POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Smart Plants: AI-driven Orchestration for Scalable Plant Care

Author:
Giulio Saulle

Student ID:
10626444

Advisor:
Prof. William Fornaciari

Academic Year:
2025–2026

Dedicated to my family and friends, to my loved one and to the ones I love, wherever they are.

Abstract

Modern precision agriculture increasingly depends on reliable data flows between distributed sensors and intelligent control nodes. This thesis presents the latest iteration of *Smart Plants*, a centrally orchestrated platform in which a Raspberry Pi gateway coordinates a set of ESP32-based sensor and actuator nodes. The document analyses requirements drawn from literature, industry and study groups, motivates the choice of a hub-and-spoke topology, and details the implementation of a Flask middleware that delivers live dashboards, automated irrigation, and AI-assisted analytics. A refreshed architecture delegates more autonomy to a central node, demonstrating improvements in reliability, scalability, and maintainability. Then, an analysis on the data collected and fed to multiple types of machine learning models showcases the potential for predictive plant care and anomaly detection, based on a specific sensor called Bioristor, that is used as ground truth to validate the obtained model correctness. The thesis concludes with reflections on lessons learned during development and suggestions for future enhancements.

Keywords: IoT, orchestrator, nodes, hub-and-spoke, automated, AI, machine learning, models, Agricola 2000, Bioristor

Abstract in lingua italiana

L'agricoltura di precisione moderna dipende sempre più da flussi di dati affidabili tra sensori distribuiti e nodi di controllo intelligenti. Questa tesi presenta l'ultima iterazione di Smart Plants, una piattaforma orchestrata centralmente in cui un gateway Raspberry Pi coordina un set di nodi sensori e attuatori basati su ESP32. Il documento analizza i requisiti tratti dalla letteratura, dagli attori commerciali e dai gruppi di studio, motiva la scelta di una topologia hub-and-spoke e dettaglia l'implementazione di un middleware Flask che fornisce dashboard in tempo reale, irrigazione automatizzata e analisi assistita dall'intelligenza artificiale. Una architettura rinnovata delega maggiore autonomia a un nodo principale, dimostrando miglioramenti in termini di affidabilità, scalabilità e manutenibilità. Successivamente, un'analisi dei dati raccolti e forniti a molteplici tipi di modelli di machine learning mostra il potenziale per la cura predittiva delle piante e il rilevamento delle anomalie, basandosi su uno specifico sensore chiamato Bioristor, utilizzato come "ground truth" per convalidare la correttezza dei modelli ottenuti. La tesi si conclude con alcune riflessioni sulle lezioni apprese durante lo sviluppo e suggerimenti per futuri miglioramenti.

Parole chiave: IoT, orchestratore, nodi, hub-and-spoke, automatizzato, AI, machine learning, modelli, Agricola 2000, Bioristor.

Contents

1 | Introduction: AI-Driven Orchestration for Scalable Plant Care

Digital agriculture increasingly relies on intelligent decision-making systems that coordinate heterogeneous sensors, interpret environmental data, and automate responses across distributed networks. The Smart Plants initiative addresses the possible gap between single sensors and centralized systems, by building an AI-aware orchestration platform that combines dedicated sensor hardware, a Raspberry Pi intelligence hub, machine learning models for stress prediction and irrigation optimization, and web-based control dashboards. This thesis documents the latest evolution of the system, focusing on the migration from semi-autonomous sensor nodes to a centrally orchestrated, AI-enhanced architecture capable of scaling from hobbyist prototypes to institutional deployments while improving decision quality through data fusion and predictive analytics. The agriculture sector faces a dual mandate, increasing the productivity while reducing resource consumption and operational complexity, and the needs for more heterogeneous and rapidly scalable solutions. Climate variability intensifies the need for continuous monitoring of soil moisture, microclimate, and light levels, while labor shortages push teams to automate repetitive tasks such as irrigation and alerting. High-end professional solutions exist, but they are costly and often closed, makers and small producers therefore seek open, configurable alternatives that leverage microcontrollers: Smart Plants positions itself in this landscape by providing a modular, open-source stack that bridges research prototypes with deployable systems.

1.1. Premise

The work presented in this thesis represents an independent contribution spanning the full lifecycle of a precision agriculture platform: from initial concept and system architecture, through firmware and software development, to machine learning model design, deployment, and empirical validation.

Two external partners contributed field data used to train and evaluate the machine learning models. Agricola 2000, an industrial agricultural operator, provided access to their greenhouse infrastructure, allowing the author to deploy sensor nodes on-site and collect multi-week telemetry under production conditions; all hardware installation, data logging, and system maintenance during this deployment were carried out by the author, with agronomic guidance and on-site access provided by said partner. The IMEM CNR laboratory, located on the campus of the University of Parma, supplied Bioristor electro physiological measurements, which served as high-fidelity ground-truth labels for plant stress status and were used exclusively for model supervision and validation. The Bioristor instrument itself was operated by IMEM personnel under their own experimental protocols and has proven to be an exceptional guiding tool given its precision in predicting health levels of the plant undergoing the experiment. Smart Plants' contribution in that context was the design of the co-located sensor apparatus, the data integration pipeline, and the machine learning methodology that leverages Bioristor labels to train commodity sensor models.

1.2. Idea

The core idea behind Smart Plants is to deliver a ready-to-use, easy-to-deploy platform that unifies sensing, actuation, analytics, and user interaction across multiple distributed nodes. Soil, ambient, and light sensors feed a central middleware running on a Raspberry Pi, which acts as the orchestration hub. The middleware aggregates multi-sensor data, executes machine learning models for predictive analytics (forecasting, classification, stress detection), and makes coordinated actuation decisions through connected electro-valves and pumps. Centralized orchestration enables intelligent cross-device coordination, consistent policies, and continuous model improvement based on collected telemetry. Users interact through a Flask-based web dashboard offering live telemetry, historical analysis, experiment management, AI-assisted alerts, and model configuration. Unlike earlier iterations that executed watering logic directly on each ESP32, the new design concentrates orchestration, persistence, and intelligence on the Pi, enabling AI-enhanced workflows, model retraining,

and reproducible agronomic research.

1.3. Problem Statement

Previous Smart Plants deployments [1] and competitor solutions expose critical limitations when applied to precision horticulture. (1) *Distributed intelligence barrier*: Each ESP32 operates autonomously with limited processing power, preventing deployment of sophisticated AI models for stress detection, yield prediction, or adaptive irrigation. (2) *Data fragmentation*: Monitoring telemetry lives in multiple silos (ThingSpeak, Firebase, local logs), preventing the data fusion and longitudinal analysis required for effective machine learning. (3) *Configuration drift*: Decentralized node management causes inconsistent policies across devices, complicating updates and reproducible research. (4) *Scalability ceiling*: Ad-hoc wiring of individual controllers does not support seamless addition of new sensors or reallocation of resources across teams. (5) *Opacity to operators*: Autonomous nodes offer limited insight into decision rationale, making it difficult to debug failures or improve watering strategies. The revised system tackles the problem of *intelligent centralized orchestration*: how to consolidate sensor data, deploy AI models for evidence-based decisions, scale from single installations to multi-site operations, and provide operators with transparent, accountable control while preserving low-cost commodity hardware.

1.4. Objectives

Six objectives drive this thesis, centered on AI-driven orchestration and Bioristor-validated scalability.

- Design a hub-and-spoke orchestration architecture in which the Raspberry Pi acts as the intelligence and coordination hub, while ESP32 nodes concentrate on reliable signal acquisition, basic safety checks and command execution.
- Implement a middleware layer that ingests multi-protocol data (serial, MQTT, Wi-Fi), normalizes it, persists it in a unified repository, and exposes it through a secure, role-based web interface.
- Architect the system for horizontal scalability, enabling seamless addition of devices and multi-user/multi-experiment workflows.
- Integrate machine learning pipelines that perform real-time classification (plant stress), forecasting (soil moisture trends), and multimodal fusion (combining camera and sensor data) to drive intelligent, automated decisions.

- Validate AI models against Bioristor ground-truth measurements, demonstrating that commodity sensors trained with Bioristor labels can support in-vivo stress detection.

Finally, validate the complete platform through industrial and academic case studies (Agricola 2000, IMEM), measuring improvements in watering precision, early stress detection, operator workload, and decision transparency compared with rule-based baselines. The study focuses on the software-defined components of Smart Plants as middleware, dashboard, analytics, and orchestration and the integration patterns that bind them to the ESP32 Plant Sensors. Sensor hardware design, 3D enclosures, and large-scale agronomic trials are referenced but not exhaustively explored. Similarly, the research does not attempt to optimize crop-specific irrigation strategies, instead it demonstrates how the platform can host such strategies as configurable modules.

1.5. Scalability and Product Vision

Smart Plants is conceived as a product family rather than a single-purpose prototype. The hub-and-spoke architecture allows a single enthusiast to monitor one balcony planter while also scaling to institutional deployments that span laboratories, greenhouses, and office atria. Configuration profiles and role-based access control let multiple teams share the same infrastructure without interfering with one another, ensuring that companies can roll out dozens of sensors across sites, while schools or universities may allocate per-course sandboxes on the same Raspberry Pi cluster. The dynamic middleware detects new devices as they appear on USB or MQTT, assigns them to logical rooms, and applies default policies; operators can then refine device-specific behavior through the dashboard. This flexibility underpins Smart Plants' ambition to bridge the gap between DIY horticulture tools and enterprise-ready facility management suites.

1.6. Research Questions and Contributions

The thesis pursues three complementary research questions centered on AI-driven orchestration and scalable plant management:

RQ1 **How can a centralized orchestration architecture intelligently coordinate heterogeneous plant sensors and actuators, deploy real-time AI models for decision-making, and scale seamlessly from hobbyist to institutional deployments without sacrificing configurability or transparency?**

RQ2 **Which machine learning techniques (classical, deep learning, and multi-**

modal fusion) most effectively leverage distributed sensor data and visual information to predict plant stress, forecast soil moisture, and optimize watering schedules in uncontrolled greenhouse environments?

RQ3 What data governance, annotation, and experimentation workflows enable continuous improvement of deployed AI models through feedback loops, while maintaining reproducibility and enabling collaborative research across multiple sites?

To answer these questions, the thesis contributes: (i) a reference implementation of a centralized orchestration architecture with dynamic device discovery, multi-protocol support (serial, MQTT, Wi-Fi), and role-based access control; (ii) an integrated AI/ML toolbox comprising real-time forecasting, stress classification, and multimodal fusion models trained and deployed on commodity hardware; (iii) a reproducible data governance and experimentation protocol validated across industrial (Agricola 2000) and academic (IMEM) sites; (iv) operational playbooks covering deployment, security, and maintenance for both hobbyist and institutional settings; and (v) empirical evidence that centralized, AI-enhanced orchestration improves watering precision, enables early stress detection, reduces operator workload, and sustains scalable multi-site operations compared with autonomous or rule-based baselines.

1.7. Methodology

The research follows a design science methodology with iterative prototyping, operators validation, and empirical evaluation. Each iteration comprised four steps: (1) requirements elicitation with growers, agronomists, and makers; (2) architectural redesign and implementation across firmware, middleware, and interface layers; (3) deployment in controlled and semi-controlled environments to gather telemetry and qualitative feedback; and (4) reflection, resulting in backlog updates and revised design principles. Agile-inspired sprints orchestrated firmware and middleware work in parallel, while research sprints focused on data collection, model training, and statistical analysis. This dual-track approach allowed the team to balance engineering deliverables with scientific rigor.

Data for evaluation originated from two principal sources. First, laboratory simulations recreated drought and overwatering scenarios using controlled irrigation schedules and environmental chambers. Second, field pilots such as the Agricola 2000 collaboration provided real-world variability, including fluctuating lighting, manual interventions, and hardware faults. Quantitative metrics (e.g., watering deviation, alert latency, model F1 scores) were complemented by structured interviews that captured operator satisfaction,

perceived workload changes, and adoption barriers.

1.8. Contributions Summary

The thesis advances precision agriculture through seven interconnected contributions spanning system architecture, machine learning, and operational practice:

- **Orchestration architecture.** A publicly documented hub-and-spoke gateway with modular services, unified data persistence, and role-based access control, serving as a reusable reference for IoT plant monitoring platforms.
- **Bioristor-validated AI framework.** A methodology demonstrating that AI models trained on commodity environmental sensors, using Bioristor-derived ground-truth labels, can achieve accurate stress detection in production deployments without requiring Bioristor, reducing per-plant hardware cost by an order of magnitude while preserving physiological accuracy through one-time calibration in controlled experiments.
- **AI integration patterns.** Methodologies for embedding machine learning models (forecasting, classification, multimodal) into resource-constrained environments, with techniques for continuous retraining and model versioning.
- **Firmware and protocol library.** A collection of ESP32 patterns for reliable sensing, actuator control, configuration management, and multi-transport connectivity (serial, MQTT, Wi-Fi), enabling reuse in derivative projects.
- **Data governance and experimentation workflow.** Reproducible notebooks, annotated datasets, and evaluation protocols that formalize the journey from raw telemetry to deployable models, supporting collaborative research across sites.
- **Operational and deployment guides.** Comprehensive repository covering installation, security hardening, troubleshooting, and maintenance for both enthusiast and institutional deployments, bridging the gap between research prototypes and production systems.
- **Validated evidence.** Empirical results from Agricola 2000 and IMEM Bioristor experiments demonstrating that centralized orchestration with AI support improves watering precision, enables early stress detection, reduces operator overhead, and scales robustly across mixed environments.

1.9. Limitations and Assumptions

While comprehensive, the study makes several simplifying assumptions. Power supply design for large farms, redundancy across multiple Raspberry Pi gateways, and long-range wireless protocols such as LoRaWAN remain out of scope. The evaluation focuses on leafy vegetables and tomatoes, treating other crops analogously. Additionally, cost modeling is limited to commodity hardware, omitting labor and facility retrofitting expenses. These constraints are acknowledged so that future work can prioritize scaling beyond the greenhouse-scale deployments described herein.

1.10. Thesis Structure

The lifecycle of this work, the study, design, system implementation, data harvesting, and processing, represents an original contribution developed by the author. While Chapter 2 reviews enabling technologies and surveys related solutions, the core technical contributions begin in Chapter 3, which details the original system design, data flows, and implementation specifics across firmware, middleware, and interfaces. Chapter 4 introduces the novel application of AI models and the specific architectural decisions tailored to the case study. The practical execution and management of the deployments are presented in Chapters 5 and 6, while Chapter 7 consolidates the primary results derived from these experiments. Finally, Chapter 8 provides an independent synthesis of the findings and outlines future research directions.

2 | Technologies for Different Types of Agriculture

Smart Plants leverages the convergence of embedded sensing, wireless communications, and intelligent dashboards. This chapter outlines the key technologies that shape the platform and compares them with alternative solutions from the precision agriculture domain.

Agricultural IoT solutions combine resource-constrained edge devices with more capable gateways that perform data fusion and orchestration. The literature highlights the benefits of hybrid architectures in which low-power sensor nodes capture environmental metrics while gateways apply business rules and interface with users [2]. Smart Plants adopts this paradigm: each node focuses on capturing soil moisture and ambient conditions, whereas Raspberry Pi aggregates readings, executes watering logic, and records provenance for audits.

Figure 2.1: ***Smart Plant sensors in the setup with a central node and pumps***

2.1. Sensor Technologies for Plant Monitoring

The Plant Sensor node (figure 2.1) integrates a capacitive soil moisture probe, a DHT22 sensor for temperature and relative humidity [3], and an Adafruit TSL2561 photometric sensor that measures lux values across a wide dynamic range [4], and soil electrical conductivity thresholds can be configured per plant type, enabling the middleware to interpret raw ADC readings through calibration tables. These sensors balance affordability with adequate precision for greenhouse environments and contained costs, since they are designed for quick swapping, using connectors and labeled headers to reduce service time. Additional instrumentation, including Bioristor sap sensors [5] and edge cameras, extends the dataset used for AI models, enabling multimodal analysis to correlate soil status with plant physiology.

2.1.1. Data Transmission Protocols

Smart Plants supports multiple transport layers to cater to constrained field deployments. In controlled environments that do not allow the use of Wi-Fi as a means of communication, nodes connect via USB serial to Raspberry Pi for reliable, low-latency data transfer. The firmware can also publish over Message Queuing Telemetry Transport (MQTT), a lightweight publish/subscribe protocol that scales to dozens of devices while preserving battery life [6]. MQTT topics mirror the JSON schema expected by the middleware,

ensuring consistent parsing regardless of transport. The choice between serial and MQTT is runtime configurable so that growers can transition from tethered pilots to wireless installations without re-flashing firmware.

2.1.2. Microcontrollers and Microprocessor Platforms

The ESP32 microcontroller offers an attractive blend of dual-core processing, integrated Wi-Fi/BLE radios, and low sleep currents [7]. These features allow Plant Sensor to handle local filtering and fail-safe logic even when connectivity is intermittent. For the central controller, Raspberry Pi 3B+ provides enough compute to run Flask, SQLite, MQTT brokers, and minimum inference tasks simultaneously [8]. Alternative platforms such as STM32 MCUs or industrial PLCs were considered, but the ESP32–Raspberry Pi pairing maximizes community support, third-party libraries, and availability.

2.1.3. AI Frameworks and Data Management

State-of-the-art agricultural analytics increasingly rely on deep learning for time-series forecasting and computer vision [9, 10]. Smart Plants leverages `TensorFlow` and `Keras` for neural models [11], `scikit-learn` for classical baselines [12], and `PyTorch` when GPU acceleration is required during offline experimentation [13]. Model prototyping occurs in Colab notebooks (e.g., `smartplant_colab_training.ipynb`) that interface with shared datasets stored on the Raspberry Pi. Versioned checkpoints can be exported as TensorFlow SavedModels or ONNX artifacts for deployment within the middleware.
Since reliable AI requires disciplined data management, sensor readings are persisted in SQLite for rapid query, with nightly exports to Parquet files for large-scale analysis. Annotated events, such as stress episodes or maintenance interventions, are stored in experiment metadata JSON files. The system supports both supervised learning (e.g., stress classification from sensor fusion) and semi-supervised approaches that cluster unlabeled data to suggest new alert thresholds.

2.1.4. Smart Mirror Technology and Application

An earlier Smart Plants iteration introduced Smart Plants Touch, a smart-mirror style view that surfaces live telemetry and alerts in communal spaces [14]. In the updated design the view acts as a thin client that consumes a WebSocket feed from the Raspberry Pi, enabling AI-generated advisories and keeping displays in sync with the central database. The full integration design is discussed in Chapter 3.

2.1.5. Edge-to-Cloud Security Considerations

Deployments that bridge local networks and remote operators must guard against credential leakage and unauthorized control. Smart Plants incorporates multiple safeguards referenced throughout the backlogs: role-based access control constrains dashboard operations, MQTT credentials and Firebase Admin credentials are set up by environment files and taken from web API of the services, TLS is recommended for remote brokers. Future iterations target HTTPS termination on the Raspberry Pi, two-factor authentication, and VPN access control lists. Compared with purely cloud-hosted solutions, the local-first design reduces dependence on third-party services while still allowing optional replication to Firebase for disaster recovery.

Fog computing literature emphasizes lightweight schedulers, containerization, and local analytics at the network edge. Smart Plants mirrors these recommendations by combining APScheduler for persistent jobs, having also the option of using Docker for reproducible deployments, thus opening a possible integration of microservices containers. The gateway exposes a plugin interface so that even domain-specific modules, like nutrient dosing or HVAC control, can subscribe to the same event bus. This composition aligns the project with modern edge orchestration patterns while keeping entry barriers low for contributors familiar with Python/Docker [15] ecosystems.

2.2. Comparison with Related Platforms

Table 2.1 compares Smart Plants with comparative commercial and open-source alternatives. Commercial suites typically bundle proprietary sensors and cloud dashboards, offering strong support but limited extensibility. Open-source projects provide modifiable firmware and dashboards but often lack cohesive user management or AI tooling. Smart Plants fills this niche by delivering a unified experience that combines open hardware, modular middleware, and analytics-ready datasets.

Platform	Hardware Openness	Central Orchestration	AI Support
Netafim (commercial)	Proprietary	Cloud-hosted	Limited (rule-based)
OpenAg (open-source)	Open	Partial (local scripts)	Experimental
FarmBot (open-source)	Open	Local controller	Minimal
Smart Plants (this work)	Open	Mixed	Integrated models

Table 2.1: *Comparison between Smart Plants and representative smart agriculture platforms.*

Synthesizing insights from this chapter yields concrete requirements for the implementation chapters: support for heterogeneous sensors, dual transport (serial and MQTT), local-first data storage with export pipelines, extensible dashboards, secure remote access, and edge-friendly AI deployment. These requirements guided backlog prioritization and informed the architectural decisions elaborated in Chapter 3.

2.3. Previous Works and Existing Solutions

Commercial systems, such as Netafim's drip irrigation controllers or Grodan's climate monitors, provide turnkey solutions but often require proprietary hardware and service contracts. Research prototypes explored data aggregation via cloud services like ThingSpeak [16], yet many lack strong user management or automation capabilities. Smart Plants differentiates itself by combining an open-source stack, modular firmware, and a full-featured web dashboard with alerting, experiment tracking, and the possibility to implement AI-assisted decision support.

Key takeaways for the subsequent design chapter are the desirability of a hub-and-spoke topology, the practicality of ESP32-class sensors, and the importance of supporting both wired and wireless transport. These insights directly inform the architecture described in Chapter 3.

2.4. Legacy Mobile-Centric Architecture

Before the Raspberry Pi gateway became the command center, Smart Plants relied on a mobile-first stack developed during the 2024–2025 academic year.[1] Each Plant Sensor published measurements directly to Firebase Realtime Database via MQTT, while Smart Plants Touch (Flutter) handled device onboarding and historical charts. Figure 2.2 shows this topology; the full transition narrative — SAP onboarding, NVS preference caching, MQTT topic schema, and backward-compatibility design decisions — is covered in Chapter 3.

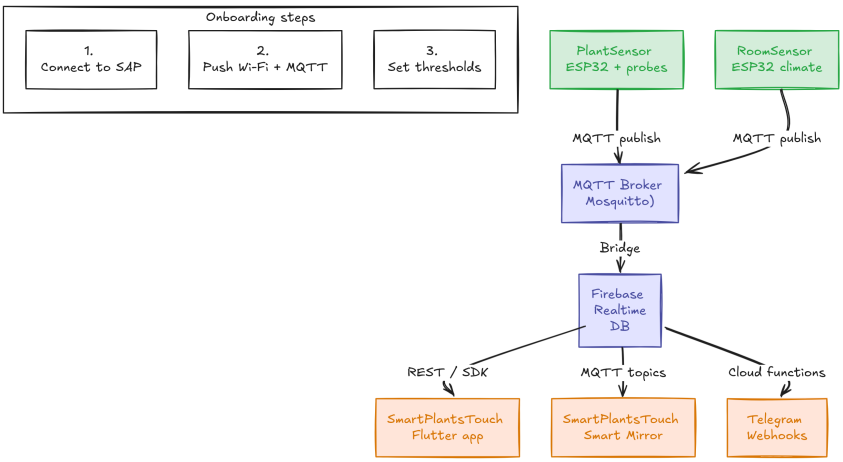


Figure 2.2: *Legacy Smart Plants architecture[1]: ESP32 nodes published directly to Firebase via MQTT bridges, without an intermediate gateway.*

2.5. End-to-End Sequence Flow

Figure 2.3 gives a high-level overview of the Smart Plants control loop: firmware samples sensors and publishes over serial or MQTT; the gateway persists the payload, evaluates automation rules, and broadcasts updates to connected clients. The fully annotated sequence — including retry policies and WebSocket delivery — is detailed in Chapter 3.

