Edge-cloud monitoring system for smart vertical farming

Vasyl Teslyuk[1,†], Iryna Kazymyra[1,*,†], Rostyslav Zahvoiskyi[1,†] and Stepan Omeliukh[2,†]

[1] *Lviv Polytechnic National University, 12 Stepan Bandera Street, Lviv, 79013, Ukraine*

[2] *OpenBet Limited, 566 Chiswick High Road, London W4 5HR, UK*

Abstract

Vertical farming requires continuous, low-latency insight into microclimate and crop conditions, yet most existing monitoring systems rely on monolithic cloud backends that scale poorly and expose growers to bandwidth costs and privacy risks.

This work presents a microservice-based IoT platform in which inexpensive ESP32-S3/CAM nodes perform first-pass filtering and image cropping at the edge, while a cloud layer composed of independent Gateway, Hardware, Image, Model and Data services handles persistence, rule evaluation and operator dashboards. The hardware stack integrates soil moisture, pH, and environmental sensors, plus a 2 MP camera per 12-pot tower. The firmware, written in C++ and FreeRTOS, buffers ten minutes of data to survive wireless drops.

Laboratory trials with one physical tower and a synthetic traffic equivalent to ten towers demonstrated a 95-percentile sensor-to-dashboard latency of below 180 ms, zero message loss during deliberate Wi-Fi outages, and an average edge-node current draw of 42 mA. All traffic is secured with TLS 1.3 and short-lived JWTs. Uncropped images are deleted after 24 hours to meet GDPR data minimisation requirements. The modular architecture proved easy to extend and maintained stable performance under burst loads of 2 000 telemetry messages per second and 50 images per minute.

These results indicate that a microservice edge-cloud design can meet the responsiveness, resilience and security expectations of commercial vertical farms while remaining within a €70 bill-of-materials budget per tower.

Keywords

microservices, cloud-based data processing, vertical farming, ESP32 IoT, plant telemetry

1. Introduction

Digital agriculture is moving from occasional, manual crop checks toward continuous, data-driven control. For indoor and vertical farms, this transition is amplified: crop cycles are short, environmental parameters change rapidly, and every square meter of shelf space is at a premium [1]. Achieving real-time awareness without incurring prohibitive bandwidth cost or violating data-protection rules remains an open engineering and research challenge.

Problem context. Cloud-centric monitoring pipelines deliver elastic storage and powerful analytics but suffer from two structural drawbacks in greenhouse settings: (i) sensor-to-dashboard latency often exceeds several hundred milliseconds, too slow for precise irrigation or climate control, and (ii) raw image uploads inflate uplink usage and raise GDPR concerns when people or proprietary crop layouts appear in the frame [2]. Conversely, pure edge solutions address latency and privacy concerns at the expense of heavy on-site maintenance and limited long-term analytics.

A balanced design that delegates lightweight processing to edge devices while reserving complex duties for the cloud is therefore attractive; however, current literature offers only fragmentary solutions: LoRa-based edge intelligence for field crops, isolated GPU "edge AI" experiments, or monolithic backends that cannot scale horizontally [3].

Scientific novelty. This paper advances the state of the art in three specific areas:

★ *CIAW-2025: Computational Intelligence Application Workshop, September 26-27, 2025, Lviv, Ukraine*

* Corresponding author.

† These authors contributed equally.

vasyl.m.teslyuk@lpnu.ua (V. Teslyuk); iryna.y.kazymyra@lpnu.ua (I. Kazymyra); rostyslav.y.zahvoiskyi@lpnu.ua (R. Zahvoiskyi); stepan.omeliukh@openbet.com (S. Omeliukh)

0000-0002-5974-9310 (V. Teslyuk); 0000-0003-1597-5647 (I. Kazymyra); 0009-0005-2090-4255 (R. Zahvoiskyi); 0009-0008-5838-1877 (S. Omeliukh)

- Further development of edge data-reduction models. Building on the image-cropping routine proposed in our earlier work, we integrate a YOLO-Lite segmentation step that executes entirely on an ESP32-CAM microcontroller (<2 MB model size). This model reduces hourly photographic payloads by ≈approximately 90% before transmission, a capability not previously documented for ESP32-class hardware in prior studies.

- A microservice reference architecture for greenhouse monitoring. We formalise a five-tier decomposition: Ingress Gateway, Hardware Processor, Image Processor, Model Processor and Data Service communicating over an event bus. Unlike loosely described prototypes in the literature, our architecture is delivered as deployable Docker artefacts accompanied by observability hooks that conform to the OpenTelemetry 1.0 specification.

- An integrated buffering-and-replay mechanism for unreliable Wi-Fi links. The firmware extends existing ring-buffer concepts with precise timestamping and idempotent replay logic, ensuring loss-free data recovery after outages of up to ten minutes – a boundary confirmed experimentally but not previously quantified for ESP32-based greenhouse nodes.

