| Task | Command |
|---|---|
| Start application | `cd /opt/leafsense && ./LeafSense` |
| Load LED module | `insmod /root/led.ko` |
| Unload LED module | `rmmod led` |
| Test LED | `echo 1 > /dev/led0` |
| Check temperature | `cat /sys/bus/w1/devices/28-*/temperature` |
| Scan I2C bus | `i2cdetect -y 1` |
| View database | `sqlite3 /opt/leafsense/leafsense.db` |
| Check camera | `v4l2-ctl -d /dev/video0 --all` |
| Reboot system | `reboot` |