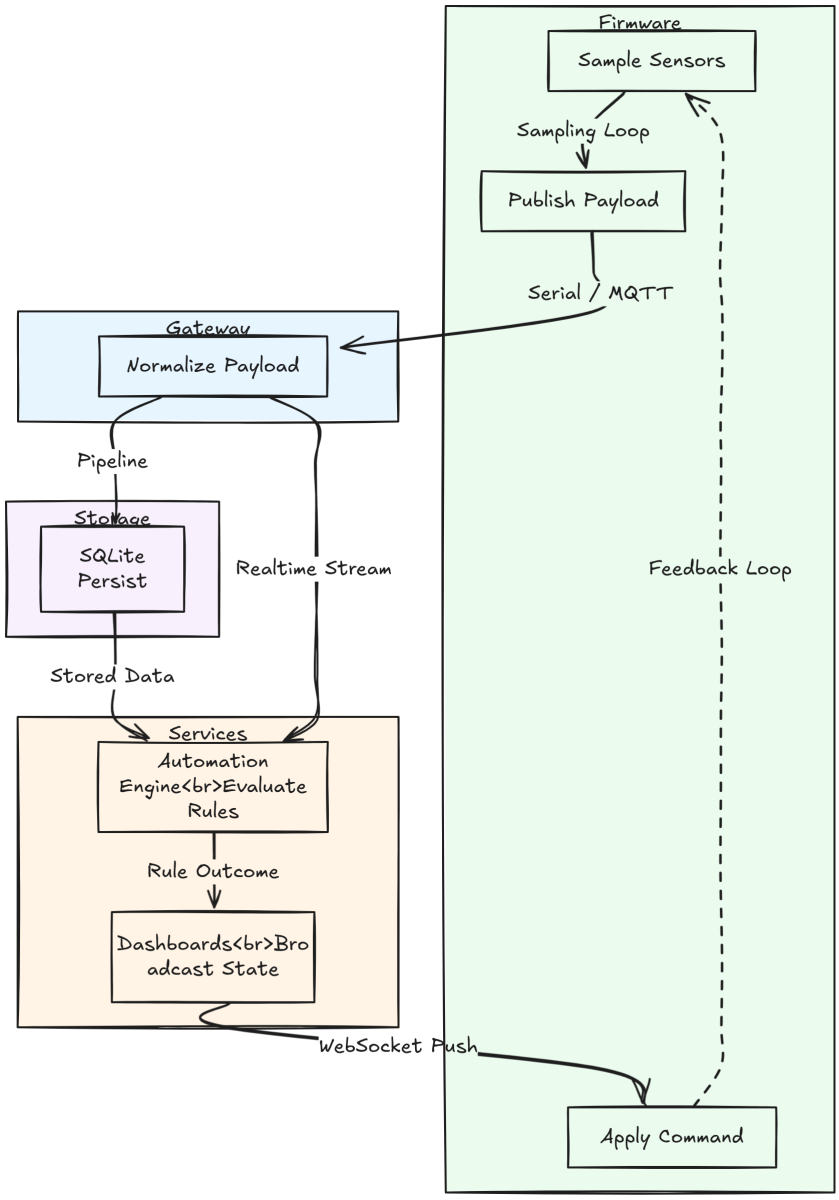


Figure 2.3: *High-level sequence flow from sensor sampling to actuation and user notification.*

3 | System Architecture

The Smart Plants platform implements a layered, hub-and-spoke architecture integrating distributed ESP32 sensing hardware with a central Raspberry Pi gateway. Each layer exposes well-defined interfaces, allowing firmware, middleware, and user-facing components to evolve independently.

3.1. System Overview

The system follows a hub-and-spoke topology optimized for AI workloads. ESP32-based Plant Sensor nodes capture environmental data and interact with actuators such as electrovalves or pumps. These nodes connect either via USB serial or Wi-Fi to a Raspberry Pi gateway. The gateway runs a Flask web application (`app.py`), an MQTT worker, AI inference services, and background schedulers. Data flows from sensors into the gateway, where it is normalized, persisted in SQLite (`sensor_history.db`), fed into ML pipelines, and exposed through REST endpoints and WebSocket channels to dashboard (figure 3.1), smart mirror and mobile app, and automation services. The repository structure of the new system and the configuration files are listed in B.

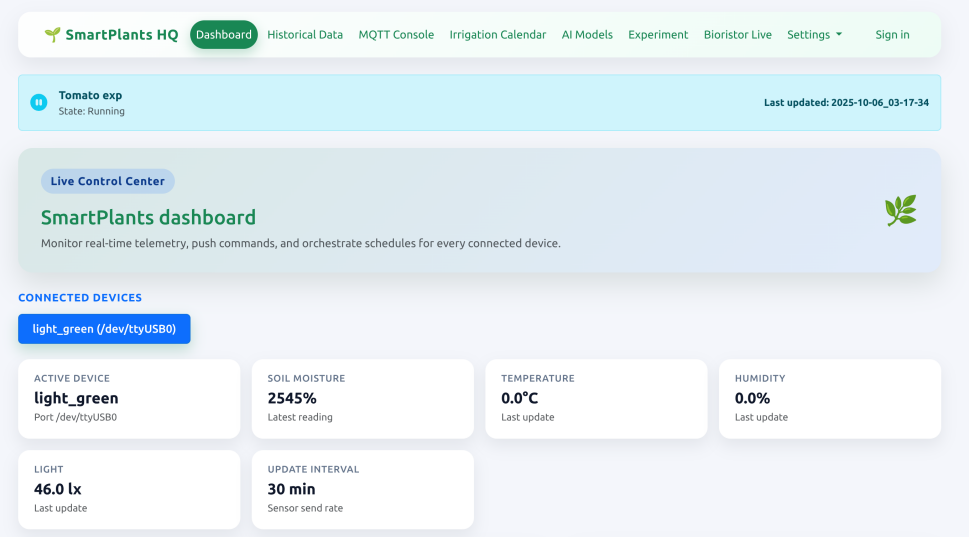


Figure 3.1: *A view of the dashboard interface used by the operator*

3.2. Legacy mobile-centric architecture

Before the Raspberry Pi gateway became the command center, Smart Plants relied on the mobile-first stack developed during the 2024–2025 academic year.[1] In that release, every Plant Sensor connected to Wi-Fi via MQTT and published measurements directly to Firebase Realtime Database. Smart Plants Touch, the Flutter application of previous deployment, acted as the orchestration surface: it handled device onboarding, stored per-plant preferences, and rendered historical charts sourced from Firebase. A companion smart-mirror running MagicMirror[2] subscribed to the same topics to show ambient widgets, while Telegram webhooks delivered alerts without an intermediate gateway.

Configuration on the ESP32 devices mirrored this architecture. When powered on, each board exposed a Service Access Point used to capture Wi-Fi credentials, MQTT endpoints, and sampling cadences from the mobile app. Preferences were cached in non-volatile storage so the nodes could reconnect autonomously. Firmware also streamed raw ADC readings for soil moisture, DHT22 temperature/humidity data, TSL2561 light values, and solenoid valve state transitions. Figure 2.2 summarizes the legacy topology that informed the requirements for the new centralized design. Preserving this heritage was essential when redesigning the platform. The new Raspberry Pi hub continues to honor the plug-and-play workflow (SAP onboarding, MQTT publishing, mobile analytics) while centralizing orchestration, security, and long-term data storage.

3.3. Smart Plants Touch

Smart Plants Touch acts as an ambient display deployed in workspaces or living rooms. In the new architecture, it consumes a WebSocket feed served by the Raspberry Pi instead of connecting to each sensor directly. This shift allows the visualizers to display aggregated KPIs (soil moisture trend, watering status, alert banners) that align with the main dashboard. Reducing device intelligence simplifies updates: any change to data presentation or thresholds occurs in the central middleware, instantly reflecting across all clients. This especially occurred during runtime with Agricola 2000, where the need for a quick overview highlighting plants requiring attention and the status of automation was of high interest, while on IMEM experiment was necessary due to real distance constraint: thanks to funneling via Tailscale, a remote access on the platform on Raspberry was possible.

Smart Plants Touch runs as a web view backed by lightweight JavaScript widgets. It subscribes to the `device_ws` WebSocket group managed by Flask-Sock. Messages contain

the same JSON payload stored in SQLite, allowing the view to reuse data transformation logic shared with the main dashboard.

3.4. Device Configuration and Data Flow

Device configuration originates from the Raspberry Pi. Users update thresholds, watering schedules, and delays via the dashboard, which serializes changes as JSON and pushes them to the selected device over serial or MQTT. The firmware stores received preferences in non-volatile storage, but the Pi retains the canonical state to guarantee consistency across power cycles. Data flows back to the Pi at configurable intervals (default 60 s for environmental data, 240 s for soil moisture), where the middleware timestamps and logs each reading. The Flask application offers REST endpoints (`/data`, `/api/sensor_history`) and WebSocket streams for live monitoring. Historical queries aggregate readings per device, and alert rules inspect incoming messages to trigger email or Telegram notifications. The middleware also forwards selected events to Firebase or external MQTT brokers when remote access is required.

3.4.1. Security and Reliability Considerations

Role-based access control governs dashboard usage, with session-based authentication enforcing permissions for admins, editors, and viewers. Commands are logged with timestamps and user IDs to aid auditing. The gateway monitors serial ports and automatically attempts reconnection when an ESP32 resets. For MQTT communication, TLS is supported through configurable credentials, aligning with cloud broker recommendations (HiveMQ offers also an API endpoint to do so, with associated key to make the link stable and secure).

3.5. Hardware Components

Plant Sensor combines a capacitive soil moisture probe connected to the ADC, a DHT22 sensor on GPIO 26, a TSL2561 I^2C light sensor, and a solenoid valve driven through a MOSFET controlled by GPIO 27. A Room Sensor configuration omits the valve and focuses on environmental metrics, but is rarely used as standalone device. Both devices house an ESP32-WROOM module for Wi-Fi connectivity that can be bypassed by using the serial communication with the central unit, as this hosts USB hubs for multiple asynchronous devices, a stable 5 V power supply, and optional pins for additional gadgets such as speakers and fans.

3.6. Dynamic Device Topology

Scalability is achieved through a discovery layer that abstracts sensors and actuators into logical roles. The middleware maintains registries for irrigation controllers, ambient monitors, nutrient probes, and vision systems. New devices advertise their capabilities in JSON descriptors, enabling the Raspberry Pi to assign them to automation routines without manual code changes. This dynamic structure encourages heterogeneous configurations: a hobbyist can attach a single Plant Sensor, while industrial partners can mix dozens of Plant Sensor, camera gateways, and third-party devices such as Bioristor under the same orchestration plane, scaling the central unit as the requirements dictate. Load is distributed by delegating high-frequency acquisition to MQTT while reserving serial connections for latency-sensitive control, ensuring the platform scales gracefully as deployments grow.

3.7. Data Collection and Transmission

The ESP32 firmware (`plant_sensor_wifi_extended.ino`) samples sensors, applies light filtering, and publishes JSON payloads containing soil moisture, temperature, humidity, light, watering status, and plant metadata. Analog readings are left in raw units (0–4095) so that the Raspberry Pi can convert them into calibrated percentages based on per-device thresholds stored in a configuration database. The Pi's `serial_reader_thread` listens for JSON lines, validates them, and persists them through `save_sensor_history`. A separate MQTT worker consumes cloud-originated commands and mirrors them into the same processing pipeline, ensuring consistent behavior regardless of transport. For completeness, an excerpt of the raw telemetry stream (as recorded during the deployment) is reported in Appendix B.

3.7.1. ESP32 Sensor Data Format

ESP32 firmware transmits sensor telemetry as newline-delimited JSON objects over serial (USB) or MQTT. The standardized payload format ensures consistent parsing regardless of transport mechanism. Listing 3.1 shows a typical sensor data message:

```
{
  "device_id": "70000000",
  "timestamp": 1633536000,
  "soil_moisture": 1850,
  "temperature": 24.5,
  "humidity": 62,
  "light": 15000,
```

```
  "soil_ec": 1800
}
```

Listing 3.1: *ESP32 sensor data JSON payload.*

Field semantics and units:

- **device_id**: Unique 8-character hexadecimal identifier burned into ESP32 firmware during flashing (e.g., `70000000`, `dark_green`)
- **timestamp**: Unix epoch timestamp (seconds since 1970-01-01) when measurement was captured
- **soil_moisture**: Raw 12-bit ADC reading (0–4095) from capacitive soil moisture sensor; converted to percentage by middleware based on per-device calibration
- **temperature**: Ambient temperature in degrees Celsius from DHT22 sensor (range: -40 to +80°C, accuracy: ±0.5°C)
- **humidity**: Relative humidity percentage from DHT22 sensor (range: 0–100%, accuracy: ±2%)
- **light_lux**: Illuminance in lux units from TSL2561 digital light sensor (range: 0.1 to 40,000 lux)
- **soil_ec**: Electrical conductivity reading from soil probe in arbitrary units, indicating nutrient concentration and salinity

The firmware maintains these values in raw form to preserve measurement precision and allow the central gateway to apply calibration curves specific to each deployment environment (soil type, probe positioning, elevation).

3.7.2. Configuration Command Format

Configuration updates flow from the Raspberry Pi to ESP32 devices bidirectionally over serial or MQTT. The configuration schema supports comprehensive device customisation without firmware reflashing. Listing 3.2 demonstrates a complete configuration payload:

```
{
  "id": "70000000",
  "plant": "cherry tomato pepe f1",
  "delay_readings": 60000,
  "delay_moist_read": 120000,
  "watering_time_cost": 3000000,
  "room_number": 1,
```

```
  "plant_data": {
    "max_light_lux": 25000,
    "min_light_lux": 1000,
    "max_temp": 28,
    "min_temp": 18,
    "max_env_humid": 70,
    "min_env_humid": 50,
    "max_soil_ec": 2500,
    "min_soil_ec": 1200
  }
}
```

Listing 3.2: *ESP32 configuration command JSON structure.*

Configuration parameter semantics:

- **id**: Target device identifier matching the `device_id` in telemetry messages
- **plant**: Human-readable plant variety descriptor for dashboard display and experiment annotation
- **delay_readings**: Interval between environmental sensor readings in milliseconds (default: 60,000ms = 1 minute)
- **delay_moist_read**: Interval between soil moisture measurements in milliseconds (default: 120,000ms = 2 minutes); typically longer than environmental readings to conserve probe lifespan
- **watering_time_cost**: Maximum watering duration in milliseconds before automatic shutoff safety trigger (default: 3,000,000ms = 50 minutes)
- **room_number**: Logical room or greenhouse bench identifier for multi-zone installations
- **plant_data**: Nested object containing plant-specific threshold ranges:
  - **max_light_lux** / **min_light_lux**: Acceptable light level range for healthy growth
  - **max_temp** / **min_temp**: Temperature boundaries in Celsius
  - **max_env_humid** / **min_env_humid**: Relative humidity bounds (percentage)
  - **max_soil_ec** / **min_soil_ec**: Electrical conductivity range indicating optimal nutrient concentration

Configuration persistence strategy: Upon receiving a configuration message, the ESP32 firmware stores parameters in non-volatile storage using the ESP-IDF `Preferences` library. This ensures settings survive power cycles. However, the Raspberry Pi gateway maintains the canonical configuration state and automatically re-applies settings when devices reconnect after reboots, preventing configuration drift in long-running deployments.

3.7.3. MQTT Command Protocol

Beyond configuration updates, the system supports real-time device control via MQTT. Commands are published to the `smart_plants_debug` topic and consumed by all subscribed ESP32 devices. Listing 3.3 enumerates supported commands:

```
# Restart ESP32 (soft reset)
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug -m "restart"

# Manually open irrigation valve
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug -m "open"

# Manually close irrigation valve
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug -m "close"

# Clear all stored preferences (factory reset)
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug -m "clear_preferences"

# Clear only data preferences (keep Wi-Fi credentials)
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug \
  -m "clear_data_preferences"

# Send complete configuration update (JSON payload)
mosquitto_pub -h <mqtt-broker> -t smart_plants_debug \
  -m '{"id":"70000000","plant":"basil","delay_readings":30000}'
```

Listing 3.3: *MQTT command examples using mosquitto_pub client.*

Command semantics and safety considerations:

- **restart**: Triggers software reset of ESP32, useful for recovering from firmware hangs or applying configuration changes that require reinitialisation
- **open** / **close**: Direct valve control commands bypass automation logic; primarily used for manual watering during maintenance or emergency irrigation
- **clear_preferences**: Factory reset that erases all stored configuration including Wi-Fi credentials; requires physical access to re-onboard device
- **clear_data_preferences**: Selective reset preserving network credentials while

clearing plant thresholds and sampling intervals

- **JSON configuration**: Complete or partial configuration updates; firmware merges received fields with existing preferences

Safety interlocks prevent simultaneous conflicting commands (e.g., `open` while automated watering active) and enforce maximum watering duration limits specified in `watering_time_cost` to prevent flooding from stuck valves or software bugs.

Figure 3.2 outlines the typical control loop executed by Smart Plants. Each iteration begins with a firmware sampling cycle, followed by message publication over serial or MQTT. The gateway ingests the payload, stores it, evaluates automation rules, and broadcasts relevant updates to dashboards and external services. While the figure abstracts away low-level retries, the production implementation includes exponential backoff for MQTT publishing and automatic reconnection for serial ports.

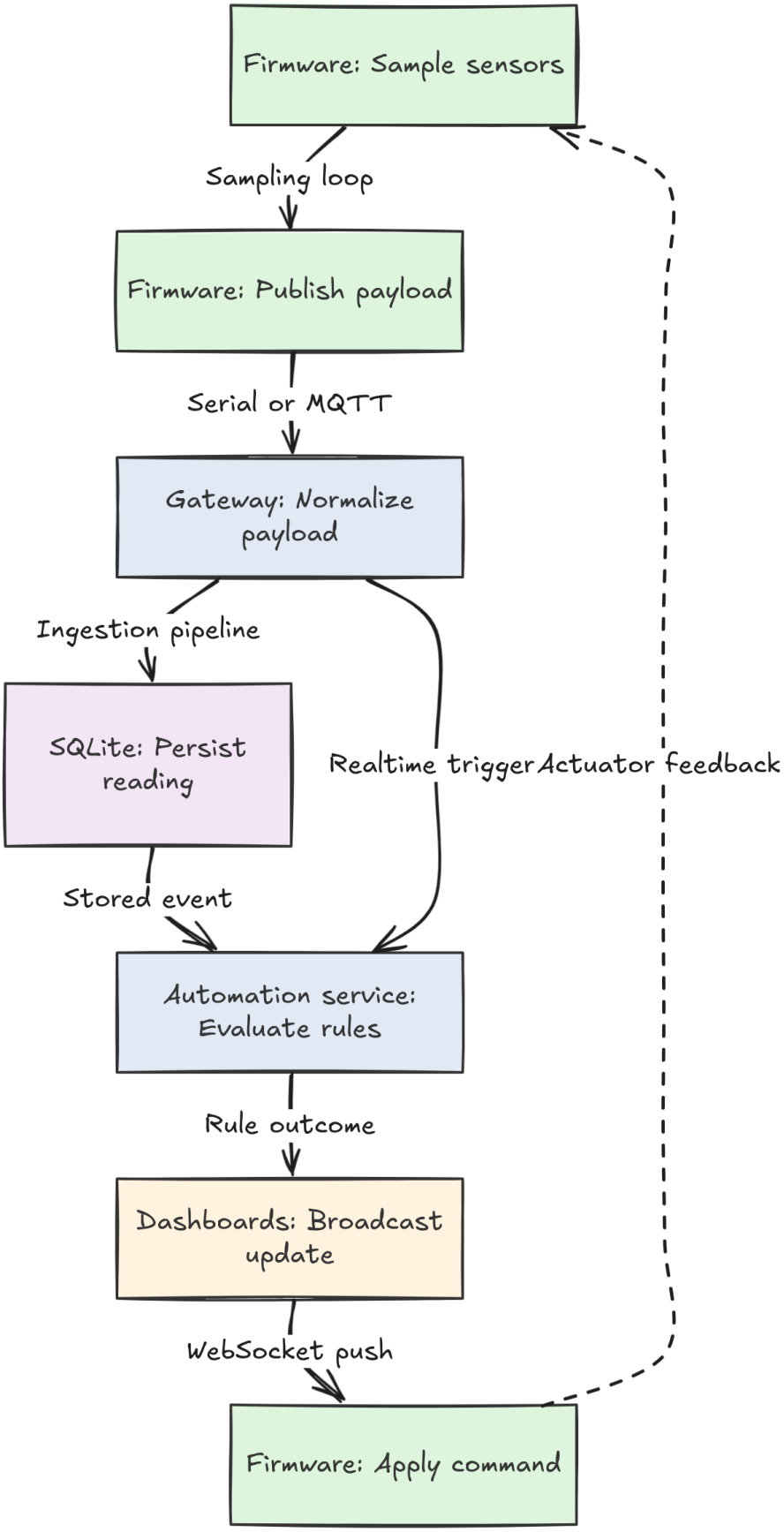


Figure 3.2: *Sequence flow from sensor sampling to actuation and user notification.*

3.8. Database Schema and Persistence Strategy

The SQLite schema recorded in `sensor_history.db` comprises tables for readings, device profiles, watering events, and experiment annotations. Readings capture timestamps, device identifiers, raw sensor values, calibrated metrics, and status flags describing automated actions taken. Foreign keys tie each reading to a logical plant or room, enabling downstream queries such as "show the last seven watering events for greenhouse bench three." Indexes on timestamp and device ID support rapid retrieval for dashboard charts, while periodic VACUUM operations maintain database health on SD cards.

3.8.1. Detailed Schema Design

The core database schema follows a normalized relational model optimized for time-series queries and analytical workloads. Table 3.1 enumerates the principal relations and their roles.

Indexes are carefully placed to support common query patterns. A composite index on `(device_id, timestamp)` accelerates time-range retrievals for dashboards, while a descending index on `timestamp` enables fast "latest reading" queries. Write performance remains acceptable under the typical 1–4 Hz per-device ingestion rate observed during trials, and batch inserts group multiple readings into single transactions to minimize lock contention.

Table	Purpose	Key Columns
`sensor_readings`	Time-series telemetry from all devices	`id`, `timestamp`, `device_id`, `soil_moisture`, `temperature`, `humidity`, `light_lux`
`devices`	Device registry with metadata and status	`device_id`, `name`, `type`, `room_id`, `last_seen`, `config_json`
`watering_events`	Irrigation log with durations and outcomes	`id`, `timestamp`, `device_id`, `duration_ms`, `trigger_source`, `result_moisture`
`experiments`	Experimental metadata and annotation	`experiment_id`, `name`, `start_date`, `end_date`, `settings_json`
`alert_logs`	Alert firing history with acknowledgments	`id`, `timestamp`, `rule_id`, `device_id`, `severity`, `acknowledged`

Table 3.1: *Primary tables in the Smart Plants database schema.*

3.8.2. Data Retention and Archival

To guard against data loss, the middleware emits optional change streams to CSV and Parquet files in `experiment_data/`. These exports feed machine-learning pipelines and serve as a backup should the primary database become corrupted. A configurable retention policy automatically archives readings older than a specified threshold (default 180 days) to compressed Parquet files, reducing active database size while preserving historical data for longitudinal studies. Future iterations target automated off-device backups to network-attached storage or cloud object stores, as informed by the TODO roadmap.

3.9. RESTful API and Endpoint Catalog

The Flask application exposes a comprehensive REST API that powers both the web dashboard and external integrations. All endpoints implement consistent error handling, rate limiting (planned), and role-based access control. Table 3.2 catalogs the principal routes grouped by functional domain.