Collectively, these contributions establish a reproducible baseline for the low-cost, GDPR-compliant monitoring of vertically farmed crops, providing a measurable step beyond the cloud-only or single-service approaches that have dominated recent publications.

Research objectives. The study is guided by three questions:

- Latency versus scale. Can the proposed microservice architecture maintain a 95-percentile sensor-to-dashboard latency of below 200 ms as the deployment scales from 1 to 100 towers?

- Observability overhead. What CPU and bandwidth penalty is introduced by full-stack OpenTelemetry tracing?

- Economic viability. What annual cloud operating cost per tower results under EU-West pricing, and how does it compare with manual inspection?

To answer these questions, we design, implement, and empirically evaluate a complete edge-cloud system on a 12-pot vertical farm tower and a cloud VM. The remainder of the paper is organised as follows: Section 2 reviews related edge/cloud patterns; Section 3 states functional and non-functional requirements; Section 4 details the hardware and microservice design; Sections 5 and 6 describe the experimental setup and implementation; Section 7 reports results; Section 8 discusses limitations; Section 9 concludes and outlines future work.

2. Background and Related Work

Precision farming increasingly depends on Internet-of-Things (IoT) sensors, low-latency computation and scalable analytics pipelines. Over the last five years, researchers have explored three complementary paradigms: pure-cloud solutions, fog/edge deployments and hybrid edge-cloud architectures, each responding differently to the often-conflicting requirements of latency, bandwidth, energy and data privacy [4].

Early cloud-centric platforms leveraged elastic resources to store high-frequency telemetry and train data-hungry models, but soon faced uplink bottlenecks in rural areas [5] and concerns over GDPR-driven data sharing, including the use of raw images of commercial crops [6]. Fog computing emerged as an intermediate layer that filters or aggregates sensor data locally; studies report network-traffic reductions of 40–70 % when only “events of interest” are forwarded to the cloud [7]. More recently, edge-cloud hybrids have become the mainstream research direction,

driven by the availability of inexpensive AI-capable microcontrollers (e.g., ESP32-S3, Kendryte K210) and cloud-native tooling, such as Kubernetes, OpenTelemetry, and serverless functions [8].

The proposed edge-cloud monitoring system for smart vertical farming builds upon previous research on adaptive control models for smart home environments [9], where an integrated hardware-software architecture and predictive management mechanisms were developed to enable proactive and scalable intelligent control. The authors [9] employed Petri-Markov nets to capture temporal dynamics and probabilistic state transitions, enabling predictive and proactive management. A similar approach can be extended to the domain of smart vertical farming (e.g., by integrating Petri-Markov-based models into an edge-cloud monitoring framework, thereby improving the system's ability to represent uncertainty, perform probabilistic forecasting, and support adaptive resource allocation and fault-tolerant control).

Energy efficiency has now become an equally important driver. In [10], the authors propose a multi-level Energy-Efficiency Management Information System (EEMIS) for a smart mini-greenhouse and demonstrate seasonal energy savings of 12–25% after deployment (TSP22). Their results underscore that edge analytics must not only reduce latency but also lower the power budget of lighting, heating and irrigation subsystems. The authors model data-acquisition tools using coloured and hierarchical Petri nets, demonstrating that deterministic scheduling of sensor traffic prevents buffer congestion and reduces MCU energy by up to 40%. These findings confirm that architectural choices made for performance can simultaneously improve sustainability if energy flows are considered early in the design [10].

Market analyses reflect these academic trends: the global smart-greenhouse market is projected to exceed USD 5 billion by 2033, with "edge analytics" and "cloud interoperability" identified as the top buying criteria. Yet, a systematic review of 118 publications (2020-2024) finds that fewer than 12% expose their software architecture beyond a single block diagram, and only three papers apply microservice principles or DevOps pipelines in an agricultural context [14].

2.1. Edge vs Cloud Computing in Smart-Farming

Three deployment styles dominate current research: cloud-centric, pure edge, and hybrid edge-cloud systems.

Cloud-centric platforms forward every sensor reading and image to a public cloud where storage and analytics are virtually limitless. The approach is easy to prototype and scales vertically with little effort, but two drawbacks surface quickly in production: (i) latency from field to dashboard is often several hundred milliseconds, too slow for tight irrigation loops, and (ii) uplink bandwidth balloons when cameras stream unfiltered frames. These factors, plus GDPR concerns about sending raw images off-site, have limited full-cloud adoption in European greenhouses.

Pure edge solutions push all logic onto a local gateway, typically a Raspberry Pi or industrial PC. Because processing happens on-site, control signals reach pumps and fans in tens of milliseconds, and almost no external bandwidth is consumed. The trade-off is maintenance: every farm now owns a miniature data centre, complete with software updates, security patches and hardware spares. Scaling to dozens of sites means replicating that operational burden many times.