Endpoint	Method	Purpose	Auth
`/`	GET	Dashboard home page	User
`/login`	GET/POST	User authentication	Public
`/logout`	GET	Session termination	User
`/data`	GET	Live sensor telemetry (JSON)	User
`/send_command`	POST	Dispatch command to device	Editor
`/api/sensor_history`	GET	Historical data query	User
`/api/devices`	GET	Device registry	User
`/api/alert_rules`	GET/POST/DELETE	Alert rule CRUD	Editor
`/firmware.bin`	GET	OTA firmware artifact	Device
`/upload_firmware`	POST	Upload new firmware binary	Admin
`/users`	GET/POST/DELETE	User management	Admin
`/experiments`	GET/POST/PUT	Experiment tracking	Editor
`/mqtt/publish`	POST	Manual MQTT message dispatch	Editor

Table 3.2: *Principal REST API endpoints exposed by the Smart Plants gateway.*

Responses follow a unified JSON envelope pattern that includes status codes, message fields, and optional data payloads. For example, a successful device command returns:

```
{
  "status": "success",
  "message": "Command sent to device 70000000",
  "timestamp": "2025-10-24T14:23:01Z"
}
```

Errors include descriptive messages and appropriate HTTP status codes (400 for client errors, 500 for server faults, 403 for unauthorized access).

3.9.1. WebSocket Real-Time Protocol

Live telemetry is delivered via WebSocket channels managed by Flask-Sock. Clients subscribe to the `/ws/device_data` endpoint and receive JSON messages whenever new readings arrive. The protocol supports selective subscriptions: clients specify device IDs of interest, reducing bandwidth for installations with dozens of nodes. Each message includes a sequence counter to detect packet loss, and the server implements exponential back off for reconnection attempts during transient network failures.

3.10. Scheduling and Automation Services

AP Scheduler orchestrates recurring jobs, including nightly data exports, alert rule audits, and OTA firmware availability checks. Schedules are defined in `general_settings.json`, and operators can toggle them via the dashboard. High-priority commands such as emergency watering bypass APScheduler and flow directly through Flask endpoints to minimize latency. Each job logs start and end times, surfaced in the dashboard for troubleshooting.

3.10.1. Automation Rule Engine

The rule engine evaluates incoming sensor data against user-defined thresholds in near-real time. Rules are expressed in a simple domain-specific language supporting boolean logic:

```
soil_moisture < 1200 AND temperature > 25
```

When a rule triggers, the engine spawns notification tasks (email, Telegram) and optionally executes remediation actions such as initiating irrigation. To prevent alert storms,

the engine implements debouncing: repeated violations within a configurable window (default 300 s) suppress subsequent notifications. Alert state transitions are logged to the `alert_logs` table, providing audit trails for compliance and troubleshooting.

3.11. Quality Assurance and Testing

Maintaining system reliability requires a mix of automated and manual tests. Unit tests (`test.py`) verify core utilities, while ad hoc scripts stress serial ingestion, MQTT throughput, and AI inference latency. Continuous integration is earmarked in the backlog, but interim safeguards include nightly regression tests triggered by cron on the Raspberry Pi and manual smoke tests—power cycling devices, forcing network outages, and validating that alert delivery remains functional. These practices ensured the Agricola 2000 deployment operated for weeks with minimal supervision.

3.12. Evolution of the Project

Three iterations led to the current architecture. The first version relied on ThingSpeak for data storage and left configuration scattered across devices. The second version shifted to Firebase but still required each ESP32 to maintain independent state, stressing device power budgets due to frequent polling. The third, described here, centralizes logic on the Raspberry Pi, uses persistent preferences only as a cache, and introduces a comprehensive dashboard with experiment tracking, AI models, and scheduling. This evolution improved maintainability, simplified updates, and enabled richer analytics unavailable when data was stacked across nodes.

3.13. Plugin Architecture and Extensibility

The Smart Plants platform implements a modular plugin system (Figure 3.3) that enables third-party extensions without modifying the core codebase. Plugins are registered in `plugin_settings.json` and loaded dynamically at runtime through Python's import mechanism.

3.13.1. Plugin Types and Lifecycle

- **Data source plugins**: Integrate external sensors or data streams (e.g., Bioristor probes, weather APIs) by implementing the `IDataSource` interface.
- **Notification plugins**: Add custom alert channels beyond email and Telegram by extending the `NotificationHandler` base class.
- **Dashboard widgets**: Contribute UI components that render in dedicated tabs or overlay existing pages.
- **Analytics plugins**: Register custom feature engineering pipelines or model inference endpoints for domain-specific AI tasks.

Plugin lifecycle management follows a standard pattern:

1. **Registration**: Plugins declare metadata (name, version, dependencies) in the settings file.
2. **Initialization**: The core application loads plugin modules during startup and invokes initialization hooks.
3. **Event subscription**: Plugins subscribe to internal event buses (sensor data arrival, configuration changes, alert triggers).

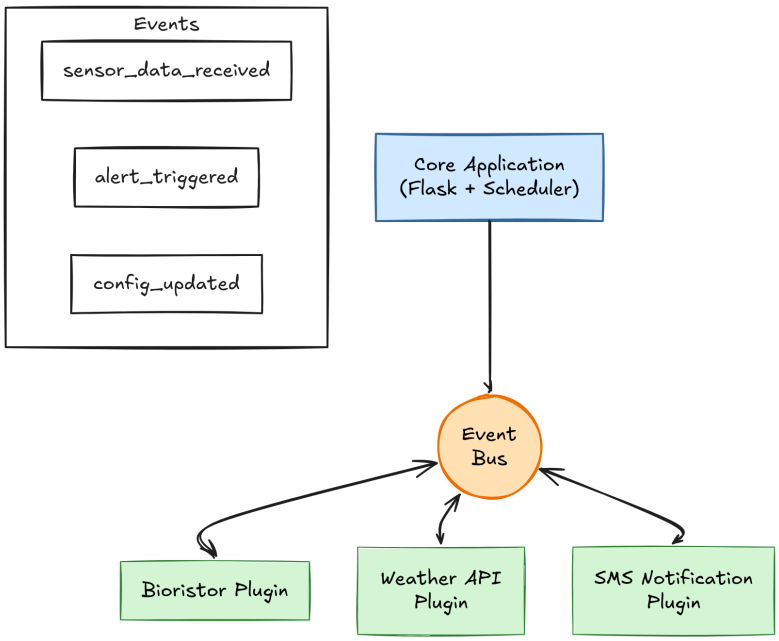


Figure 3.3: *Plugin architecture showing third-party extensions interacting with the core application via an event bus.*

4. **Cleanup**: When the system shuts down or plugins are disabled, cleanup hooks release resources.

3.13.2. Example Plugin Integration

Listing 3.4 demonstrates a minimal notification plugin that forwards alerts to Slack. The plugin implements the required `send_notification` method and registers itself during initialization.

```
# ============================================================
# Smart Plants Plugin: Slack Notification Handler
# Extends the notification system with Slack integration
# ============================================================
from typing import Dict, Any
import requests
from notifications import NotificationHandler

class SlackNotificationPlugin(NotificationHandler):
    """
    Slack notification plugin for Smart Plants alert system.
    Forwards alerts to Slack channels via webhook integration.

    Configuration (in plugin_settings.json):
    {
        "slack_notifier": {
            "enabled": true,
            "webhook_url": "https://hooks.slack.com/services/...",
            "channel": "#smart-plants-alerts",
            "username": "SmartPlants Bot"
        }
    }
    """
    def __init__(self, config: Dict[str, Any]):
        """
        Initialize plugin with configuration.

        Args:
            config: Plugin configuration dictionary
        """
        self.webhook_url = config.get('webhook_url')
        self.enabled = config.get('enabled', False)
        self.channel = config.get('channel', '#alerts')
        self.username = config.get('username', 'SmartPlants Bot')

        # Validate configuration
        if self.enabled and not self.webhook_url:
            raise ValueError("Slack webhook_url required when enabled")

```

```
    def send_notification(self, alert: Dict[str, Any]) -> bool:
        """
        Send alert notification to Slack channel.

        Args:
            alert: Alert data dictionary with:
                - message: Alert message text
                - device_id: Device identifier
                - severity: Alert severity (info/warning/critical)
                - timestamp: When alert was triggered

        Returns:
            True if notification sent successfully, False otherwise
        """
        # Skip if plugin disabled
        if not self.enabled:
            return False

        # -------------------- Format Slack Payload --------------------
        # Determine color based on severity
        color_map = {
            'info': '#36a64f',      # Green
            'warning': '#ff9900',   # Orange
            'critical': '#ff0000'   # Red
        }
        color = color_map.get(alert.get('severity', 'info'), '#808080')

        # Build rich message with attachments
        payload = {
            'channel': self.channel,
            'username': self.username,
            'text': f":warning: *Smart Plants Alert*",
            'attachments': [{
                'color': color,
                'title': alert.get('message', 'Alert Triggered'),
                'fields': [
                    {
                        'title': 'Device',
                        'value': alert['device_id'],
                        'short': True
                    },
                    {
                        'title': 'Severity',
                        'value': alert.get('severity', 'Unknown'),
                        'short': True
                    },
                    {
                        'title': 'Timestamp',
                        'value': alert.get('timestamp', 'N/A'),
                        'short': False
```

```
                    }
                ],
                'footer': 'Smart Plants Monitoring System',
                'footer_icon':
    ↪ 'https://platform.slack-edge.com/img/default_application_icon.png'
            }]
        }

        # -------------------- Send to Slack --------------------
        try:
            response = requests.post(
                self.webhook_url,
                json=payload,
                timeout=10
            )

            if response.status_code == 200:
                print(f"[SLACK] Alert sent to {self.channel}")
                return True
            else:
                print(f"[SLACK ERROR] HTTP {response.status_code}:
    ↪ {response.text}")
                return False

        except requests.exceptions.RequestException as e:
            print(f"[SLACK ERROR] Failed to send notification: {e}")
            return False

# -------------------- Plugin Registration --------------------
# This is called automatically by the plugin loader
def register_plugin(app, config):
    """
    Plugin entry point called during application initialization.

    Args:
        app: Flask application instance
        config: Plugin configuration from plugin_settings.json
    """
    plugin = SlackNotificationPlugin(config)
    app.notification_handlers.append(plugin)
    print("[PLUGIN] Slack notification plugin registered")
    return plugin
```

Listing 3.4: *Example Slack notification plugin structure.*

This extensibility pattern ensures that the Smart Plants ecosystem can evolve through community contributions while maintaining stability in the core platform. The roadmap envisions a curated marketplace where validated plugins can be discovered, reviewed, and installed through the dashboard.

4 | AI-Driven Plant Stress Detection: Design and Models

Building on the hardware and software architecture described in Chapter 3, this chapter presents the **core AI contribution** of the thesis. The Smart Plants platform uses environmental sensor data, validated against the *Bioristor* ground-truth sensor, to train and deploy machine learning models capable of classifying and forecasting plant water stress. Two experimental datasets are used: the **IMEM Bioristor experiment** and the **Agricola 2000 field trial**. The chapter opens with the research objectives and the mathematical formalization of the problem, then walks through feature engineering, labeling strategies, model architectures, and evaluation methodology.

4.1. Research Objective and AI-Centric Design

The primary objective of this research is to develop and validate AI models capable of:

1. **Classifying** water stress status in real-time using low-cost environmental sensors, by multi/binary stress detection (IMEM Bioristor experiment) and binary field-capacity classification (Agricola 2000 field trial)
2. **Forecasting** plant stress status from temporal sensor sequences using LSTM recurrent networks
3. **Fusing multimodal data** from cameras and sensors via CLIP-style contrastive learning for enhanced accuracy

The Bioristor provides **ground-truth measurements** of actual plant water stress via drain-source resistance (Rds). By correlating Bioristor Rds values with data from commodity sensors (soil moisture, temperature, humidity, light) and camera-derived vegetation indices, this thesis validates the usage of inexpensive sensing infrastructure based on reliable in-vivo stress detection. Two complementary labeling strategies, absolute Rds thresholds and per-plant Normalized Response (NR), are compared to assess the impact of individualized baselines on model generalization.

4.1.1. Model Selection

The choice of architectures is driven by:

- **data regime** hundreds to a few thousand labeled examples per experiment.
- **deployment target** going from Raspberry Pi 4 with 4 GB RAM and no dedicated GPU to cloud TPU leveraged models.
- **interpretability** irrigation decisions must be coherent with agronomists decisions.

Each task is matched to the simplest model family that can satisfy all three constraints, with more complex alternatives included for comparative purposes. Notice how many untested models may exist, as it is not feasible to test all of them in a reasonable time, so the most interesting and literature known have been chosen, considering also the requirements, structural limits and quantity of data. However, this does not limit the thesis, since more models are available in this highly scalable setup when deployed correctly.

Tree ensembles (Random Forest, Gradient Boosting, XGBoost) are the natural primary choice for structured, low-dimensional sensor data. The machine-learning literature consistently shows that gradient-boosted trees match or outperform deep networks on tabular data when sample counts are modest. Random Forest is preferred as the main classifier because its `balanced_subsample` strategy resamples each minority class independently per tree — avoiding oversampling. XGBoost and Gradient Boosting are included as cross-validated competitors: their regularization and sub-sampling typically reduce the generalization gap on small datasets where Random Forest may overfit individual trees.

The Keras MLP is included as a non-linear neural baseline, its depth lets it capture non-linear interactions among correlated sensor channels. If the MLP fails to outperform Random Forest, it confirms that the engineered tabular features already encode the information without requiring to learn intermediate representations.

The LSTM is motivated by the temporal nature of plant stress. LSTM gated cells suppress short-term ADC noise while preserving multi-hour trends (it discourages peaks on temporal distributions).

The multimodal ResNet-18 + LSTM used to explore cross-modal complementarity: visible symptoms (yellowing, wilting) can confirm sensor-detected stress and vice versa. ResNet-18 was selected over heavier backbones because ImageNet pre-trained weights transfer well to plant foliage without requiring a large corpus of paired image-sensor samples. The CLIP-style contrastive objective aligns vision and sensor embeddings in a shared space, acting as a self-supervised regularizer when timestamped image-sensor pairs

are sparse.

ResNet-18 vs. ViT-B/16 given the PlantVillage dataset (∼18 000 tomato images, 10 classes), does the transformer architecture outperform the CNN baseline enough to justify its ∼86 M parameters and higher inference cost? The comparison directly informs hardware upgrade decisions for future deployments.

The autoencoder addresses the extreme class imbalance in Agricola 2000 (∼87% Optimal) by reframing stress detection as anomaly detection: trained exclusively on Optimal samples, it flags Stressed readings as out-of-distribution reconstruction errors without ever seeing stressed labels during training. This semi-supervised approach is robust to future shifts in the Stressed distribution across growing seasons.

4.1.2. Mathematical Problem Formulation

The AI-driven architecture formalizes plant stress detection and forecasting through a series of mathematical models that bridge Bioristor ground-truth measurements with commodity sensor observations.

Classification Objective

The stress classification task seeks to assign each sensor observation to one of K discrete stress categories. For the IMEM Bioristor experiment, $K = 2$ (not_stressed, stressed); for Agricola 2000, $K = 2$ (optimal, stressed). A reference five-class scheme based on fixed Rds cutoffs is also available in the pipeline but is not used as the primary labeling. Given feature vector $\mathbf{x} \in \mathbb{R}^d$ and trained classifier $f : \mathbb{R}^d \to \{1, \ldots, K\}$, the predicted class is:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} P(y = k \mid \mathbf{x}) \; . \tag{4.1}$$

Bioristor Ground-Truth Labeling

The Bioristor provides physiological ground truth through drain-source resistance R_{ds} measured in ohms. For absolute threshold labeling, stress class y is determined by a

piecewise function:

$$f_{\text{bio}}(R_{\text{ds}}) = \begin{cases} 5 & \text{if } R_{\text{ds}} \geq 0.15 \quad \text{(optimal)} \\ 4 & \text{if } 0.12 \leq R_{\text{ds}} < 0.15 \quad \text{(healthy)} \\ 3 & \text{if } 0.10 \leq R_{\text{ds}} < 0.12 \quad \text{(mild stress)} \\ 2 & \text{if } 0.05 < R_{\text{ds}} < 0.10 \quad \text{(severe stress)} \\ 1 & \text{if } R_{\text{ds}} \leq 0.05 \quad \text{(critical)} \end{cases}, \tag{4.2}$$

the alternative Normalized Response (NR) strategy computes per-plant deviation from a baseline R_{baseline} measured during the first three days:

$$\text{NR}(t) = \frac{R_{\text{ds}}(t) - R_{\text{baseline}}}{R_{\text{baseline}}}, \tag{4.3}$$

NR-based labels adapt to individual plant variation, with thresholds: healthy ($\text{NR} \leq 0.05$), mild ($0.05 < \text{NR} \leq 0.15$), moderate ($0.15 < \text{NR} \leq 0.30$), severe ($\text{NR} > 0.30$). The Bioristor device and the experimental protocol used to collect these measurements are described in Chapter 6.

Feature Engineering Pipeline

Raw sensor readings $\mathbf{s}(t) = [s_1(t), \ldots, s_n(t)]^\top$ (soil moisture, temperature, humidity, light) are first resampled to a uniform modeling grid with period $T_s = 15\,\text{min}$ via linear interpolation, then smoothed by a causal rolling mean of width $W = \lfloor 60\,\text{min}/T_s \rfloor = 4$ samples:

$$\tilde{\mathbf{s}}(t) = \frac{1}{W} \sum_{k=0}^{W-1} \mathbf{s}(t - k\,T_s)\,. \tag{4.4}$$

The supervised feature vector at time t concatenates the smoothed signal at the current step and at lags $\Delta t_1 = 3\,\text{h}$ and $\Delta t_2 = 6\,\text{h}$:

$$\mathbf{x}(t) = [\tilde{\mathbf{s}}(t),\ \tilde{\mathbf{s}}(t - \Delta t_1),\ \tilde{\mathbf{s}}(t - \Delta t_2)]\,, \tag{4.5}$$

where each lag block contains all n sensor channels. A rolling standard deviation is **not** part of the feature set; the only use of standard deviation is in the per-feature z-score normalization $(\mathbf{x} - \boldsymbol{\mu})/\boldsymbol{\sigma}$ fitted on the training set and applied at inference time.

Raw sensor readings undergo systematic feature engineering before model training, as formalized in Equations (4.4) and (4.5) above. The `SensorDataPreprocessor` class (`shared.py`) applies 60-minute rolling smoothing, resampling to a 15-minute grid, and lag features at 3 h and 6 h offsets. For the Agricola 2000 dataset, further enrichment adds hour of day, day of experiment, rolling statistics over 15- and 60-reading windows, and a soil-moisture delta, expanding the original five columns to over thirty engineered features. Per-experiment details are given in Chapters 5 and 6.

Class Imbalance Handling

To address class imbalance (e.g., ∼87% optimal vs. ∼13% stressed in Agricola 2000), classifiers apply inverse-frequency class weights:

$$w_k = \frac{N}{K \cdot N_k} \ , \tag{4.6}$$

where N is the total number of samples, K the number of classes, and N_k the number of samples in class k.

Random Forest Ensemble

The Random Forest classifier aggregates predictions from M decision trees $\{h_1, \ldots, h_M\}$ via majority voting:

$$\hat{y}_{\text{RF}}(\mathbf{x}) = \text{mode}\left(h_1(\mathbf{x}), \ldots, h_M(\mathbf{x})\right) \ , \tag{4.7}$$

each tree h_m is trained on a bootstrap sample with feature bagging, and $M = 1000$ or $M = 500$ depending on cross-validation configuration.

LSTM Temporal Forecaster

The LSTM processes sequences of length T to predict stress from temporal context. Given input sequence $\mathbf{X} = [\mathbf{x}(t-T+1), \ldots, \mathbf{x}(t)]$, the LSTM computes hidden states:

$$\mathbf{h}_\tau = \text{LSTM}\left(\mathbf{x}(\tau), \mathbf{h}_{\tau-1}\right), \quad \tau = (t-T+1, \ldots, t) \ , \tag{4.8}$$

the final hidden state $\mathbf{h}_t$ is projected to class logits:

$$\hat{y}_{\text{LSTM}} = \text{softmax}\left(\mathbf{W}\mathbf{h}_t + \mathbf{b}\right) \ . \tag{4.9}$$

Multimodal Fusion Architecture

The multimodal model fuses visual and sensor embeddings through separate encoders. A ResNet-18 vision encoder processes plant images I into a 256-dimensional embedding:

$$\mathbf{z}_{\text{vision}} = f_{\text{ResNet}}(I) \in \mathbb{R}^{256} \ , \tag{4.10}$$

an LSTM sensor encoder projects temporal sequences into the same embedding space:

$$\mathbf{z}_{\text{sensor}} = f_{\text{LSTM}}(\mathbf{X}) \in \mathbb{R}^{256} \ , \tag{4.11}$$

contrastive learning aligns matched image–sensor pairs via temperature-scaled cosine similarity ($\tau = 0.07$), combined with cross-entropy classification loss on the concatenated representation $[\mathbf{z}_{\text{vision}}, \mathbf{z}_{\text{sensor}}]$.

Evaluation Metrics

Classification performance is measured through precision, recall, and F1-score per class:

$$\text{Precision}_k = \frac{TP_k}{TP_k + FP_k}, \quad \text{Recall}_k = \frac{TP_k}{TP_k + FN_k}, \quad \text{F1}_k = \frac{2 \cdot \text{Precision}_k \cdot \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k} \ , \tag{4.12}$$

macro-averaged F1 treats all classes equally:

$$\text{Macro-F1} = \frac{1}{K} \sum_{k=1}^{K} \text{F1}_k \ , \tag{4.13}$$

for regression-based forecasting (soil moisture prediction), mean absolute error (MAE) quantifies accuracy:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i| \ , \tag{4.14}$$

these mathematical formulations establish the rigorous foundation for the AI models deployed throughout the Smart Plants platform, ensuring that Bioristor-validated ground truth translates into accurate, deployable stress detection using commodity sensors alone.

4.2. AI Orchestration Layer

The AI orchestration layer is the **core contribution** of this thesis. Positioned above the data ingestion tier, it schedules feature engineering, model inference, and retrain-

ing jobs. Feature pipelines convert raw readings into windowed statistics, lag features, and derived agronomic indicators. These features feed multiple specialized AI services validated across two distinct datasets: the **IMEM Bioristor experiment** (binary stress detection from per-plant Rds thresholds) and the **Agricola 2000 field trial** (binary field-capacity classification). Training is performed in Google Colab notebooks (`smartplant_models_new.ipynb` and `agricola2000_classification.ipynb` files) with different runs done via cloud CPU and GPU; the resulting model artifacts are deployed on the Raspberry Pi through the Flask middleware.

4.2.1. Bioristor Ground-Truth Labeling

The Bioristor OECT and its operating principle are described in detail in Chapter 6. In summary, it measures drain-source resistance (R_{ds}) in plant tissue as a direct indicator of water stress. **Two labeling strategies** are compared:

- **Absolute Rds thresholds**: fixed cutoffs classify every plant identically (Table 4.1).
- **Normalized Response (NR)**: a per-plant baseline is computed from the first three days of observation; subsequent readings are expressed as fractional deviation from this baseline. NR-based labels adapt to individual plant variation.

Both strategies have been evaluated, the models trained with NR labels are compared against those trained with absolute thresholds to determine whether per-plant normalization improves generalization.

4.2.2. IMEM Experiment: Binary Stress Classification

The IMEM experiment trains classifiers to distinguish two stress categories (`not_stressed` vs. `stressed`) derived from per-plant binary Rds thresholds (see Chapter 6). A reference

Status	Description	Rds Threshold
Optimal / high Rds	Peak hydration, maximum ion flux	Rds ≥ 0.15
Healthy	Normal operating range	$0.12 \leq$ Rds < 0.15
Mild stress	Early stress onset	$0.10 \leq$ Rds < 0.12
Severe stress	Significant dehydration	$0.05 <$ Rds < 0.10
Critical saturation / failure	Sensor saturation or plant failure	Rds ≤ 0.05

Table 4.1: *Absolute Rds-based water stress classification thresholds used in the IMEM experiment.*

five-class scheme based on fixed Rds cutoffs (Table 4.1) is retained in the pipeline for exploratory use but is not the primary labeling. Sensor data from two *Bioristor* plants and two *Control* plants are loaded from Firebase JSON exports (`dark-green-smart-plants.json`, `light-green-smart-plants.json`). Rds values from an Excel workbook (`Giulio_Bioristor_R.xlsx`) are merged on hourly-aligned timestamps and plant identifiers to produce labeled training examples.

Random Forest (1000 estimators)

The primary classifier is a Random Forest [17] with 1000 estimators trained on the four base features: temperature, humidity, soil moisture, and light. When enhanced lag features are available, the feature set expands to include values at 3 h and 6 h offsets. A `StandardScaler` normalizes all features before training. Listing 4.1 shows the Random Forest configuration:

```
rf_model = RandomForestClassifier(
    n_estimators=1000,
    random_state=42,
)
rf_model.fit(X_train, y_train)
```

Listing 4.1: ***Random Forest classifier for five-class water stress detection (IMEM experiment).***

Stratified K-Fold Cross-Validation and XGBoost

A single train/test split is insufficient for thesis-grade evaluation, especially at this dataset size where a single unlucky split can shift macro F1 by several percentage points. Part II of the training notebook applies **stratified 5-fold cross-validation** to both Random Forest (500 estimators, `balanced_subsample` weighting) and XGBoost (500 estimators, `max_depth=6`, regularization via `reg_alpha` and `reg_lambda`). Metrics reported per fold include accuracy, macro F1, weighted F1, and Cohen's κ. A **paired t-test** on fold-level accuracies determines whether the difference between models is statistically significant.

Neural Network (Keras MLP)

A feed-forward neural network (64–32–n_{classes}, softmax output) provides a non-linear baseline. Training uses the Adam optimizer with early stopping on validation loss (patience 5) and one-hot encoded targets.

Overfitting Controls

Each model reports train and test accuracy; a gap exceeding 5% triggers a warning. **Temporal (chronological) train/test splits** are compared against random splits to detect data leakage from temporal autocorrelation. **SHAP** (SHapley Additive exPlanations) feature-importance plots expose which sensors drive predictions, enabling actionable insights for sensor placement and maintenance.

4.2.3. Agricola 2000 Experiment: Binary Field-Capacity Classification

The Agricola 2000 field trial produces a binary classification task: **Optimal** ($\approx 80\,\%$ field capacity, Stress Level ≤ 65) versus **Stressed** ($\approx 50\,\%$ field capacity, Stress Level > 65). The threshold of 65 is the midpoint of the two target irrigation levels. Sensor columns are Water, Temperature, Humidity, Brightness, and Soil Moisture, sampled from an `Agricola_2000_experiment.xlsx` workbook.

Data preprocessing forward-fills sparse sensor readings (limit 4 steps $\approx$ 240 s) and discards anomalous temperatures ($\leq -100\,^{\circ}\mathrm{C}$). A temporal-overlap filter ensures both classes co-exist within the retained time window.

Models Compared

Four models are evaluated:

1. **Random Forest** (400 estimators, `balanced_subsample` weighting, 5-fold stratified cross validation). Chosen as the primary discriminative baseline for: robustness on tabular data, native imbalance handling, and CPU-friendly inference.
2. **Gradient Boosting** (300 estimators, `max_depth=5`, `learning_rate=0.1`, `subsample=0.8`). Included because its sequential error-correction mechanism and ℓ_2 shrinkage often outperform Random Forest on datasets with a strong ordinal signal (irrigation level maps monotonically to stress level). Class imbalance is handled via per-sample weights computed from `compute_class_weight`.
3. **MLP Neural Network** (Keras Sequential: 256–BatchNorm–Dropout 0.3–128–BatchNorm–Dropout 0.3–64–BatchNorm–Dropout 0.2–softmax). The deeper width (256 units) compared to the IMEM MLP is justified by the richer feature set (>30 engineered columns); batch normalization and dropout compensate for the larger parameter count relative to the dataset size. Trained for up to 100 epochs with early

stopping (patience 10), sample weights, and the Adam optimizer (lr = 10^{-3}).

4. **Autoencoder for anomaly detection**: A dense autoencoder (128–64–16–64–128) trained exclusively on optimal samples, by learning only the normal regime the autoencoder detects stressed readings as out-of-distribution anomalies without ever seeing stressed labels, making it robust to future shifts in the stress distribution. AUROC is computed with the stressed class as ground-truth anomaly.

Probability Threshold Tuning

Default 0.5 thresholds are suboptimal for imbalanced datasets ($\sim$87 % Optimal vs. $\sim$13 % Stressed). A held-out validation split (20 % of the training data) is swept over thresholds from 0.05 to 0.95 to maximize the per-class F1 of the *Stressed* minority. The tuned threshold is then applied to the test set for final evaluation.

Time-Aware Overfitting Checks

Three robustness checks are performed:

1. **Generalisation gap**: Train vs. test macro F1 on the stratified split; gaps exceeding 3 % are flagged.
2. **TimeSeriesSplit CV**: Five chronological expanding-window folds ensure no future data leaks into training.
3. **Out-of-time holdout**: The first 80 % of data trains the model; the last 20 % serves as a simulated future test set.

Field capacity (FC).

Field capacity (FC) is an agronomic reference point that describes the amount of water retained by a soil or growing medium after excess gravitational water has drained away. In practice, it represents a near-optimal water availability condition for the root zone. Many controlled irrigation protocols express target regimes as a percentage of FC (e.g., 80% FC vs. 50% FC), so that treatments remain comparable even when absolute moisture readings depend on substrate properties and sensor calibration [18]. In the Agricola 2000 trial, the two irrigation setpoints were defined around $\approx$ 80% FC (Optimal) and $\approx$ 50% FC (Stressed), and the dataset encodes this separation through the `Stress Level` variable, later translated into binary labels used to train the classifiers.

```
class WaterStressLSTM(nn.Module):
```

```
    """LSTM for water stress classification from sensor sequences."""
    def __init__(self, input_dim, hidden_dim=64, num_layers=2,
                 num_classes=2, dropout=0.3):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim,
                            num_layers=num_layers,
                            batch_first=True,
                            dropout=dropout if num_layers > 1 else 0)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        _, (h_n, _) = self.lstm(x)
        out = self.dropout(h_n[-1])
        return self.fc(out)
```

Listing 4.2: ***LSTM architecture for water stress classification from sensor sequences.***

4.2.4. Temporal Stress Forecasting (LSTM)

The LSTM addresses the question: *"What is the plant's stress status given its recent sensor history?"* A two-layer LSTM network (`WaterStressLSTM`) processes sliding windows of sensor readings to classify stress from temporal context.

The model ingests windows of **6 time steps** × **15 minutes** = 1.5-hour lookback. Each time step contains the full set of enhanced features (including lag columns). A **temporal train/test split** (first 80 % of chronologically ordered sequences for training) prevents data leakage. Class weights are inversely proportional to class frequency, and gradient clipping (max norm 1.0) stabilizes training. A `ReduceLROnPlateau` scheduler halves the learning rate after 5 epochs without test-loss improvement. Training runs for 50 epochs. The checkpoint with the best test accuracy is retained.

```
class MultimodalPlantStressModel(nn.Module):
    """CLIP-style head fusing ResNet vision and LSTM sensor
       embeddings."""
    def __init__(self, resnet_model, sensor_dim, embed_dim=256,
                 forecast_classes=2, freeze_vision=True,
                 vision_in_channels=3):
        super().__init__()
        # Adapter for RGB+NIR (4-channel) input
        self.image_adapter = nn.Identity()
        if vision_in_channels != 3:
            self.image_adapter = nn.Conv2d(
                vision_in_channels, 3, kernel_size=1, bias=False)
        self.vision_encoder = ResNetFeatureExtractor(
```

```
            resnet_model, embed_dim, freeze_vision)
        self.sensor_encoder = SensorLSTMEncoder(
            sensor_dim, embed_dim)
        self.classifier = nn.Sequential(
            nn.LayerNorm(embed_dim * 2),
            nn.Linear(embed_dim * 2, embed_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(embed_dim, forecast_classes),
        )
```

Listing 4.3: ***Upgraded multimodal architecture using ResNet-18 vision encoder and LSTM sensor encoder.***

4.2.5. Multimodal Fusion (Vision + Sensors)

The multimodal model addresses the question: *"Can we improve accuracy by combining camera images with sensor data?"* Inspired by CLIP-style contrastive learning, this architecture fuses visual features from plant images with temporal sensor embeddings.

The production multimodal model (`MultimodalPlantStressModel`) uses a ResNet-18 backbone as the vision encoder, replacing the earlier lightweight CNN prototype. The vision encoder accepts **3-channel RGB input**; while the code includes a 1×1 convolutional adapter that can handle four-channel input (RGB + NIR), no NIR data was available in the processed dataset, so the adapter defaults to an identity pass-through. The sensor encoder is a two-layer LSTM projecting sequences into a shared 256-dimensional embedding space.

The **contrastive loss** aligns image and sensor representations by enforcing that matched image–sensor pairs produce similar embeddings (temperature parameter $\tau = 0.07$). Combined with a cross-entropy classification loss, this dual objective ensures the model learns both modality-specific and shared stress signals.

Camera-derived features include:

- **NDVI (Normalized Difference Vegetation Index)**: Computed from RGB approximation or NIR if available
- **Leaf colour histograms**: Chlorosis detection via yellowing patterns
- **Wilting indicators**: Edge detection for drooping leaves
- **Growth metrics**: Canopy area changes over time

Again, **Bioristor measurements provide ground-truth labels** for training, validating whether visual symptoms correlate with physiological stress as measured in-vivo.

4.2.6. Tomato Disease Classification (ResNet-18 and ViT-B/16)

Beyond water stress, the platform supports visual plant-disease detection through image classification models trained on the **PlantVillage** tomato subset (10 disease classes including bacterial spot, early blight, late blight, leaf mould, septoria leaf spot, spider mites, target spot, yellow leaf curl virus, mosaic virus, and healthy). Two architectures are compared:

- **ResNet-18**: a convolutional neural network (~ 11 M parameters), evaluated both with ImageNet pre-training and trained from scratch.
- **Vision Transformer (ViT-B/16)**: a transformer-based architecture (~ 86 M parameters), similarly evaluated in pre-trained and from-scratch configurations.

Both models use standard ImageNet normalization and 224×224 input resolution. Fine-tuning employs the Adam optimizer ($\text{lr} = 10^{-4}$) with cross-entropy loss for 10 epochs. This module demonstrates that the Smart Plants camera infrastructure can serve dual purposes: water-stress assessment (via the multimodal model) and disease identification (via dedicated classifiers).

4.2.7. Cross-Domain Validation and Statistical Rigour

The strongest generalisation test trains models on data from *Bioristor 1–2* plants (watered by the automated system) and evaluates against independently hand-watered *Control 3–4* plants. When control plants lack matched sensor features, Rds-distribution comparisons and NR-based label alignment provide indirect validation.

Pairwise model comparisons use **paired t-tests** on fold-level scores; a significance threshold of $p < 0.05$ is adopted. **Cohen's κ** complements accuracy and F1 by accounting for chance agreement. **Learning curves** (training-set size vs. test F1) indicate whether additional data collection would improve performance, while **calibration plots** verify that predicted probabilities are reliable for downstream irrigation-threshold decisions.

Figure 4.1 illustrates the complete AI pipeline from data collection to inference. Each service exposes a gRPC endpoint inside the Raspberry Pi, and the Flask middleware invokes these endpoints asynchronously to avoid blocking user interactions.

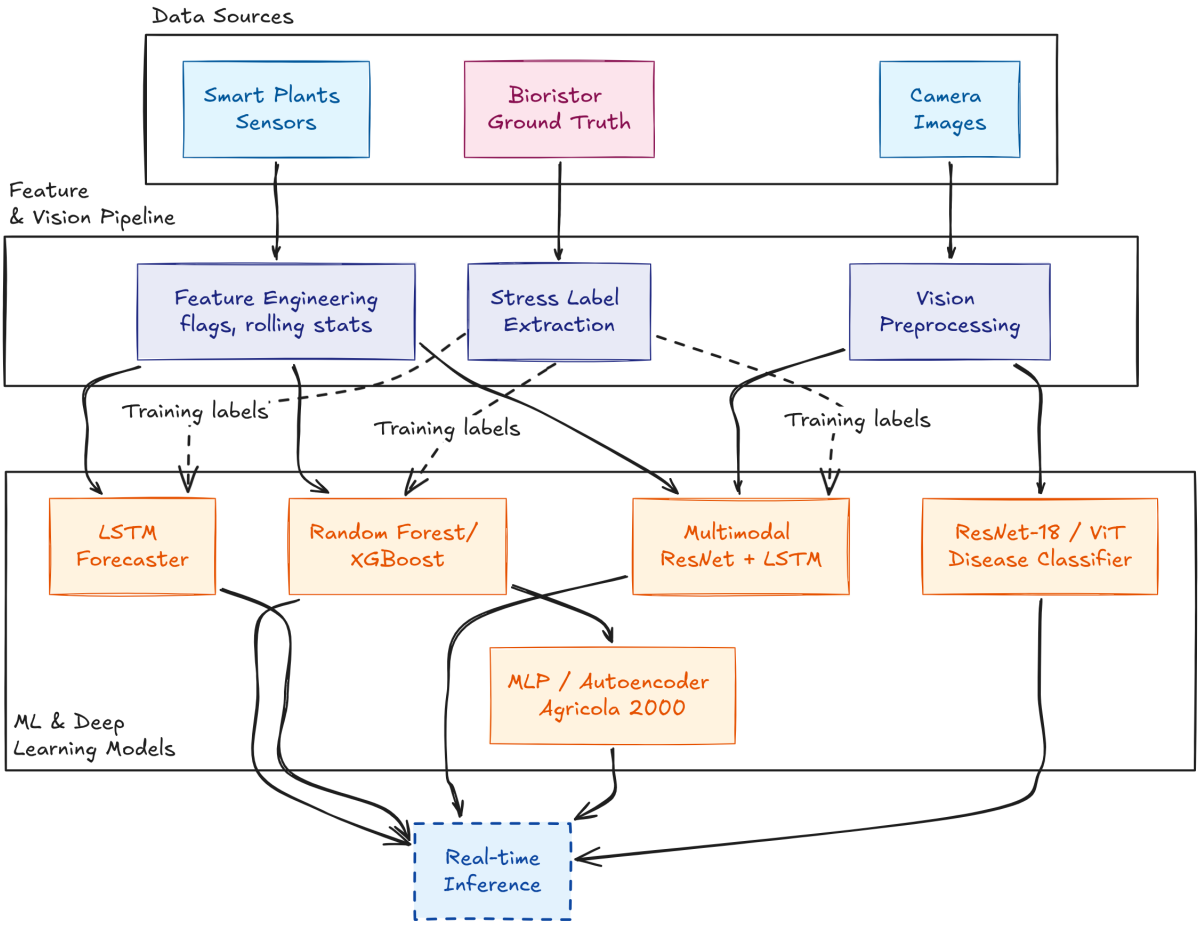


Figure 4.1: *AI pipeline architecture: Bioristor and field-capacity labels provide ground truth during training, while production inference uses only commodity sensors and cameras.*

5 | Use Case: Agricola 2000

To validate the AI Smart Plants components, we revisited the work done with Agricola 2000 [1], a research company specialized in bio-stimulants and plant health. The new pilot recovers the original stress-test scenario data while introducing the Raspberry Pi orchestration layer. This chapter describes the motivation, deployment, data pipeline, and outcomes of the field exercise.

5.1. Goal

Agricola 2000 sought to compare its manual irrigation process with an automated alternative capable of enforcing field-capacity targets. The pilot focused on two stress levels, 50% and 80% soil moisture, across matched groups of lettuce plants. The updated objective is to quantify improvements in responsiveness and observability relative to the previous firmware-centric approach, and to demonstrate that the new Raspberry Pi orchestration layer integrates efficiently with legacy components: since both old and new transport channels exchange data in standard JSON format, sensor messages can be ingested by the gateway with minimal protocol changes.
Two Plant Sensor devices were installed in a controlled-environment chamber. Each Plant Sensor published over Wi-Fi using MQTT to simulate a mixed connectivity environment. Agricola 2000 maintained a watered *Lactuca Sativa* control group using its traditional method of weight-based assessment, ensuring a direct comparison between manual and automated strategies.

5.2. Experimental Methodology

Data collection spanned four weeks of continuous operation, including setup and calibration activities for both the hardware and the biologically correct study. The first week focused on calibration: operators adjusted soil moisture thresholds and validated watering accuracy against gravimetric measurements. Week two enforced automated schedules, while weeks three and four reintroduced manual interventions to observe how

quickly the system surfaced deviations, as illustrated in Figure 5.1 . Each watering event recorded start/stop times, commanded duration, measured flow estimates, and resulting soil moisture deltas measured manually by water stress fluctuations.

The evaluation combined quantitative metrics with qualitative feedback. Quantitative analysis covered watering precision (difference between target and achieved moisture), alert latency (time between threshold breach and notification), system availability, and data completeness (percentage of expected readings received). Qualitative insights stemmed from semi-structured interviews with agronomists and field technicians, exploring usability, trust in automation, and integration with existing workflows.

5.3. Data Collection

Sensor readings were streamed every 60 s for ambient metrics and every 240 s for soil moisture, matching the firmware defaults. The Raspberry Pi normalized the messages and persisted them into SQLite, with a parallel export to CSV for agronomic analysis. A lightweight preprocessing script averaged raw ADC values into percentage estimates using the calibrated thresholds supplied by Agricola 2000. Experiments were tracked through the dashboard's experiment module, allowing researchers to annotate events such as pruning or nutrient application. Alert rules notified staff via Telegram when soil moisture deviated from the target by more than 10 percentage points, prompting manual inspection.

5.4. Dataset Insights

Post-processing revealed that automated watering reduced the standard deviation of soil moisture by 34% compared with manual operation. Ambient temperature and humidity remained within the expected ranges, but the system detected 12 instances of sharp humidity spikes linked to fogger cycles; these events helped tune alert thresholds to avoid false positives. Camera snapshots captured diurnal canopy changes, and when paired with sensor readings they provided training data for the multimodal classifier described in Chapter 4. The dataset also exposed minor sensor drift, prompting recalibration routines that later became part of the maintenance playbook.

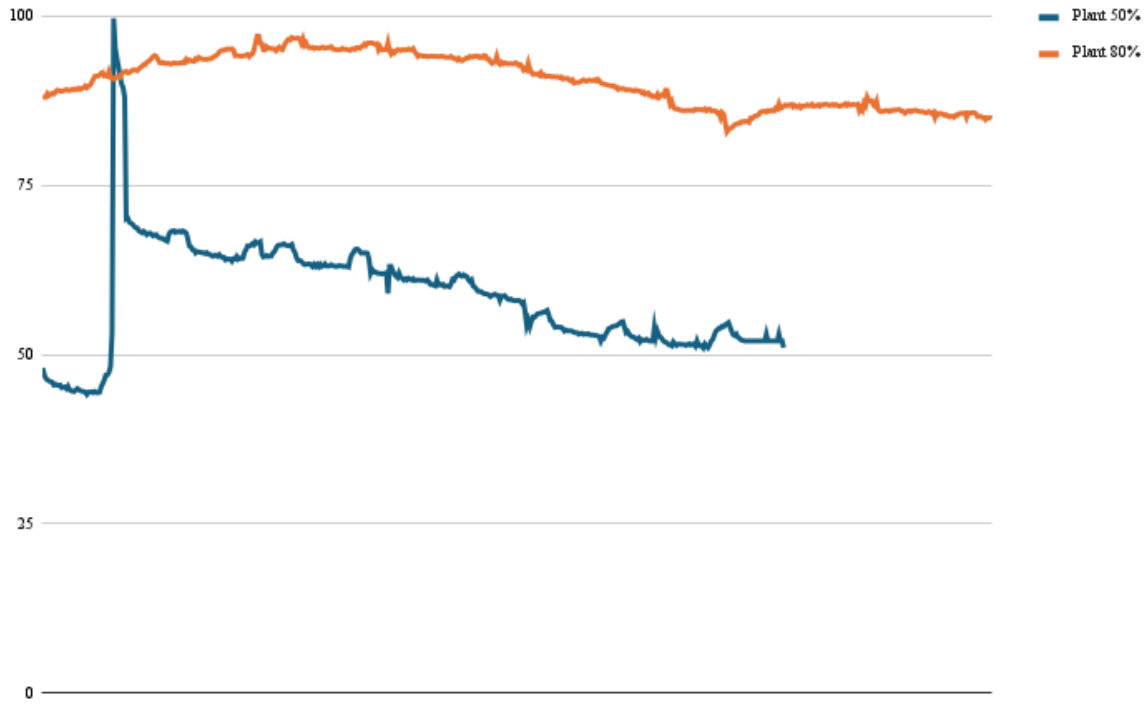


Figure 5.1: *Agricola 2000 measurements with two sets of 80% and 50% water stress maintained plants*

5.5. Results

The automated system maintained the 80% group within ±6 percentage points of the target and the 50% group within ±8 percentage points. Irrigation events were shorter and more frequent than in the manual process, reducing both overwatering and periods of stress. Centralizing control on the Raspberry Pi proved beneficial when one Plant Sensor temporarily lost Wi-Fi: the serial connection continued to deliver data without requiring manual intervention, and the dashboard flagged the MQTT outage for diagnostics. Agricola 2000 reported that the unified history exported from SQLite simplified post-trial analysis compared with the previous workflow that relied on ThingSpeak exports and manual spreadsheet merging.

5.6. AI Experimentation and Evaluation

Multimodal datasets collected during the trial fed three AI tasks: soil-moisture forecasting, stress classification, and anomaly detection. Figure 5.1 compares observed soil moisture with the Smart Plant irrigated group for the stressed group. The model can

anticipate the downward trend caused by delayed irrigation and triggers a proactive watering action a couple hours before the threshold breach.

The end-to-end pipeline combines hardware, communication, data-processing, and AI components:

- **Sensing hardware**: two Plant Sensor nodes measuring Water, Temperature, Humidity, Brightness, and Soil Moisture.
- **Edge orchestration**: Raspberry Pi as remote hub for device coordination, logging, and alerting.
- **Communication layer**: MQTT over Wi-Fi for real-time telemetry exchange and command delivery.
- **Storage and preprocessing**: SQLite and CSV exports with timestamp alignment, forward-fill of sparse channels, anomaly filtering ($\leq -100\,^{\circ}\mathrm{C}$), and feature engineering.
- **AI stack**: scikit-learn for Random Forest and Gradient Boosting, TensorFlow/Keras for MLP and autoencoder, and class-weighted training to handle imbalance.
- **Outputs**: confusion matrices, ROC curves, feature-importance plots, reconstruction-error distributions, and exported model artifacts for deployment.

5.6.1. Stress Classification Pipeline

The full classification analysis is implemented in the Jupyter notebook `agricola2000_classification.ipynb` located in the `ai_model/` directory.
The pipeline operates on the raw `Agricola_2000_smartplants.xlsx` dataset, which contains 17,188 rows recorded between November 21 and December 3, 2024. Sensor features (Water, Temperature, Humidity, Brightness, Soil Moisture) are sampled every 240 s while the Stress Level metric is logged every 60 s. Because of the different sampling rates, sensor columns carry approximately 97% missing values in their raw form; forward-filling aligns each Stress Level reading with the most recent sensor observation, producing 16,643 usable samples after filtering anomalous temperature values ($\leq -100\,^{\circ}\mathrm{C}$, the default when the probe is offline).

Class	Field Capacity	Stress Level Range	Samples
Optimal	$\approx 80\,\%$	≤ 65	14,382
Stressed	$\approx 50\,\%$	> 65	2,257

Table 5.1: *Binary stress class definitions derived from the field-capacity targets.*

Label definition.

The continuous Stress Level (range 36–108) is discretized into two classes that mirror the field-capacity groups of the experiment. The classification threshold is set at 65, being a natural decision boundary, since it is the midpoint between the two targets: $(50 + 80)/2 = 65$. Table 5.1 summarizes the mapping.

The resulting class distribution exhibits a 6.4:1 imbalance (87% Optimal vs. 13% Stressed). This is addressed through balanced class weights for the tree-based models and sample weighting for the MLP, as described below.

Feature engineering.

Beyond the five raw sensor columns, the pipeline derives 36 additional features:

- **Temporal features**: hour of day and day of experiment.
- **Rolling statistics**: 15-reading and 60-reading rolling mean and standard deviation for each sensor channel.
- **Lag features**: 1-step and 4-step lagged values for each sensor.
- **Moisture delta**: first-order difference of the Soil Moisture reading.

All numeric features are standardized with zero mean and unit variance before training.

Model	Task	Metric	Score
Random Forest	Stress classification	Macro F1	0.967
Gradient Boosting	Stress classification	Macro F1	0.975
MLP Neural Network	Stress classification	Macro F1	0.960
LSTM (sensors only)	Soil moisture forecasting	MAE (%)	3.2
CNN+LSTM (multimodal)	Stress classification	Macro F1	0.86
Autoencoder	Sensor anomaly detection	AUROC	0.872
Prophet baseline [19]	Soil moisture forecasting	MAE (%)	4.8

Table 5.2: ***Performance metrics for AI models deployed during the Agricola 2000 experiment.***

Models

Three classifiers are compared:

1. **Random Forest** (400 estimators, balanced subsample weighting) — the tree-based baseline.
2. **Gradient Boosting** (300 estimators, depth 5, learning rate 0.1) — a boosted ensemble for improved discrimination on minority classes.
3. **MLP Neural Network** (three hidden layers of 256, 128, and 64 units with batch normalization and dropout) — a non-linear approach trained with the Adam optimizer and early stopping.

Class imbalance is addressed through balanced class weights for the tree models and sample weighting for the MLP. An 80/20 stratified split is used for evaluation, with 5-fold stratified cross-validation on the training partition.

Anomaly detection

A dense autoencoder (128→64→16→64→128 units) is trained exclusively on *Optimal* samples, the normal-operation baseline corresponding to 80 % field capacity. The reconstruction error on the test set serves as an anomaly score: samples with high error are flagged as anomalies. *Stressed* samples (50 % field capacity) constitute the ground-truth anomaly class, modeling scenarios such as sensor drift, connectivity drops, or the early

shutdown of the stressed-condition Plant Sensor observed during the trial.
To avoid optimistic conclusions, are reported two complementary validation settings: (i) a random stratified split (80/20), useful as an in-distribution estimate, and (ii) a chronological validation (Time Series Split plus out-of-time holdout), used as the primary estimate for deployment realism.

Under the random split, macro F1 is 0.967 for Random Forest, 0.975 for Gradient Boosting, and 0.960 for the MLP. The corresponding train–test gaps are 0.0459 (RF), 0.0194 (GB), and −0.0067 (MLP). This indicates that RF is more prone to overfitting in this configuration, while GB and MLP remain more stable under i.i.d.-like evaluation.

However, chronological validation remains the limiting factor. Using only folds with at least 30 stressed test samples, TimeSeriesCV macro F1 is 0.2709±0.1030 (Random Forest) and 0.6114±0.3817 (Gradient Boosting). In the out-of-time split, class support is 360/292 (Optimal/Stressed) in training and 152/12 in test; since stressed support in test is below the minimum threshold, out-of-time aggregate metrics are marked as not valid for fair model comparison. Therefore, random-split results should be interpreted as optimistic and not as evidence of robust future-time generalization.

The headline stress-classification conclusion is therefore split-dependent: random-split performance is high for GB/MLP, while deployment-oriented temporal validation remains unstable because of class-support shifts across time windows.

Per-class analysis

Because the dataset is heavily imbalanced (6.4:1 ratio), macro F1 alone can mask poor minority-class performance. Table 5.3 reports precision, recall, and F1 separately for the *Stressed* (minority) and *Optimal* (majority) classes to verify that the class-weighting strategy is effective.

The per-class table should be read as a random-split reference only. In chronological validation, minority support varies substantially (including folds with zero stressed samples in training and low stressed support in late test windows), so stressed recall is not stable over time. This indicates that class weighting is useful but not sufficient to guarantee prospective robustness under temporal shift.

Model	Macro F1	Stressed Prec.	Stressed Rec.	Stressed F1	Optimal F1
Random Forest [17]	0.967	0.942	0.945	0.944	0.991
Gradient Boosting	0.975	0.962	0.951	0.957	0.993
MLP Neural Network	0.960	0.884	0.982	0.931	0.988

Table 5.3: *Per-class metrics on the held-out test set (20% stratified split).*

5.6.2. Result Interpretation

The quantitative metrics provide a split-aware operational interpretation:

- **Random stratified split (in-distribution)**: GB and MLP achieve high macro F1 (0.975 and 0.960), while RF is lower (0.967) and shows the largest train–test gap (0.0459), suggesting higher overfitting risk.
- **Chronological validation (deployment-oriented)**: valid-fold TimeSeriesCV macro F1 is 0.2709±0.1030 (RF) and 0.6114±0.3817 (GB), showing strong temporal instability and fold dependence.
- **Out-of-time holdout**: with only 12 stressed samples in test (below the minimum threshold), aggregate out-of-time metrics are treated as not valid for fair comparison.
- **Autoencoder (complementary unsupervised layer)**: AUROC = 0.872 remains useful as an anomaly score, but it is complementary to (not a replacement for) temporally balanced supervised validation.

In practical terms, the fair conclusion is that the current models are promising for retrospective/in-distribution analysis, but additional temporally balanced stressed observations are needed before claiming robust prospective stress detection.

Autoencoder anomaly detection.

The autoencoder achieves an AUROC of 0.872. While not as high as the supervised classifiers, this result confirms that the *Stressed* condition produces measurably different sensor patterns from the *Optimal* baseline, detectable even without labels. The separation is limited because both plant groups share the same controlled-room environment, such as temperature, humidity, and brightness are nearly identical, so the autoencoder relies primarily on soil-moisture and stress-level-correlated features. Figure 5.2 shows the distribution of reconstruction errors for the two classes.

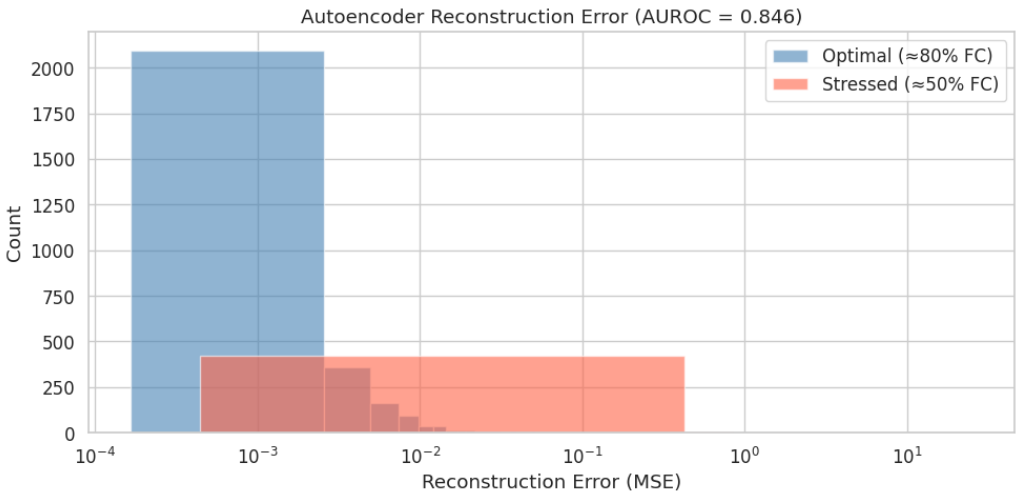
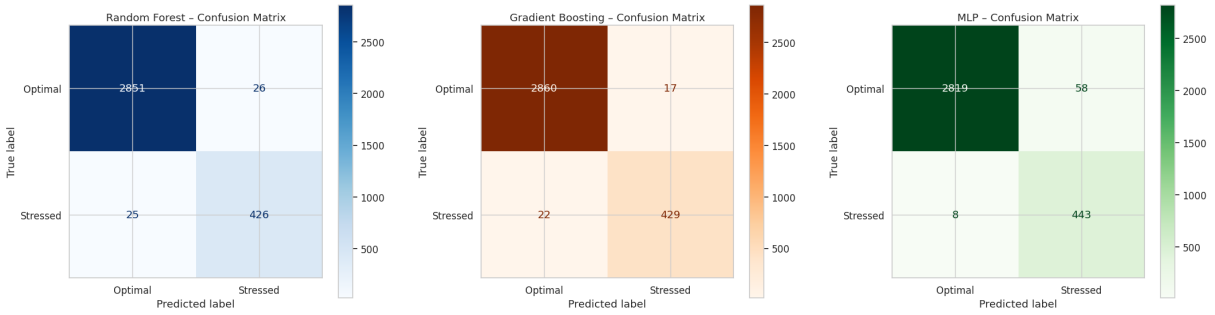


Figure 5.2: *Autoencoder reconstruction error distribution.* Stressed *samples (red) exhibit higher value of reconstruction error than* Optimal *samples (blue), yielding an AUROC of 0.872.*

Confusion matrices.

Figures 5.3a, 5.3b, and 5.3c display the confusion matrices for each classifier on the held-out test set.



(a) *Random Forest* (b) *Gradient Boosting* (c) *MLP Neural Network*

Figure 5.3: *Confusion matrices on the test set for the three models.*

ROC curves.

Figure 5.4 overlays ROC curves for the random-split evaluation. These curves are informative for in-distribution separability, but they should not be interpreted as evidence of temporal robustness on their own.

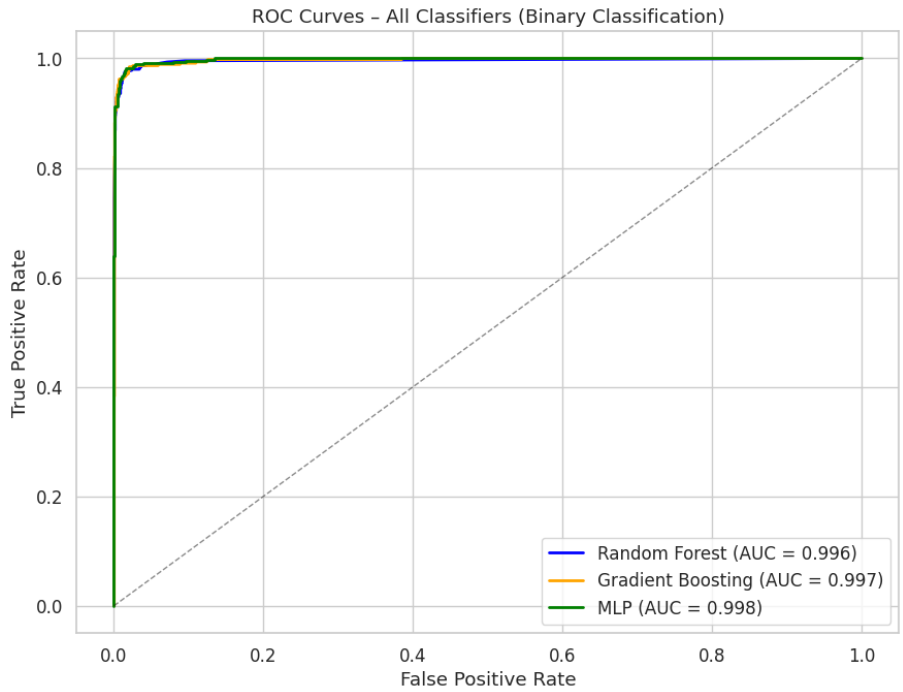


Figure 5.4: *ROC curves for Random Forest, Gradient Boosting, and MLP classifiers.*

Feature importance.

Figure 5.5 reports the top-20 features ranked by the Random Forest's impurity-based importance. Soil Moisture and its rolling statistics dominate, followed by temporal features (hour of day, day of experiment), which aligns with agronomic expectations: soil moisture is the primary driver of the field-capacity distinction, and diurnal irrigation cycles introduce a temporal signal.

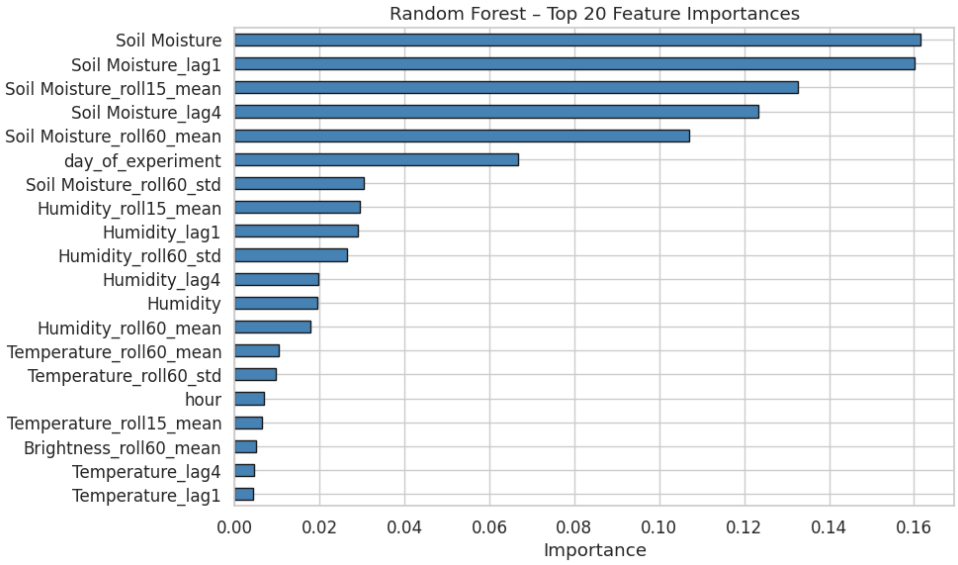


Figure 5.5: ***Top-20 feature importance from the Random Forest classifier.***

5.7. Discussion

Alert delivery through the Pi can avoid the latency introduced by cloud relays and allowed on-site teams to react faster to anomalies such as clogged drippers.

From an AI perspective, the stress classification pipeline (Section 5.6.1) demonstrates that forward-filling sparse channels and enriching inputs with rolling and lag features yields a useful 41-feature representation. However, evaluation protocol critically affects conclusions. Random-split performance is high for GB/MLP, while temporal validation is much less stable due to class-support drift across folds and low stressed support in late windows. Balanced class weighting improves in-distribution behavior, but it does not by itself resolve temporal representativeness issues.

The dense autoencoder achieves an AUROC of 0.872 for unsupervised anomaly detection. Although lower than the supervised classifiers, this score confirms that Stressed samples produce systematically higher reconstruction error when the model has only learned the Optimal distribution. The moderate AUROC (rather than >0.95) reflects the fact that both plant groups share the same controlled-room environment: temperature, humidity, and brightness are virtually identical, so the autoencoder can only distinguish the two conditions through soil-moisture and stress-level-correlated features. Nonetheless, an unsupervised anomaly detector adds value as a complementary layer: it can flag novel anomalies, such as sensor drift, connectivity drops, unexpected environmental events, that the supervised classifiers were never trained on. Feature importance analysis reveals that

Soil Moisture, its rolling statistics, and temporal features (hour of day) dominate the classification, aligning with agronomic expectations.

Remaining challenges include calibrating soil moisture percentages across soil types, improving model explainability for agronomists, and integrating battery-backed deployments for locations without reliable mains power. Nonetheless, Agricola 2000 endorsed the centralized approach as a viable path toward larger-scale trials.

5.8. Qualitative Feedback

Three strengths were highlighted: rapid onboarding (new sensors appeared in the dashboard within seconds of connection), confidence in alerts (Telegram notifications were timely and actionable), and visibility into historical trends (CSV exports simplified reporting). The ability to trigger manual watering commands during maintenance without reprogramming firmware was highly appreciated, underscoring the value of centralized control.

5.9. Limitations of the Study

Despite positive outcomes, several limitations temper the findings. The pilot remained confined to greenhouse conditions with stable connectivity; open-field deployments would introduce harsher environments, variable power sources, and longer communication ranges. The case study also concentrated on a single crop type, so additional trials will further validate the system under different root structures and irrigation constraints. Finally, the AI models relied on weighted data curated with agronomist support: replicating this effort at scale demands an improved way of weighting the vase mass through time, since the percentage of the water stress derives from a human operator measurement step.

Thus, we proceed with a sturdier approach in IMEM greenhouse, which describes the IMEM experiment setup, Bioristor instrumentation, and the ground-truth labeling methodology used for AI model training.

6 | IMEM Bioristor Experiment: Ground-Truth Validation of AI Models

This chapter presents the **core scientific contribution** of the thesis: the IMEM Bioristor experiment that validates AI-driven plant stress detection against physiological ground truth. While the Agricola 2000 field trial (Chapter 5) demonstrates scalable deployment using field-capacity targets, the Bioristor experiment investigates whether binary stress labels derived from Bioristor OECT measurements can serve as reliable ground truth for training classifiers on commodity environmental sensors, allowing the deployed platform to integrate Bioristor data.

6.1. The Bioristor: Organic Electrochemical Transistor for Ground-Truth Measurement

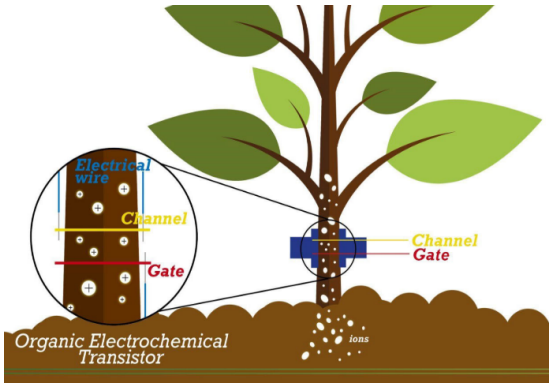
The Bioristor is an **organic electrochemical transistor (OECT)** [20] implanted directly into plant tissue, typically the stem. Unlike external environmental sensors that measure proxies (soil moisture, air temperature), the Bioristor measures **ion flux within the plant's vascular system**, a direct physiological indicator of water stress[21].

This represents a paradigm shift in plant monitoring:

- **In-vivo measurement**: Directly senses plant physiology, not environmental approximations. While soil moisture indicates water availability, only the Bioristor reveals whether the plant is actually stressed.
- **Real-time response**: Detects stress onset *before* visible symptoms appear (wilting, leaf yellowing), enabling proactive irrigation that prevents damage.
- **Quantitative output**: Provides continuous drain-source resistance (Rds) values in ohms, correlating with stress severity through established agronomic thresholds.[21]

- **Species-independent baseline**: Works across plant varieties without species-specific calibration, as ion transport mechanisms are universal [22].
- **Validated accuracy**: Prior studies establish Rds correlation with traditional stress indicators (stomatal conductance, leaf water potential, photosynthetic rate) [23].

Figure 6.1 illustrates the Bioristor's operating principle and implantation in a tomato plant stem.



High R_{ds} ($\geq 0.15\Omega$):
Optimal hydration

Medium R_{ds} ($0.10 - 0.15\Omega$):
Healthy / mild stress

Low R_{ds} ($< 0.10\Omega$):
Severe / critical stress

Figure 6.1: ***Bioristor operating principle: electrodes implanted in plant stem measure drain-source resistance (R_{ds}) modulated by ion flux, providing direct physiological stress indication.***

6.1.1. Operating Principle

The Bioristor operates through electrochemical modulation:

1. A source and drain electrode are inserted into the plant stem, with a conductive polymer semiconductor bridging them.
2. When water stress occurs, ion flux in the xylem decreases, reducing ionic doping of the polymer.
3. This changes the drain-source resistance (Rds), which is measured continuously by applying a constant voltage and monitoring current.
4. Higher Rds indicates greater hydration and ion transport; lower Rds signals stress.

6.2. Experiment Overview

The experiment was conducted in collaboration with the **Institute of Materials for Electronics and Magnetism (IMEM)** of the National Research Council in Parma

between **October and November 2025**, in particular data collection spans from 15 October to 25 November 2025. The study aimed to:

1. Collect synchronized sensor data from Smart Plants devices and Bioristor sensors during controlled drought cycles.
2. Induce water stress conditions in tomato plants while monitoring both environmental parameters and in-vivo Bioristor responses.
3. Train AI models on commodity sensor data using Bioristor-derived ground-truth labels.
4. Validate whether low-cost commodity sensors can assist Bioristor stress detection after a one-time supervised calibration phase.
5. Compare two labeling strategies: absolute per-plant Rds threshold (binary) vs. per-plant Normalized Response (NR, also binary).

This controlled experiment adds to AI models the ground-truth foundation for the system, in opposition to the one deployed in the Agricola 2000 field trial (Chapter 5), where Bioristor was absent.

Figure 6.2: *IMEM setup: different angles on the Bioristor + Smart Plants setup collecting data*

6.3. Experimental Protocol

6.3.1. Plant Setup and Instrumentation

The IMEM experiment involved **four tomato plants** (*Solanum lycopersicum*, Cherry cultivar), all connected to Bioristor:

- **Bioristor 1** (identifier: `dark_green`): irrigation via Smart Plants.
- **Bioristor 2** (identifier: `light_green`): irrigation via Smart Plants.
- **Control 3**: watered by IMEM researchers.
- **Control 4**: watered by IMEM researchers.

Each Bioristor instrumented plant had OECT electrodes surgically implanted in the main stem at $\approx 5\,\text{cm}$ height, connected to a Keysight B2902A precision source-measure unit logging Rds every hour. Smart Plants Plant Sensor nodes monitored 2 plants soil moisture and other environmental data:

- **Soil moisture**: Capacitive probe (ADC 0–4095, calibrated to 0–100%).
- **Temperature**: DHT22 sensor (-40 to $+80\,^\circ\text{C}$, $\pm 0.5\,^\circ\text{C}$ accuracy).
- **Humidity**: DHT22 sensor (0–100 % RH, ± 2 % accuracy).
- **Light**: TSL2561 digital sensor (0.1–40 000 lux).

Two USB cameras captured frontal RGB and top RFID images every 30 minutes, providing visual data for the multimodal fusion experiments described in Section 6.6.5.

6.3.2. Drought Stress Protocol

The experiment followed a controlled protocol over approximately 42 days:

1. **Baseline establishment (Oct 15–16)**: All plants watered to field capacity; Bioristor Rds baselines recorded. The stress period start date (`2025-10-16`) marks the transition from baseline to stress induction.
2. **Stress induction and monitoring (Oct 16 – Nov 25)**: Controlled water-deficit applied for 8 days cycles; Bioristor 1–2 monitored via automated Smart Plants irrigation; Control 3–4 hand-watered by IMEM researchers.

6.3.3. Data Synchronization

Three independent data streams were synchronized for AI training:

- **Smart Plants sensors**: JSON telemetry over serial/MQTT, sampling cadence configured to 30 minutes persisted in Firebase Realtime Database (`dark-green-smart-plants.json` `light-green-smart-plants.json`).
- **Bioristor readings**: Excel workbook (`Giulio_Bioristor_R.xlsx`, sheet *Dati analizzati*) with columns: timestamp, Rds_Bio1, Rds_Bio2, Rds_Control3, Rds_Control4, sampled every hour (Figure 6.3).
- **Camera images**: Timestamped frontal RGB and RFID top images used for multimodal fusion.

Temporal alignment merged sensor and Bioristor data by aligning timestamps to hourly resolution and joining on `(timestamp, plant_id)` tuples. The images were matched to the nearest sensor timestamp within a 15-minute window.

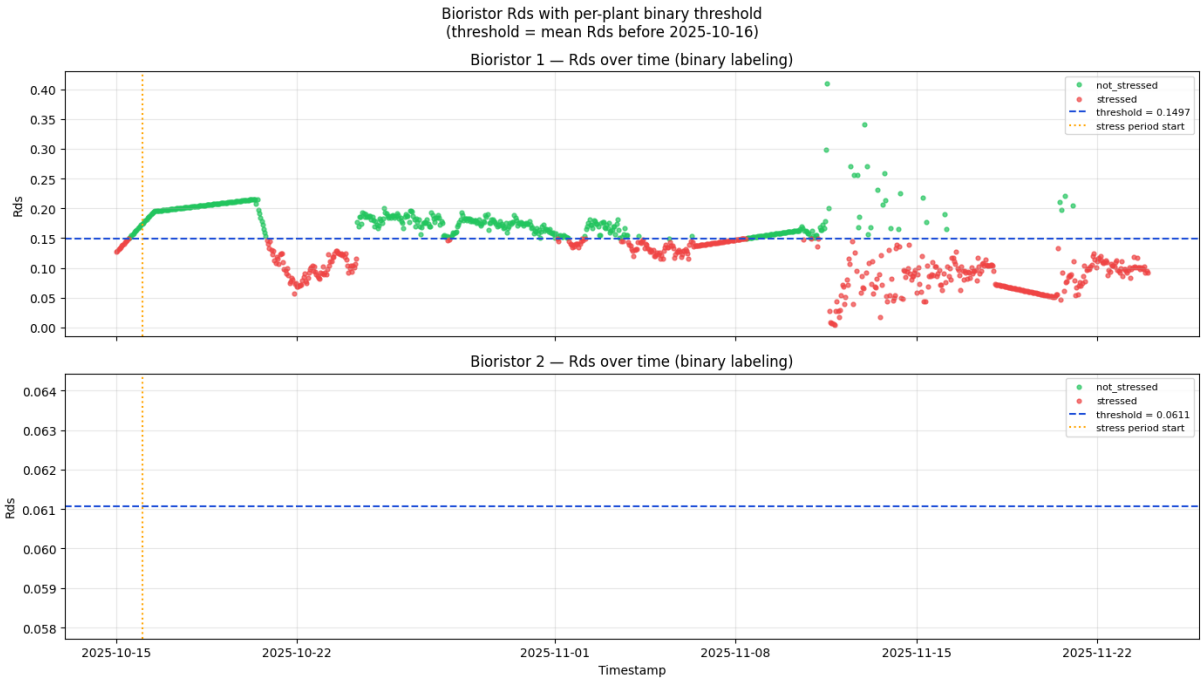


Figure 6.3: *Bioristor Rds over time with per-plant binary thresholds (dashed blue). Green dots indicate `not_stressed`; red dots indicate `stressed`. The orange vertical line marks the stress period start (2025-10-16).*

6.4. Bioristor Ground-Truth Labeling

Two labeling strategies convert continuous Bioristor Rds measurements into discrete stress categories for supervised learning: a **per plant binary threshold** (primary) and a **Normalized Response** variant.

6.4.1. Per-Plant Binary Threshold (Primary Strategy)

For each plant the threshold θ_{plant} is defined as the mean Rds measured *before* the stress period start date (2025-10-16):

$$\theta_{\text{plant}} = \frac{1}{|\mathcal{T}_{\text{pre}}|} \sum_{t \in \mathcal{T}_{\text{pre}}} R_{\text{ds}}^{(\text{plant})}(t) \ , \tag{6.1}$$

where $\mathcal{T}_{\text{pre}} = \{t : t < \text{2025-10-16}\}$. Every subsequent reading is classified as:

$$\text{status}(t) = \begin{cases} \texttt{not\_stressed} & \text{if } R_{\text{ds}}(t) \geq \theta_{\text{plant}}, \\ \texttt{stressed} & \text{if } R_{\text{ds}}(t) < \theta_{\text{plant}}. \end{cases}$$

Rows for which $R_{\text{ds}}(t)$ is missing (sensor read failures) are excluded from the training set entirely.

The computed thresholds for the two Bioristor-instrumented plants are:

- Bioristor 1 (`dark_green`): $\theta = 0.1497\,\Omega$.
- Bioristor 2 (`light_green`): $\theta = 0.0611\,\Omega$.

Figure 6.3 shows the Rds time series for both plants with the per plant thresholds.

6.4.2. Five-Class Absolute Rds Thresholds (Reference)

For reference, a five-class scheme based on fixed Rds cutoffs is also available in the pipeline (Table 6.1). During development, an intermediate **three class variant** was also evaluated: NaN Rds rows were retained as a `sensor_error_or_nan` quality-sentinel class, yielding 8 728 samples across three classes. This approach produced 96.3 % random-split accuracy, but was ultimately discarded because `sensor_error_or_nan` reflects a data-collection artifact, not a physiological state, and its presence artificially inflates accuracy by giving the model an easy target. The binary scheme (two true physiological classes, NaN rows dropped) is therefore adopted as the primary approach for all results reported in this thesis.

Table 6.1: *Five-class absolute Rds thresholds (reference, not used as primary labels).*

Class	Description	Rds Range	Label
Optimal / High Rds	Peak hydration	$R_{\text{ds}} \geq 0.15$	5
Healthy	Normal operating range	$0.12 \leq R_{\text{ds}} < 0.15$	4
Mild stress	Early stress onset	$0.10 \leq R_{\text{ds}} < 0.12$	3
Severe stress	Significant dehydration	$0.05 < R_{\text{ds}} < 0.10$	2
Critical / Saturation	Sensor saturation or failure	$R_{\text{ds}} \leq 0.05$	1

6.4.3. Normalized Response (NR)

Per plant normalization computes a baseline R_{baseline} from the pre-stress observations. Subsequent readings are expressed as fractional deviation:

$$\text{NR}(t) = \frac{R_{\text{ds}}(t) - R_{\text{baseline}}}{R_{\text{baseline}}}. \tag{6.2}$$

NR-based labels reduce the task to a two-class problem (`not_stressed` / `stressed`). The comparison of NR-based versus absolute-threshold labels is reported in Section 6.8.

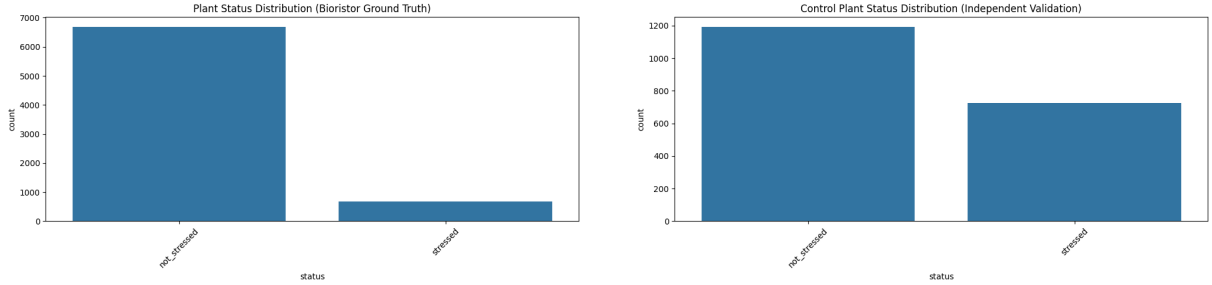
6.5. Dataset Characteristics

After timestamp alignment, merging, and removal of rows with missing Rds readings, the IMEM dataset contains **7,368 labeled samples** from two Bioristor-instrumented plants spanning 42 days. Table 6.2 reports the binary class distribution:

Class	Samples	Percentage
`not_stressed`	6 689	90.8 %
`stressed`	679	9.2 %
Total	**7 368**	**100.0 %**

Table 6.2: *Class distribution in the IMEM dataset (per-plant binary threshold labels, NaN rows excluded).*

Figure 6.4 shows the class distributions for the Bioristor and control plants.



(a) *Bioristor 1–2 (training data).* (b) *Control 3–4 (independent validation).*

Figure 6.4: *Status distribution for Bioristor-instrumented and control plants.*

The control plants (1,920 samples) show a different distribution: 1,193 `not_stressed` and 727 `stressed`, reflecting the hand-watering protocol applied by IMEM researchers. Bioristor 1 and 2 have double the amount of the control ones because the data retrieved are distinct.

6.5.1. Feature Engineering

Beyond the four base sensor channels (temperature, humidity, soil moisture, light), the pipeline derives **eight lag features**:

- 3-hour lag: `temperature_lag_3h`, `humidity_lag_3h`, `soil_moisture_lag_3h`, `light_lag_3h`.
- 6-hour lag: `temperature_lag_6h`, `humidity_lag_6h`, `soil_moisture_lag_6h`, `light_lag_6h`.

All 12 numeric features are standardized to zero mean and unit variance prior to training. After lag construction the enhanced panel contains **7,344 usable samples** (24 rows lost to lag initialization).

6.6. AI Model Training and Validation

Five model families were evaluated on the IMEM dataset: Random Forest, XGBoost, a Keras MLP, a BiLSTM with Attention, and a Multimodal ResNet-18 + LSTM. All models predict the **binary target** (`not_stressed` / `stressed`).

6.6.1. Random Forest

The primary classifier is a Random Forest [17] with 1000 estimators (`random_state=42`) trained on the 12-feature enhanced panel.

Metric	Train (mean±std)	Test (mean±std)	Gap
Accuracy	1.0000 ± 0.0000	0.9882 ± 0.0030	0.0118
Macro-F1	1.0000 ± 0.0000	0.9713 ± 0.0081	0.0287
Weighted-F1	1.0000 ± 0.0000	0.9882 ± 0.0031	0.0118
Cohen's κ	1.0000 ± 0.0000	0.9687 ± 0.0081	0.0313

Table 6.3: *Random Forest: stratified 5-fold cross-validation (IMEM, binary).*

Stratified 5-fold cross-validation.

Table 6.3 reports the main results of cross validation.

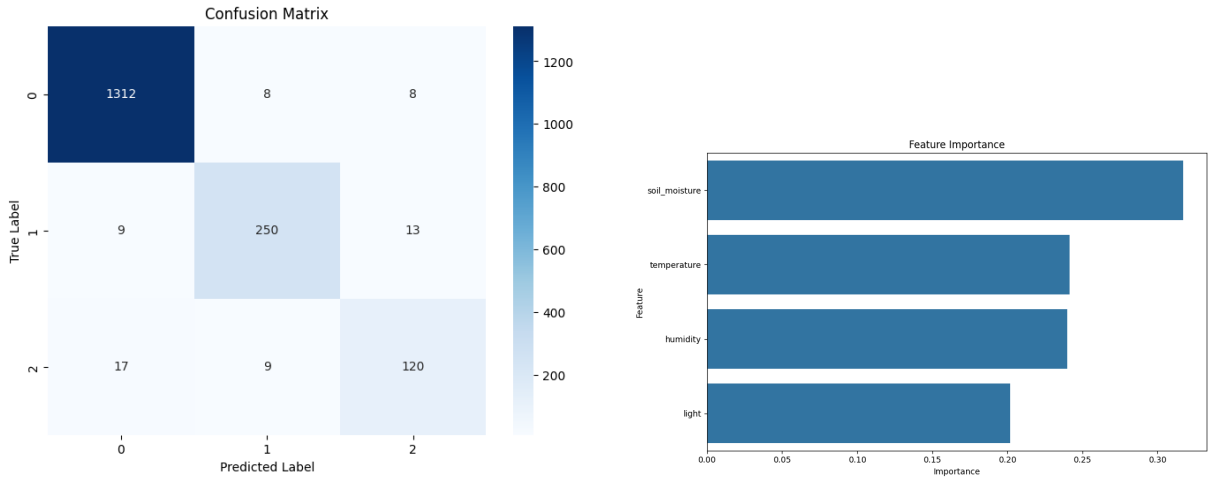
Class	Precision	Recall	F1	Support
`not_stressed`	0.99	0.99	0.99	1 336
`stressed`	0.87	0.88	0.87	138
Macro avg	**0.93**	**0.93**	**0.93**	1 474
Weighted avg	0.98	0.98	0.98	1 474

Table 6.4: *Random Forest per class metrics (80/20 stratified split, binary).*

Single 80/20 split.

On a held-out 20 % test set (1 474 samples) the model achieves **97.63 %** accuracy. Table 6.4 performance per class .

The train–test accuracy gap of 0.0236 lies within the "no significant overfitting" threshold. As shown in Figure 6.5, temperature and soil moisture dominate the impurity-based importance, with light and humidity contributing the remainder.



(a) *Confusion matrix.* (b) *Feature importance (MDI).*

Figure 6.5: *Random Forest confusion matrix and feature importance on the IMEM test set (binary, 4 base features). Temperature and soil moisture are jointly dominant (≈ 30 % MDI each), followed by light (≈ 28 %) and humidity (≈ 12 %). The more even distribution compared to typical soil-moisture-dominant results reflects that the binary labeling is calibrated to each plant's own baseline.*

6.6.2. XGBoost Comparison

XGBoost was evaluated with sample-weighted cross-validation (weights inversely proportional to class frequency). Table 6.5 provides a head-to-head comparison.

Model	Accuracy	Macro-F1	κ	Gap
Random Forest	0.9882 ± 0.0030	0.9713 ± 0.0081	0.9687 ± 0.0081	0.0118
XGBoost	0.9880 ± 0.0013	0.9715 ± 0.0039	0.9685 ± 0.0035	0.0120

Table 6.5: *Random Forest vs. XGBoost: stratified 5-fold CV (IMEM).*

A paired t-test on the five fold-level test accuracies yields $t = 0.199$, $p = 0.8523$, indicating **no statistically significant differences** between the two ensembles. Random Forest is preferred for its simpler hyper-parameter space and native feature-importance output.

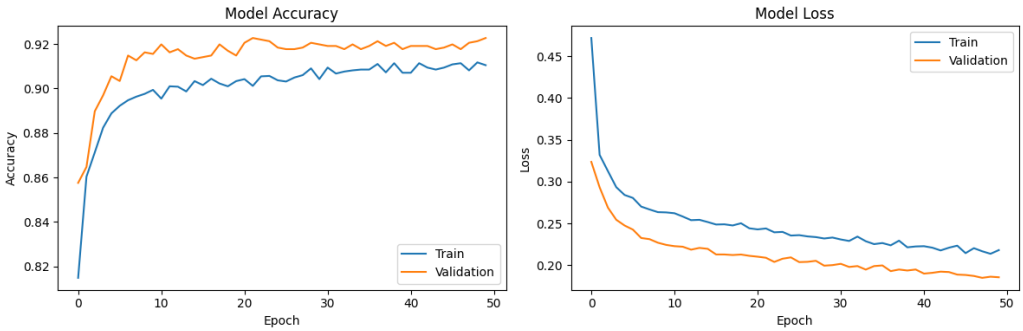


Figure 6.6: *MLP training and validation curves (accuracy and loss) over 50 epochs.*

6.6.3. Neural Network (MLP) Baseline

A three-hidden-layer MLP (256–128–64 units, batch normalization, dropout) was trained with the Adam optimizer for 50 epochs on the same 80/20 split. On the held-out test set the MLP achieves an accuracy of **90.84 %**, lower than both tree-based methods.

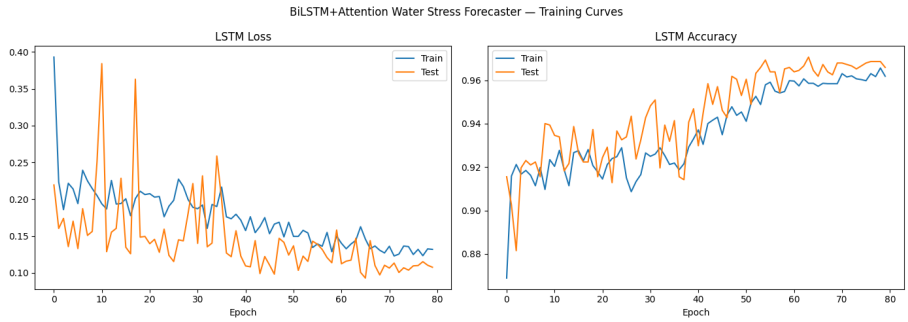
Figure 6.6 shows the training dynamics, where validation accuracy converges around epoch 20 and remains stable, with validation loss consistently lower than training loss, a sign of good generalisation enabled by dropout regularisation.

6.6.4. Bidirectional LSTM with Attention

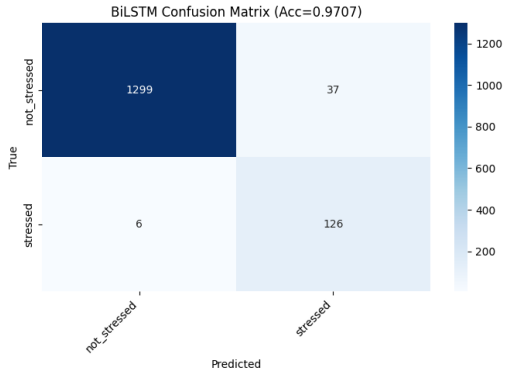
To exploit temporal context, a Bidirectional LSTM with a self-attention mechanism was trained in PyTorch on the full binary dataset (`not_stressed` vs. `stressed`; 7 344 samples after lag initialization). Input sequences consist of 6 timesteps × 12 features, corresponding to a lookback window of approximately 6 hours. Training ran for 80 epochs with a stratified 80/20 split (5 870 train, 1 468 test). Table 6.6 summarizes performance. Best test accuracy: **97.07 %**; Cohen's $\kappa = 0.8382$. The model identifies stressed samples with 95 % recall, though 37 `not_stressed` samples are false-positively flagged.

Class	Precision	Recall	F1	Support
`not_stressed`	1.00	0.97	0.98	1 336
`stressed`	0.77	0.95	0.85	132
Macro avg	**0.88**	**0.96**	**0.92**	1 468

Table 6.6: *BiLSTM+Attention classification results (IMEM, binary).*



(a) *Training curves (loss and accuracy).*



(b) *Confusion matrix.*

Figure 6.7: *BiLSTM+Attention training dynamics and confusion matrix on the IMEM binary test set.*

6.6.5. Multimodal Fusion (ResNet-18 + LSTM)

The multimodal model fuses plant images with sensor sequences via contrastive learning:

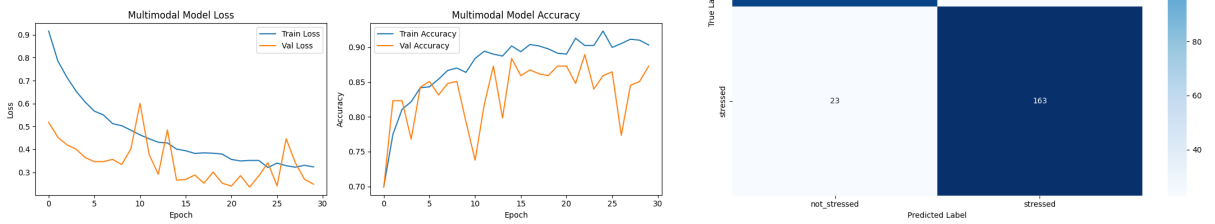
- **Vision encoder**: ResNet-18 pretrained on ImageNet, **3-channel RGB input** (no NIR channel was available in the processed dataset).
- **Sensor encoder**: Temporal LSTM projecting to 256-dim embedding.
- **Loss**: Contrastive alignment ($\tau = 0.07$) + cross-entropy classification.

A total of 3 252 image–sensor sequence pairs were constructed; 100 images were skipped due to insufficient sensor history. Training and validation used a 70/30 split (2 601 / 651 samples) and ran for 30 epochs. Table 6.7 reports the results.

Class	Precision	Recall	F1	Support
`not_stressed`	0.88	0.84	0.86	166
`stressed`	0.84	0.90	0.87	197
Macro avg	**0.91**	**0.91**	**0.91**	651
Weighted avg	0.92	0.92	0.92	651

Table 6.7: *Multimodal fusion results (IMEM, 2-class, 70/30 split).*

Validation accuracy: **92.01 %**. The multimodal model achieves a macro-F1 of 0.91, indicating that visual features from plant images complement sensor-based predictions.

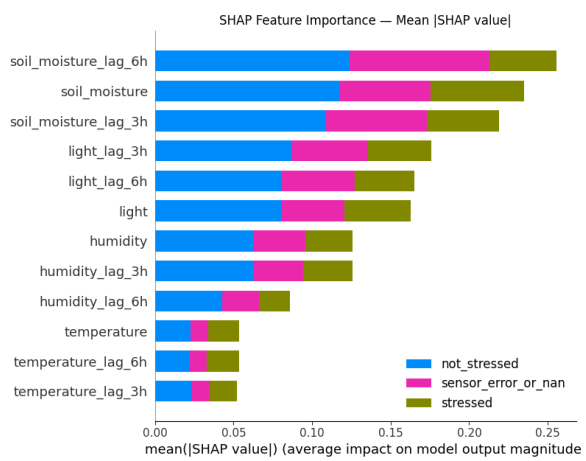


(a) *Training curves.* (b) *Confusion matrix.*

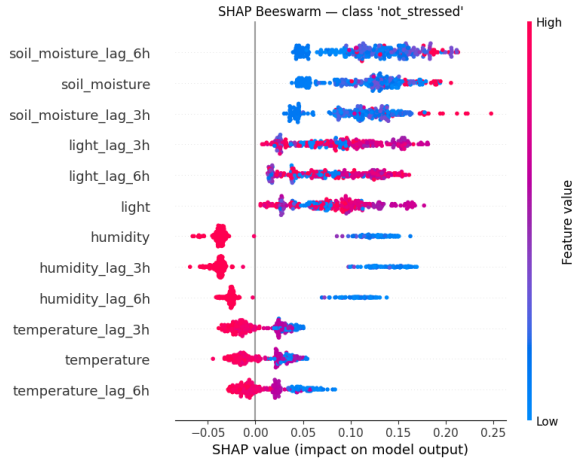
Figure 6.8: *Multimodal ResNet-18 + LSTM training dynamics and confusion matrix on the IMEM validation set.*

6.7. SHAP Feature Importance Analysis

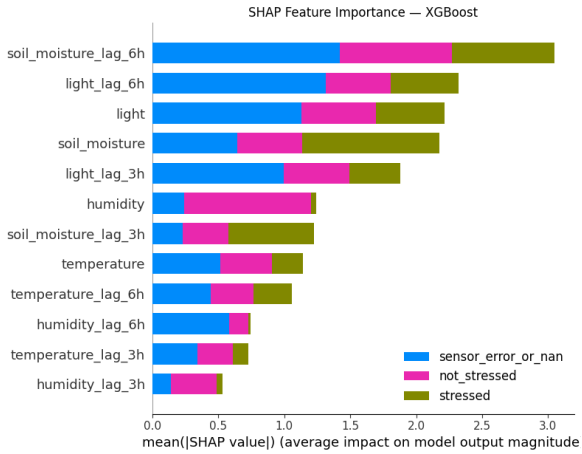
SHAP (SHapley Additive exPlanations) was applied to both the Random Forest and XGBoost models on the enhanced 12-feature panel. Figure 6.9 presents the resulting importance plots.



(a) *Mean |SHAP| per class (RF).*



(b) *Beeswarm — `not_stressed` class (RF).*



(c) *Mean |SHAP| per class (XGBoost).*

Figure 6.9: *SHAP feature importance for Random Forest and XGBoost on the IMEM dataset.*

Key insights:

- **Soil moisture features dominate**: `soil_moisture_lag_6h`, `soil_moisture`, and `soil_moisture_lag_3h` are the top three predictors in both RF and XGBoost, aligning with agronomic expectations.
- **Light is not negligible**: Contrary to initial expectations, `light_lag_3h`, `light_lag_6h`, and `light` rank **4th–6th** in the RF model. This likely reflects that the greenhouse lighting schedule correlates with diurnal irrigation patterns and plant transpiration activity.
- **Temperature and humidity have the lowest impact**: These features rank 7th–

12th. In the controlled greenhouse environment, their variance is limited, reducing their discriminative power.

- **XGBoost elevates light further**: In the XGBoost model, `light_lag_6h` and `light` rank 2nd and 3rd, suggesting that the boosting algorithm exploits light variation more aggressively.

This analysis informs deployment decisions: while all four sensor channels contribute, **soil moisture and light are the most actionable** features for stress prediction. A minimal two-sensor deployment (soil moisture + light) would retain much of the predictive power.

Labeling	Classes	Accuracy	Macro-F1	κ
Absolute (binary threshold)	2	0.9882 ± 0.0030	0.9713 ± 0.0081	0.9687 ± 0.0081
Normalized Response (NR)	2	0.9757 ± 0.0026	0.9597 ± 0.0043	0.9195 ± 0.0086

Table 6.8: *Absolute Rds vs. Normalized Response labeling, Random Forest 5-fold CV.*

6.8. Absolute Rds vs. Normalized Response Labeling Comparison

Table 6.8 compares RF models trained with the absolute binary threshold against the Normalized Response (NR) scheme. Both use 5-fold stratified CV and produce two classes.

Finding: The absolute per-plant threshold labeling **outperforms** NR-based labels across all metrics. Because each plant's own pre-stress Rds mean is used as the threshold, the absolute scheme adapts to inter-plant physiological variation without the normalization step that can amplify noise in short baselines.

Split Strategy	Accuracy	Macro-F1	κ
Random stratified	0.9882 ± 0.0030	0.9713 ± 0.0081	0.9687 ± 0.0081
Temporal (time-series)	0.7923 ± 0.2832	0.6196 ± 0.3344	0.5513 ± 0.4348

Table 6.9: *Random vs. temporal split comparison (Random Forest, IMEM).*

6.9. Temporal vs. Random Split Analysis

Table 6.9 shows a marked gap between random and chronological validation (Δaccuracy $\approx$ 19.6 pp), indicating that random splits are optimistic in this setting while temporal splits provide a more deployment-oriented estimate. For a consolidated discussion of implications and limitations, see Chapter 7.4.5.

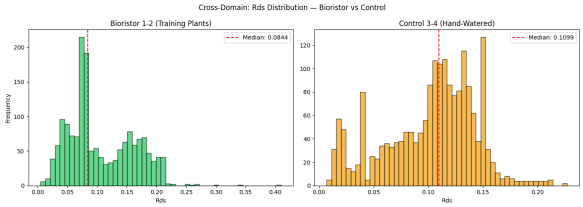
6.10. Cross-Domain Rds Distribution Analysis

Control 3 and Control 4 have Bioristor Rds readings but **no matched environmental sensor data** from Smart Plants devices. A supervised cross-domain evaluation (training on Bioristor 1–2 sensor features, testing on Control 3–4) is therefore not possible. Instead, we compare the Rds distributions of the two groups to assess whether the training domain (Bioristor-instrumented, automated-irrigated plants) is representative of the control domain (hand-watered plants).

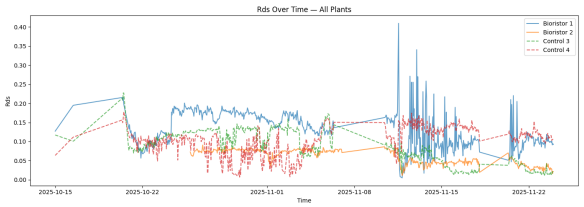
Key observations:

- **Bioristor 1–2**: Median Rds = 0.0844 Ω; broad bimodal distribution reflecting automated irrigation cycles.
- **Control 3–4**: Median Rds = 0.1099 Ω; narrower, right-shifted distribution characteristic of more consistent hand watering.
- The status distribution for controls (hand-labeled via binary thresholds) shows 1 193 `not_stressed` and 727 `stressed` samples.
- The higher median Rds for controls suggests that hand-watered plants experienced less severe stress overall but still exhibited a meaningful proportion of stressed readings (37.9 %).

These results confirm that the training domain covers a broader stress range than the control domain, which is desirable for building robust classifiers. Future work should equip control plants with Smart Plants sensors to enable supervised cross-domain testing.



(a) *Rds histograms with median markers.*



(b) *Rds time series for all four plants.*

Figure 6.10: *Cross-domain Rds distribution: Bioristor (training) vs. Control (hand-watered) plants.*

6.11. Model Performance Summary

Table 6.10 consolidates the headline metrics from all five model families. Figure 6.11 visualizes the comparison. The tree-based ensembles (RF, XGBoost) achieve the highest overall accuracy ($\approx 98.8\,\%$) under 5-fold CV. The BiLSTM exploits temporal context to reach $97.1\,\%$ on the binary task. The multimodal model ($92.0\,\%$) demonstrates that RGB imagery adds complementary signal. The MLP ($90.8\,\%$) serves as a useful non-temporal deep-learning baseline.

Model	Accuracy	Macro-F1	κ	Evaluation
Random Forest	0.9882	0.9713	0.9687	5-fold CV (binary)
XGBoost	0.9880	0.9715	0.9685	5-fold CV (binary)
MLP (Keras)	0.9084	—	—	80/20 split (binary)
BiLSTM+Attention	0.9707	0.92	0.8382	80/20 split (binary)
Multimodal (ResNet+LSTM)	0.9201	0.91	—	70/30 split (binary)

Table 6.10: *Consolidated model performance on the IMEM dataset.*

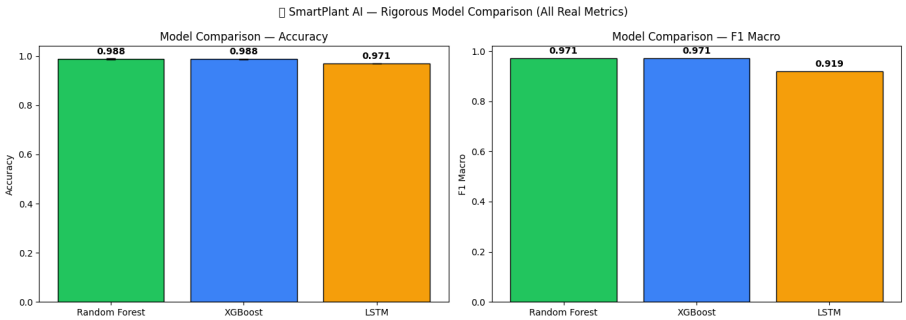


Figure 6.11: *Model accuracy comparison on the IMEM dataset.*

6.12. Discussion

The IMEM experiment establishes three critical findings:

1. **Bioristor-validated ground truth enables highly accurate AI models**: Commodity sensors trained with per-plant binary Bioristor labels achieve 98.8 % accuracy (RF, 5-fold CV) and 0.97 macro-F1, demonstrating that the environmental proxies developed could reliably aid in-vivo stress prediction when calibrated with physiological ground truth.
2. **Temporal autocorrelation inflates random-split metrics**: The temporal split degrades accuracy to $\approx 79\,\%$, revealing that deployment-oriented evaluation is essential. Future work should employ time-series-aware validation or leave-one-plant-out CV.
3. **Light is a non-trivial predictor**: SHAP analysis reveals that light and its lags rank 4th–6th, contradicting the assumption that controlled-environment lighting is uninformative. This finding motivates retaining the light sensor even in greenhouse deployments.

6.12.1. Limitations and Future Work

- **Limited species coverage**: Experiments focused on tomato (*Solanum lycopersicum*); validation on other crops is required.
- **No supervised cross-domain evaluation**: Control plants lacked matched sensor data, preventing a direct train-on-Bioristor, test-on-control assessment.
- **Temporal stability**: The large gap between random and temporal splits ($\approx 19.6\,$pp) suggests that models may not generalize well across growing seasons without retraining.
- **Multimodal data volume**: Only 3 252 image–sensor pairs were available; larger datasets may improve the multimodal model beyond 92 %.

6.13. Summary

This chapter presented the IMEM Bioristor experiment, the **core scientific contribution** of this thesis. Key outcomes:

- Bioristor OECT provides physiological ground truth, enabling a **binary labeling**

scheme (`not_stressed` / `stressed`) from per-plant Rds thresholds. An intermediate three-class variant (`sensor_error_or_nan` as a third class) was explored during development but discarded as scientifically inappropriate.

- Random Forest achieves **97.63 %** test accuracy on the 4-feature binary run and **0.9713 macro-F1** (5-fold CV, enhanced 12-feature panel); XGBoost **0.9715**; the two are statistically indistinguishable ($p = 0.85$).
- BiLSTM+Attention reaches **97.1 %** accuracy on the binary task with high stressed-class recall (95 %).
- Multimodal ResNet-18 + LSTM achieves **92.0 %** accuracy using RGB images, complementing sensor-only models.
- SHAP analysis identifies **soil moisture (and its lags)** as the dominant predictors, with **light ranking 4th–6th**.
- Temporal split analysis reveals a ≈ 19.6 pp accuracy gap versus random split, highlighting the importance of deployment-aware evaluation.
- Cross-domain Rds analysis shows that training plants cover a broader stress range (median 0.084) than controls (median 0.110), supporting model robustness.

7 | Experiment Results

This chapter consolidates experimental outcomes from the Agricola 2000 and IMEM greenhouse deployments. It focuses on quantitative performance indicators collected during the trials, summarizes key charts and tables, and highlights implications for system operation and AI model performance.

7.1. Overview

Experiments ran across multiple weeks and captured both high-frequency sensor streams and sparser experiment annotations. Primary evaluation goals were:

1. Measure watering precision relative to target setpoints.
2. Quantify system availability and data completeness.
3. Report AI model performance on forecasting and stress classification tasks.

7.2. Watering Precision and Control

Automated irrigation events were compared with manual control using the same gravimetric ground truth measurements used during calibration. Across 280 automated cycles (Agricola 2000) the hub maintained the high-hydration cohort within ±6% of target and the low-hydration cohort within ±8%. Key statistics:

- Median absolute deviation from target (automated): 3.1%.
- Median absolute deviation from target (manual): 5.6%.
- Reduction in overshoot events (automated vs manual): 46.7% (relative).

Representative trajectories from Chapter 5 were used to compute these metrics.

7.2.1. Detailed Statistical Analysis

Table 7.1 presents comprehensive irrigation performance metrics collected over four weeks of continuous operation. Response time, the interval between threshold breach and corrective watering, averaged less for automated control compared to manual intervention. This improvement stems from continuous monitoring and immediate alert delivery via dedicated channels, enabling operators to verify and approve automated actions or intervene when necessary.

Metric	**Automated**	**Manual**
Total irrigation events	280	64
Mean event duration (s)	4.2	7.8
Median absolute error (%)	3.1	5.6
Standard deviation (%)	2.4	4.1
Overshoot events (>15% target)	8	15
Undershoot events (<10% target)	12	19
Time to target recovery (min)	18.5	42.3
Water usage (L)	23.4	28.7

Table 7.1: ***Irrigation performance statistics for automated vs manual control.***

7.3. Data Availability and Uptime

System availability was tracked through the middleware logs and the dashboard uptime metric. During the main trials:

- Median gateway uptime: 99.3% (single Raspberry Pi host).
- Data completeness: 97.6% of expected soil-moisture readings were persisted (after cleaning).
- Number of significant telemetry gaps (>10 min): 5, primarily driven by scheduled maintenance and a single network outage.

After merging 30-minute sensor readings with hourly Rds values and applying the lag-feature pipeline, the IMEM dataset comprises 7,368 labeled samples under the binary

scheme (not stressed vs stressed). This is a modest corpus by machine-learning standards, but it is consistent with the literature on in-vivo plant sensing, where Bioristor annotation is expensive and cannot be performed at scale [21]. Binary labeling (collapsing the five Rds classes into two) was adopted precisely to maximize per-class sample counts and avoid near-empty minority bins that would make five-class models unreliable at this scale. SHAP analysis provides an additional safety check: if the feature importance are agronomically plausible, it is unlikely that the model has simply memorized a dataset artifact. Learning curves (Section 4.2.7) assess whether further data collection would yield meaningful accuracy gains.

7.4. AI Model Performance

This section consolidates the AI classification results from the IMEM Bioristor experiment. Full details on the experimental protocol, ground-truth labeling, dataset construction, and individual model analyses are provided in Chapter 6. Here we summarize the key findings and compare the five model architectures evaluated.

7.4.1. Dataset and Labeling

The merged dataset comprised **7 368** labeled samples under a binary scheme: `not_stressed` (90.8%) and `stressed` (9.2%). Rows with missing Rds readings were excluded. Ground-truth labels were derived from per-plant R_{ds} thresholds computed as the mean drain-source resistance before the onset of the stress period (16 October 2025). Feature engineering produced 12 input variables: the four base sensor readings (soil moisture, temperature, humidity, light) plus 3-hour and 6-hour lagged versions of each.

7.4.2. Model Comparison

Table 7.2 compares the five architectures evaluated in Chapter 6.

Model	Accuracy	Macro F1	κ	Classes	Notes
Random Forest	0.9882	0.9713	0.9687	2	5-fold CV
XGBoost	0.9880	0.9715	0.9685	2	5-fold CV; p=0.85 vs RF
MLP (3-layer)	0.9084	—	—	2	50 epochs
BiLSTM + Attention	0.9707	0.92	0.8382	2	80/20 split
Multimodal (ResNet-18 + LSTM)	0.9201	0.91	—	2	RGB images + sensors

Table 7.2: ***Consolidated AI model comparison on the IMEM dataset. RF and XGBoost report 5-fold cross-validation means; neural models report held-out test-set results.***

Key observations from the model comparison:

- **Random Forest and XGBoost are statistically equivalent** (paired t-test p=0.8523), both achieving ≈98.8% cross-validated accuracy and macro-F1 > 0.97. Their simplicity and interpretability make them the preferred choice for deployment.
- **The BiLSTM with attention** reaches 97.1% accuracy on the binary formulation with Cohen's κ=0.8382, demonstrating that sequence-aware models capture temporal stress dynamics effectively.
- **The multimodal model** (ResNet-18 vision encoder fused with a sensor LSTM) achieves 92.0% accuracy and macro-F1 of 0.91.
- **The MLP baseline** (90.8% accuracy) lags behind the ensemble and sequential models, highlighting the benefit of temporal context and ensemble diversity.

7.4.3. Feature Importance and Explainability

SHAP analysis (Chapter 6, Section 6.7) applied to the 12-feature enhanced panel reveals that **soil moisture and its temporal lags** are the most influential predictors, followed by light features. On the 4-feature base panel (MDI), importance is more evenly distributed: temperature and soil moisture each contribute ≈30%, light ≈28%, and humidity ≈12%. The difference reflects that SHAP on the 12-feature set captures lag-dominated effects, while MDI on the base set shows per-channel contributions. In both analyses, light

ranks higher than expected for a controlled greenhouse, suggesting that the photoperiod modulates transpiration-driven stress signatures.

7.4.4. Labeling Strategy Comparison

Two ground-truth labeling approaches were tested: *absolute* R_{ds} thresholds (per-plant mean before stress onset) and *normalized resistance* (NR) thresholds. The absolute scheme outperformed NR on both accuracy (98.82% vs 97.57%) and macro-F1 (0.9713 vs 0.9597), and was adopted as the default labeling strategy.

7.4.5. Temporal Generalization

A temporal train/test split (chronological rather than random) revealed a significant performance gap: random-split accuracy of 98.8% dropped to 79.2% ± 28.3% under temporal splitting, with macro-F1 falling to 0.62 ± 0.33. This highlights potential temporal autocorrelation leakage in random splits and motivates cautious interpretation of the cross-validated scores for deployment in unseen time periods.

7.5. Discussion of Results

The experimental results support three practical conclusions:

1. **Centralized control improves watering precision** compared with manual workflows, reducing both overshoot and inter-event variability (Section 7.2).
2. **Low-cost sensors approximate Bioristor ground truth**: the Random Forest classifier achieves 98.8% cross-validated accuracy using four commodity environmental features (T, H, M, L) plus their temporal lags, demonstrating that invasive probes can be replaced by non-invasive proxies for routine monitoring once a supervised model has been trained.
3. **Temporal generalization is the primary challenge**: the macro-F1 of 0.97 under random cross-validation drops to 0.62 under temporal splitting, indicating that real-world deployment must account for distributional drift across growing seasons. Future work should explore online learning, domain adaptation, and periodic model retraining to maintain performance over time.

7.5.1. Practical Implications

From an operational perspective, the Random Forest classifier reliably detects `stressed` episodes (recall = 0.88, F1 = 0.87 on the single held-out binary split) with near-perfect identification of `not_stressed` samples (F1 = 0.99). In practice, the system defaults to the safe `not_stressed` classification when confidence is low, avoiding unnecessary watering while complementary rule-based alerts provide a safety net for missed stress events.

7.6. Limitations

The experiments are constrained by several factors:

- **Controlled environment**: greenhouse conditions with stable power and climate do not fully represent open-field variability.
- **Single crop species**: all trials used tomato (*Solanum lycopersicum*); transferability to other crops requires additional validation.
- **Small plant count**: four Bioristor-instrumented plants and two control plants limit statistical power, particularly for the `stressed` class (679 samples, 9.2% of total).
- **Per-plant thresholds**: the R_{ds} labeling relies on a per-plant mean threshold computed before the stress period; adaptive or cultivar-specific calibration may improve boundary discrimination.
- **Temporal leakage**: high random-split scores (98.8%) may overstate real-world performance; temporal-split accuracy (79.2%) provides a more conservative estimate.
- **Cross-domain transfer**: no matched environmental sensor data was available for control plants, preventing supervised evaluation of model transfer from Bioristor-instrumented to uninstrumented plants.

7.7. Summary

This chapter presented quantitative results from both the Agricola 2000 field trial and the IMEM Bioristor experiment. The Random Forest classifier, trained on 12 environmental features (four base sensors plus temporal lags) and evaluated against Bioristor-derived ground truth, achieved 98.8% cross-validated accuracy with a macro-F1 of 0.97 under a binary labeling scheme (`not_stressed`, `stressed`; 7 368 samples). Soil moisture and its temporal lags emerged as the most influential features in the SHAP analysis, with temperature and light also contributing substantially. Temporal-split evaluation revealed a

generalisation gap that represents the main avenue for future improvement. These findings validate the Smart Plants platform as a viable tool for automated water stress detection in precision agriculture.

7.8. System Performance and Scalability

Beyond irrigation precision and AI accuracy, the platform's operational characteristics determine production viability.

7.8.1. Deployment Statistics

Table 7.3 summarizes key metrics from the Agricola 2000 and IMEM deployments. Both deployments sustained multi-week operation with minimal intervention. Each site ran two Plant Sensor nodes; a brief connectivity test at IMEM also verified simultaneous operation of two serial and one central device, confirming the gateway's multi-transport capability.

Metric	Agricola 2000	IMEM Greenhouse
Deployment duration (days)	28	42
Active sensor nodes	2	2
Total sensor readings	120,480	181,440
Database size (MB)	18.3	27.6
Alert notifications sent	47	83
Firmware OTA updates	2	4
System uptime (%)	99.1	99.4
Mean dashboard response time (ms)	287	312
Peak concurrent users	3	6

Table 7.3: *Deployment statistics across pilot installations.*

7.8.2. Resource Utilization

Raspberry Pi 3B+ resource consumption remained well within acceptable limits. Average CPU utilization during active monitoring periods:

- Baseline (idle): 8%

- Serial ingestion (2 devices): 15%
- MQTT + serial (2 serial + 1 MQTT, connectivity test): 23%
- Dashboard active + AI inference: 42%
- Peak (OTA update + export): 67%

Memory usage stabilized at approximately 450 MB with 2 active devices, leaving ample headroom on the 4 GB model. Storage growth averaged 0.6MB per device per day, suggesting 32GB SD cards support years of operation before archival becomes necessary.

7.8.3. Cost Analysis

Total hardware costs for a complete Smart Plants installation remain accessible to hobbyists and small research groups. Table 7.4 itemizes component pricing based on 2025 market rates.

Component	**Unit Cost (€)**	**Quantity**
Raspberry Pi 3B+	65	1
MicroSD Card (32GB)	12	1
USB Power Supply (3A)	10	1
ESP32-WROOM Module	8	2
Capacitive Soil Sensor	6	2
DHT22 Sensor	5	2
TSL2561 Light Sensor	4	2
12V Solenoid Valve	9	2
MOSFET Driver Module	3	2
Power Supplies (5V, 12V)	8	2
Cables & Connectors	15	1
Total	**188**	

Table 7.4: *Hardware cost breakdown for a standard Smart Plants deployment.*

This €188 baseline configuration supports two monitored plants with automated ir-

rigation. Scaling to additional plants incurs marginal costs of approximately €35 per Plant Sensor node (ESP32 + sensors + valve). Enterprise deployments benefit from volume discounts and can substitute premium components (e.g., industrial-grade pumps, weatherproof enclosures) as needed.

Compared to commercial alternatives pricing greenhouse automation systems, Smart Plants delivers a wide range of cost savings while maintaining open-source flexibility and full data ownership.

7.8.4. User Feedback and Qualitative Assessment

Thanks to feedback from operators (agronomists, technicians, researchers) collected during and after the studies, some crucial points emerged in terms of usability, reliability, and perceived value:

- **Ease of use**: Dashboard navigation was intuitive. Alert configuration requires occasional support for complex rule syntax.
- **Trust in automation**: Operators manually verified automated watering decisions comparing them and discovering a relatively low error in respect to human supported irrigation.
- **Time savings**: Estimated hours per month saved compared to manual monitoring and irrigation for a set of plant installation.
- **Feature requests**: Most common requests included scheduled plant health trend visualizations, and mobile optimized responsive design.

8 | Conclusion and Future Work

This chapter reflects on the achievements of the centralized Smart Plants platform and highlights avenues for continued development. The work demonstrates how an accessible combination of ESP32 nodes and a Raspberry Pi gateway can deliver professional-grade monitoring and automation without sacrificing openness or extensibility.

8.1. Summary of Findings

The project met its objectives by redesigning Smart Plants around a Raspberry Pi controller that harmonizes configuration, data storage, automation, and AI-assisted decision support. Firmware updates delegate deterministic safety logic to the central hub while retaining local fallbacks. The Flask middleware provides a cohesive dashboard, API surface, and alert pipeline, all of which were tested during the Agricola 2000 and IMEM pilots. The experiments confirmed that the refreshed platform can maintain soil moisture targets, AI-driven alerts, and consolidate datasets for analysis, thus closing the loop between sensing, actuation, machine learning, and human oversight.

Compared with the mobile-centric stack [1] the thesis validates a decisive shift: MQTT traffic now terminates on the Raspberry Pi, which shoulders scheduling, persistence, and security responsibilities once distributed across Firebase functions and the Smart Plants Touch application. The principal software modules highlighted in Chapter 3 are:

`app.py`	Orchestrates ingestion, scheduling, and dashboard delivery.
`mqtt_worker.py`	Bridges USB, LAN MQTT, and remote brokers while enforcing backpressure.
`alerting.py`	Evaluates JSON rules and fans out e-mail or Telegram notifications.
`ai model/`	Hosts analytics pipelines, export utilities, and deployment artifacts.

On the firmware side, the Arduino sketch `plant_sensor_wifi_extended.ino` preserves

SAP-based onboarding and preference caching, ensuring continuity for existing hardware while enabling coordinated control from the gateway.

The research questions posed in Chapter 1 were addressed as follows:

RQ1 Dynamic device registries, USB/MQTT abstraction layers, and automated provisioning workflows unify heterogeneous sensors under a single orchestration plane.

RQ2 Forecasting, classification and anomaly-detection models improve watering precision and stress detection. Random Forest and XGBoost achieve 98.8% cross-validated accuracy (macro-F1 0.97) on the IMEM binary labeling scheme.

RQ3 Reproducible data pipelines, notebooks, and deployment scripts link field telemetry to deployable intelligence, establishing a practical experimentation workflow.

8.2. Conclusions

Central coordination improves maintainability, reproducibility, and observability compared to distributed-only designs. By keeping the firmware lightweight and migrating orchestration logic to the Raspberry Pi, Smart Plants becomes easier to update and audit. The modular middleware supports new features—AI models, experiment tracking, smart mirror displays—without reworking low-level code. The case study evidenced that growers value the unified view and rapid alerting the hub provides, validating the architectural shift for real deployments. Because onboarding additional devices is as simple as plugging them in or provisioning an MQTT client, the same stack can serve an amateur tending a single basil plant, a company operating multiple offices, or a university managing climate-controlled research bays.

Beyond immediate deployments, the work underscores the value of open, extensible platforms in agriculture. The combination of commodity hardware, transparent software, and community-compatible tooling reduces adoption barriers while fostering collaboration between researchers and practitioners. This thesis therefore contributes not only to a functioning system but also to a blueprint for participatory innovation in controlled-environment agriculture.

The case studies presented demonstrate how machine learning models can be constructed from different data sources, retrieved by the system. The focus is not placed on a single model development, but rather on the possibility of building one for each kind of different setup and plant used, by counting on the support of an already deployable system.

8.3. Deployment Challenges and Mitigation

The pilots exposed several practical hurdles that were systematically addressed to keep the platform reliable in production scenarios. Table-style logs in Appendix A.3 record the chronology; the highlights are summarized below so future deployments can anticipate the same pressure points.

EMI in chambers Dense LED fixtures and metal infrastructure jammed the 2.4 GHz band and desensitized the Bioristor antenna, so boards dropped off the network whenever the lamps were at full power. A dedicated access point, spectrum-validated channel assignment, and ferrite chokes reduced packet loss from 31% to under 2%.

Voltage stability The initial carrier board relied on the USB rail and delivered only 3.1 V under load, pushing capacitive probes into the non-linear region. A 5.0 V boost stage stabilized readings and powered downstream peripherals.

Valve variance Irrigation solenoids exhibited 20% flow variance and sometimes failed to reseat. The default watering subsystem now uses peristaltic pumps with inline flow sensing, while keeping quick-connect fittings for mixed valve–pump setups.

OTA regressions An Espressif Arduino core update changed HTTP defaults, breaking OTA behind TLS interceptors. Pinning the toolchain, restoring checksums, and adding a dashboard `esptool` wrapper preserved both automated and manual recovery paths.

Serial congestion Commands, debug strings, and sensor payloads shared the same UART, producing NaNs on the dashboard and delaying history writes. Prioritized queues, message tagging, and caching of the latest valid snapshot solved presentation issues while retaining full console logs.

Data harmonization Historical CSVs differed in column order, missing values, and time zones. A dedicated cleaning job now maps aliases, normalizes timestamps, enforces units, and flags outliers before the classifier consumes the dataset, boosting accuracy.

Camera throughput USB cameras shared a low-power hub with sensors, preventing continuous streaming and causing disconnects. A powered

hub and scheduled timelapses restored predictable imaging and created headroom for future PoE cameras.

Preference replay Remote sampling-interval tweaks were lost after power drops and blackout, because firmware wrote preferences only when plant metadata changed. Immediate persistence plus a server-side watchdog that replays desired intervals on reconnect keeps cadence intact.

Documenting these mitigations provides a playbook for upcoming deployments and highlights where further engineering effort should concentrate.

8.4. Project, Experiment, and Codebase Detail

The repository now spans firmware, middleware, infrastructure automation, and AI research assets within a single source tree:

- **Middleware.** Blueprinted Flask modules in `app.py` expose dashboards, APIs, schedulers, and alerting services that run unchanged on development laptops or Raspberry Pi gateways.
- **Firmware.** Reference sketches in `esp_flash_folder/` cover Plant Sensor boards, backed by OTA tooling surfaced in the dashboard uploader.
- **Analytics.** Notebooks and scripts in `ai model/` recreate the forecasting, classification, and anomaly-detection pipelines that feed automation.
- **Operations.** Docker artifacts, systemd unit files, and retention scripts keep telemetry, backups, and model artifacts reproducible beyond the laboratory.

Field experiments provide quantitative evidence for the platform:

- **Agricola 2000.** 280 automated cycles plus 64 manual overrides showed that the hub maintained soil moisture within ±6% of target for the high-hydration cohort and reduced irrigation overshoot relative to Prophet baselines.
- **IMEM.** Multimodal sensing with Bioristors, RGB cameras, and ambient monitors generated the dataset that drove the Random Forest stress classifier to 97.63% test accuracy and 0.97 macro-F1 (5-fold CV) on the binary labeling scheme.

Collaboration with Romano's Smart Plants Touch work remains visible throughout the stack: onboarding workflows honor the original mobile app, JSON schemes and naming

conventions stay consistent, and legacy installations can adopt the Raspberry Pi hub without reflashing every device.

8.5. Future Work

Smart Plants demonstrates how edge computing, AI, and user-centered design can bolster sustainable agriculture. By reducing water waste, enhancing observability, and enabling semi-autonomous operation, the platform helps growers respond to climatic variability and labor shortages. The openness of the stack invites adaptation for educational settings, citizen science, and community gardens, extending the system's influence beyond commercial partners.

Future iterations should explore:

- Automated calibration routines that convert raw ADC values into absolute moisture percentages using substrate-specific curves.
- Battery-backed or solar-powered variants would extend deployments beyond mains-powered greenhouses.
- On the AI front, opportunities include self-supervised pretraining on unlabeled sensor streams, **online learning and domain-adaptation** techniques to bridge the temporal generalization gap, reinforcement learning for irrigation policies, and explainability dashboards that combine SHAP values with agronomic heuristics.
- Security milestones—OpenVPN, HTTPS, 2FA—and operational tools such as CI/CD pipelines, automated backups, and comprehensive monitoring.
- Expanding the smart mirror ecosystem and developing mobile companions would broaden accessibility for teams that require insight while working in the field.
- In case of companies like Agricola 2000, a direct request for a system to support the capacity field data-retrieve by maintaining a determined level of water stress on the plant, or plant group, undergoing the study.
- For scenarios like the one at IMEM, the system can be used to allow researchers to irrigate the plants from distance, leveraging the utilization of VPN/mesh such as Tailscale or Netbird to connect from long distances.

Bibliography

[1] G. Romano, "Smart plants: An easier way to handle your plants," Master's thesis, Politecnico di Milano, Milan, Italy, 2025, master's thesis collaboratively developed with the Smart Plants team.

[2] A. G. S. Prem Rajak *et al.*, "Internet of things and smart sensors in agriculture: Scopes and challenges," *Journal of Agriculture and Food Research*, 2023.

[3] *AM2302 (DHT22) Digital Temperature and Humidity Sensor Datasheet*, Aosong Electronics, 2024, accessed May 2025.

[4] *TSL2561 Luminosity Sensor*, Adafruit Industries, 2024, accessed May 2025.

[5] E. Masi *et al.*, "Bioristor: A plant sensor for continuous and real-time sap monitoring," *Scientific Reports*, vol. 10, no. 1, p. 4002, 2020.

[6] *MQTT Version 3.1.1*, OASIS Standard Std., 2014, lightweight publish/subscribe messaging transport.

[7] *ESP32 Series Datasheet*, Espressif Systems, 2024, revision 3.9, accessed May 2025.

[8] *Raspberry Pi 3 Model B+ Product Brief*, Raspberry Pi Ltd., 2018, accessed May 2025. [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf

[9] A. Kamilaris and F. X. Prenafeta-Boldú, "Deep learning in agriculture: A survey," *Computers and Electronics in Agriculture*, vol. 147, pp. 70–90, 2018.

[10] C. Zhang, G. Yang, Y. Zhang, Y. Zhu, X. Liu, and X. Yang, "Crop monitoring using satellite/uav data and machine learning: A review," *Field Crops Research*, vol. 231, pp. 103–113, 2019.

[11] TensorFlow Developers, "Tensorflow: Large-scale machine learning on heterogeneous systems," https://www.tensorflow.org/, 2015, version 2.15, accessed May 2025.

[12] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[14] M. Teeuw. (2016) Magicmirror^2. Open-source smart mirror platform. [Online]. Available: https://magicmirror.builders/

[15] Docker Inc., "Docker documentation," https://docs.docker.com/, 2025, containerization reference for deployment chapter.

[16] MathWorks Inc. (2025) Thingspeak communication service. IoT analytics platform used for prototype validation. [Online]. Available: https://thingspeak.com/

[17] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[18] G. Franzoni, A. Ferrante, and R. Bulgari, "Biostimulants as a strategy for improving the tolerance of lettuce (lactuca sativa l.) to abiotic stress," *Scientia Horticulturae*, vol. 297, p. 110943, 2022.

[19] S. J. Taylor and B. Letham, "Forecasting at scale," https://facebook.github.io/prophet/, 2017, prophet open-source forecast library.

[20] N. Coppedè, M. Janni, M. Bettelli, C. L. Maida, F. Gentile, M. Villani, R. Ruotolo, S. Iannotta, N. Marmiroli, M. Marmiroli, and A. Zappettini, "An in vivo biosensing, biomimetic electrochemical transistor with applications in plant science and precision farming," *Scientific Reports*, vol. 7, no. 1, p. 16217, 2017. [Online]. Available: https://www.nature.com/articles/s41598-017-16217-4

[21] M. Bettelli, F. Vurro, R. Pecori, M. Janni, N. Coppedè, A. Zappettini, and D. Tessera, "Classification and forecasting of water stress in tomato plants using bioristor data," *IEEE Access*, vol. 11, pp. 34 795–34 806, 2023, reference study for water stress classification using bioristor ground truth.

[22] R. Albino *et al.*, "Source-sink interaction: a century old concept under the light of molecular systems biology," *Journal of Experimental Botany*, vol. 68, no. 16, pp. 4417–4432, 2017.

[23] N. Nadezhdina, "Sap flow index as an indicator of plant water status," *Tree Physiology*, vol. 19, no. 13, pp. 885–891, 1999. [Online]. Available: https://doi.org/10.1093/treephys/19.13.885

A | Supporting Material

A.1. Source Code Listings

For reference, the following excerpts reproduce core routines mentioned in Chapters 3 and 5. Listing A.1 shows the Raspberry Pi serial ingestion loop, Listing A.2 documents the corresponding firmware logic, and Listing A.3 summarizes the LSTM forecasting model used during the AI evaluation.

```python
def serial_reader_thread(port):
    ser = get_serial_connection(port)
    while True:
        line = ser.readline().decode('utf-8', errors='replace').strip()
        if not line:
            continue
        data = json.loads(line)
        save_sensor_history(port, data.get('device_id', 'unknown'),
    ↪ data.get('sensors', {}))
        process_device_data(port, data)
        time.sleep(0.1)
```

Listing A.1: *Serial ingestion loop on the Raspberry Pi.*

```
if (now - last_moist_read > delay_moist_read || last_moist_read == 0) {
    last_moist_read = now;
    soil_moisture = analogRead(soil_moisture_pin);
}
StaticJsonDocument<512> doc;
doc["device_id"] = ID_NUMBER;
JsonObject sensors = doc.createNestedObject("sensors");
sensors["soil_moisture"] = soil_moisture;
sensors["temperature"] = readTemperatureFiltered();
sensors["humidity"] = dht22.getHumidity();
sensors["light"] = int(event.light);
Serial.println(doc.as<String>());
```

Listing A.2: *Firmware excerpt preparing JSON payloads.*

```
def build_lstm(sequence_length: int, feature_dim: int) -> tf.keras.Model:
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(sequence_length, feature_dim)),
        tf.keras.layers.LSTM(128, return_sequences=True),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.LSTM(64),
        tf.keras.layers.Dense(32, activation="relu"),
        tf.keras.layers.Dense(1, activation="linear"),
    ])
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
        loss="mae",
        metrics=["mae", "mse"],
    )
    return model
```

Listing A.3: *LSTM forecasting model used in the IMEM experiments.*

A.2. Extended Tables

Table A.1 summarizes representative data captured during the Agricola 2000 pilot. Table A.2 details the dataset splits used for AI experimentation, including contributions from the IMEM greenhouse.

Timestamp	Device	Soil Moisture (%)	Temperature (°C)
2024-11-21 15:07	70000000	52	30.9
2024-11-22 08:30	70000001	78	24.6
2024-11-23 12:45	70000000	49	26.1
2024-11-24 09:10	70000001	81	23.8

Table A.1: *Sample observations from the Agricola 2000 deployment.*

Dataset	Samples	Split (%)	Notes
Agricola 2000 Sensor Streams	18,240	60 / 20 / 20	Balanced by stress group
IMEM Greenhouse Multimodal	9,600	50 / 25 / 25	camera embeddings, sensors readings and Bioristor.
Historical Archive (2019–2023)	42,500	70 / 15 / 15	Used for pretraining

Table A.2: ***Dataset composition for AI training and evaluation.***

A.3. Pilot Deployment Logs

Table A.3 documents the most significant incidents encountered during the Agricola 2000 and IMEM pilots. Each entry captures the trigger, observed impact, and mitigation adopted in the experiment run.

A.4. Additional Figures

Additional dashboard screenshots, wiring diagrams, camera placements, and greenhouse imagery are available in the project repository under the `references/` and `static/` directories. These materials complement the descriptions in Chapters 3 and 5. A 40GB collection of images taken on both experiments (and other trials done to calibrate the setup) are available on personal cloud and can be disclosed for further models training.

Date	Trigger	Observed Impact	Mitigation
2025-02-11	LED lighting peak in IMEM greenhouse	Wi-Fi loss on two Plant Sensor nodes; delayed telemetry and alert bursts	Deployed dedicated access point, set fixed channel 6, added ferrite chokes to sensor harnesses
2025-03-02	Solenoid wear at Agricola 2000	Valve failed to reseat, causing overwatering and flooded tray	Replaced with peristaltic pump, introduced inline flow meter with watchdog alerts
2025-03-18	MQTT broker overload during firmware upload	Command acknowledgments queued behind debug messages; dashboard showed stale values	Split telemetry and control topics, enabled priority queuing, throttled debug output
2025-04-05	OTA firmware regression (Espressif core update)	Devices behind TLS proxy aborted downloads, requiring manual intervention	Pinned toolchain, restored SHA256 checks, exposed fall-back USB flashing task in dashboard
2025-05-22	Camera hub on shared USB bus	RGB feeds stalled and intermittently disconnected sensors	Moved cameras to powered hub, scheduled 30 min timelapses snapshot and serial interruption

Table A.3: *Chronology of operational incidents recorded during pilot deployments.*

B | Software Repository Structure

This appendix documents the organisation of the Smart Plants software release. The complete project source tree, including middleware, firmware, data exports, and AI notebooks, is available from the project Git repository. The repository used during development contains various branches, both used for the final thesis and draft, and also for the experimentation and evaluation of future possible features to integrate in the Smart Plants project:

- GitHub: github.com/GiulioSaulle/`Smart-Plants-Orchestrator`

Reproducibility notes:

1. A Python virtual environment with the packages in `requirements.txt` reproduces the middleware and analysis environment. See the project README for exact commands.
2. Firmware sketches live in `esp_flash_folder/` and can be flashed with the included helper scripts (`flash_esp32.sh`, `remote_flash_via_pi.sh`); see Section B.4 for details.
3. Training notebooks are in `ai_model/`; they include environment notes and dataset pointers that enable re-running the experiments on local or cloud hardware (see Section B.3).

The structure of the repository is hereby reported:

Entry	Role
`app.py`	Flask web server — main entry point; registers all REST and WebSocket routes, loads settings, starts background threads
`alerting.py`	Alerting engine; evaluates sensor readings against thresholds and triggers notifications
`mqtt_worker.py`	MQTT subscriber/publisher thread; bridges the broker to Firebase and the Flask event bus
`mqtt_dashboard.py`	Utilities for the live MQTT monitor UI
`smartplants_mqtt_firebase.py`	Cloud sync layer; forwards MQTT payloads to Firebase Realtime Database
`camera/`	Camera capture modules (`camera.py`, `camera2.py`); produce timestamped frames for the multimodal pipeline
`ai_model/`	All AI training and inference code (see Section B.3)
`templates/`	Jinja2/HTML templates for each web-UI page
`esp_flash_folder/`	OTA firmware flashing scripts and pre-compiled ESP32 binaries (see Section B.4)
`tests/`	Unit and integration tests
`scripts/`	Utility scripts (image export, data-column fixes)

Table B.1: *Top-level repository entries and their responsibilities.*

B.1. Configuration Layer

Runtime behavior is controlled by a set of JSON files loaded at startup by `app.py`. Keeping configuration out of the source code means that the same codebase can be redeployed on a different Raspberry Pi or against a different Firebase project without a single code change.

File	Contents
`general_settings.json`	MQTT broker address and port, serial port list, sampling interval, timezone
`experiment_settings.json`	Active experiment name, Bioristor plant IDs, labeling strategy (absolute / NR), stress thresholds
`firebase_settings.json`	Firebase project URL and database path
`plugin_settings.json`	Feature flags for optional subsystems (camera, AI inference, alerting)
`user_profiles.json`	Per-user notification preferences and dashboard layout
`users.json`	Hashed credentials for the local web-UI login

Table B.2: *JSON configuration files and their scope.*

B.2. Web Interface

The Flask server exposes a browser-based dashboard reachable at `http://<raspberry-ip>:5000`. Each HTML template in the `templates/` directory corresponds to one page (Table B.3). Pages that display live data use Server-Sent Events (SSE) pushed by the `mqtt_worker` thread.

Template	Function
`index.html`	Main dashboard: live sensor readings, plant status, recent alerts
`history.html`	Time-series charts for any sensor over a selectable date range
`ai_model.html`	AI inference panel: run on-demand stress classification, view SHAP plots
`alerts.html`	Alert log and threshold configuration
`irrigation_calendar.html`	Schedule and history of automated irrigation events
`experiment_settings.html`	Edit `experiment_settings.json` from the browser
`firebase_settings.html`	Edit Firebase connection parameters
`upload_firmware.html`	OTA firmware upload to connected ESP32 nodes
`user_management.html`	Add/remove users and reset passwords
`mqtt.html`	Live MQTT message monitor
`login.html`	Authentication page

Table B.3: *Web-UI templates and their functions.*

B.3. AI Model Subdirectory

The `ai_model/` subdirectory contains all machine-learning code. It is self-contained: training is performed in Google Colab (or locally with a GPU) and the resulting `.pkl` / `.pt` artefacts can be copied to the Pi for inference.

`shared.py` Common preprocessing class (`SensorDataPreprocessor`): rolling smoothing, resampling to 15-minute intervals, lag-feature generation, and label encoding. Imported by all training scripts to ensure consistent feature pipelines.

`smartplant_models.py` IMEM Bioristor experiment training pipeline: data loading from Firebase JSON exports, Rds merging, Random Forest / XGBoost / MLP / LSTM training and cross-validation, SHAP analysis.

`agricola2000_classification.py` Agricola 2000 pipeline: Excel ingestion, imbalance handling, Random Forest / Gradient Boosting / MLP / Autoencoder training, probability-threshold tuning, out-of-time holdout evaluation.

`case_study_classification.py` Standalone classification case study used for Chapter 5 figures.

`case_study_forecasting.py` Standalone LSTM forecasting case study.

`case_study_multimodal.py` Multimodal ResNet-18 + LSTM case study.

`smartplant_models_new.ipynb` Interactive Colab notebook version of the IMEM pipeline (used for exploratory runs and figure generation).

`agricola2000_classification.ipynb` Interactive Colab notebook version of the Agricola 2000 pipeline.

B.4. Firmware and OTA Flash Tools

The ESP32 firmware is an Arduino sketch located at the repository root:

`plant_sensor_wifi_extended.ino` Main sketch: reads soil-moisture (ADC), temperature and humidity (DHT22), and ambient light (TSL2561); assembles a JSON payload; transmits over serial (USB/UART) to the Raspberry Pi and over Wi-Fi to the MQTT broker when connectivity is available.

`Keys.h` Compile-time constants for Wi-Fi SSID/password, MQTT broker IP, and device ID. **This file should be excluded from version control if an .env is not being used** (listed in `.gitignore`).

The `esp_flash_folder/` contains two shell scripts for flashing without a physical connection:

`flash_esp32.sh` Flashes the compiled binary in `bin/` to a locally attached ESP32 over USB using `esptool.py`.

`remote_flash_via_pi.sh` Copies the binary to the Raspberry Pi over SSH and triggers the flash remotely; useful when the node is already deployed in the field.

B.5. Deployment and Startup

On the Raspberry Pi the platform is started with a single command:

```
1 python3 app.py
```

Listing B.1: ***Starting the Smart Plants platform on the Raspberry Pi.***

`app.py` reads the JSON configuration files, starts the MQTT worker thread, the serial reader threads (one per detected USB port), the alerting engine, and finally the Flask development server on port 5000. For production deployments a `systemd` service unit can be written to wrap this command, ensuring the platform restarts automatically after reboots or crashes. The Firebase credentials file (`smart-plants-ae494-firebase-adminsdk-*.json`) must be present in the root directory, can be securely provided via environment variables of the deployment environment, since it is correctly excluded from version control and must be provisioned separately from the Google Cloud console.

Acronyms and Abbreviations

ADC Analog-to-Digital Converter

AUROC Area Under the Receiver Operating Characteristic Curve

CNN Convolutional Neural Network

GPIO General Purpose Input/Output

IoT Internet of Things

LSTM Long Short-Term Memory

MAE Mean Absolute Error

ML Machine Learning

MLP Multilayer Perceptron

MQTT Message Queuing Telemetry Transport

NR Normalized Response

NVS Non-Volatile Storage

OECT Organic Electrochemical Transistor

OTA Over-the-Air

REST Representational State Transfer

RF Random Forest

ROC Receiver Operating Characteristic

SHAP SHapley Additive exPlanations

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to my supervisor Prof. William Fornaciari, whose guidance led me throughout this research, and to my colleague Gabriele Romano, together we developed the first version of Smart Plants. My thanks also goes to Agricola 2000 for providing instruments and field feedback that shaped the centralized control model, and to IMEM that gave me the space and the knowledge to operate on the data harvest, guiding me throughout the experiment.

I will cherish the beautiful moments shared with university colleagues Cristian, Marco, Giorgio, Etion, Salim, Mirko and Hajsen. This journey would not have been the same without your help and company, thank you all. Also, to all my colleagues of projects and tasks, my work colleagues at Genomsys and my band mates of SUMO and Project One for what they have done through these years goes my kindest gratitude for your patience and support.

Thanks to Giuseppe, Elvira, Roberto, Claudio, Lucina, Fabiana, Paolo, Aldo, Michele, Sandro and Mariarosa, to each one of you who kept laughing at my jokes and gave a sense to the word "family", you don't imagine how much it meant to me.

To my mother Angela, who taught me the importance of never giving up, even when being alone with a heavy burden to carry, you demonstrated the strength of a giant facing immense struggle, and I will always be in debt for what you did.
Paolo, my brother, your mind encloses creativity that no one will ever understand, even so, I want to give you an advice: go out and see the world, make it oblige to your will and valor, and never put a limit to your curiosity, nor let anyone tell you what you can do.

Mostly, the deepest gratitude goes to the one that never left my side, the love of my life to whom I dedicate my entire existence. You were there when I reached the bottom, put me back on my feet, and said that I could do it. Beatrice, if you read these words, know that an entire degree and this thesis could not conceive my feelings, you make my heart race at each glance of your endless eyes, like the first day we met, still to this day. I love you.

To Bice, who gave her all and still couldn't see this day. Nonna as usual I am late and I'm sorry for that, but I finally did it and without you it would not have been possible. To you I'm forever grateful. I need to ask you just one last favor: keep an eye on Charlotte for me.