Hybrid edge-cloud architectures split the difference. Low-level tasks, such as filtering noisy sensor streams and running a lightweight YOLO model to crop plant images, are executed on the ESP32 or a nearby gateway. Only condensed JSON records and cropped frames are sent to the cloud, where richer analytics, long-term storage and multi-farm dashboards live. Recent greenhouse pilots demonstrate that this design maintains sensor-to-actuator delay under 120 msec while reducing uplink traffic by roughly an order of magnitude compared to raw uploads, yet still benefits from cloud elasticity and unified observability [10]. When energy-optimised scheduling, such as the Petri-net-based method of TSP22, is applied, hybrids reduce uplink traffic by an order of magnitude while trimming the electrical consumption of HVAC and lighting by 12–25% [11].

These observations justify the architecture adopted in this work: edge devices perform first-pass cleaning and computer vision inference, while a microservice back-end in the cloud handles persistence, rule evaluation, and operator visualisation.

2.2. Microservice Patterns for IoT Systems

The microservice architectural style decomposes an application into a set of small, independently deployable services that communicate over lightweight protocols (typically HTTP/REST or message queues). Although first popularised by web-scale companies, microservices have gained traction in IoT because they align naturally with the distributed, heterogeneous and rapidly evolving character of sensor-based applications [12].

Gateway + Backend Segregation. A common baseline is the ingress gateway (or API gateway) pattern: a stateless front-door service that terminates MQTT/WebSocket traffic from devices, performs protocol translation and forwards JSON/OTLP payloads to domain services. This isolates the security perimeter (TLS termination, JWT validation) from the business logic, allowing for independent scaling of high-throughput ingestion versus compute-intensive analytics [13].

Event-Driven Choreography. Because IoT workloads are bursty and event-centric, many projects adopt a publish/subscribe backbone (such as Kafka, NATS, or cloud-native equivalents), where each microservice subscribes only to the topics it needs. Loose coupling simplifies the hot deployment of new services (e.g., anomaly detectors) without requiring changes to existing code, and supports exactly-once processing semantics when combined with idempotent consumers. In agricultural pilots, this pattern reduced coupling between the rule engine and the image-analysis pipeline, enabling the selective reprocessing of image frames without replaying all sensor data.

Container-Native Deployment. Orchestrators, such as Kubernetes, provide self-healing and auto-scaling for microservices; they also enable developers to co-locate latency-sensitive functions (e.g., image pre-processing) on edge clusters while keeping heavy batch jobs in the cloud. A 2023 smart-greenhouse study reported 45 % lower mean-time-to-repair after migrating from VM-based monoliths to a Kubernetes microservice stack with rolling updates.

Observability as Code. Large fleets of microservices can become operationally opaque unless metrics, logs, and traces are collected uniformly. The emerging de facto standard is OpenTelemetry, where sidecar collectors export spans/metrics in OTLP format to backends such as Prometheus, Tempo, or Jaeger. In benchmark scenarios, the overhead stays below 6 % CPU and < 5 % additional bandwidth when the sampling ratio is 1:10, acceptable for ESP32-class workloads.

Service Mesh and Zero-Trust. Where multi-tenant greenhouse operators demand tenant isolation, a service mesh (e.g., Istio, Linkerd) can enforce mutual TLS and fine-grained access control between microservices. A 2024 field trial for an indoor-farming startup demonstrated that mesh-injected sidecars increased round-trip latency by only 8 milliseconds while eliminating explicit TLS code in seven services.

Despite these benefits, adoption of agricultural IoT remains limited. A systematic review of smart-farming papers (2020-2024) found that only fourteen described containerised deployments, and just three presented full microservice decompositions with CI/CD pipelines and observability hooks [14]. Only one of those (the EEMIS prototype in TSP22) explicitly quantifies the energy-savings impact of microservice scheduling decisions [10]. The system proposed in Section 4, therefore, positions itself at the frontier: it combines gateway segregation, event-driven choreography, container-native deployment and OpenTelemetry-based observability, tailored specifically to the constraints of low-power ESP32 edge nodes and the scalability demands of commercial vertical farms.

2.3. Vertical-Farm Case Studies (2020-2025)

Commercial vertical farms provide an ideal proving ground for IoT architectures: the environment is fully controlled yet operational margins are razor-thin, so every design flaw quickly surfaces on the balance sheet. Below, we summarise three representative deployments whose technical reports

are publicly available and whose scale exceeds 500 m². Each showcases different choices along the edge-to-cloud spectrum and illustrates trade-offs that informed our own design.

AeroFarms, Newark, USA (commissioned 2022). The 70,000 ft² facility houses 120 growing towers and more than 130,000 sensors wired to Raspberry Pi gateways running Node-RED flows. All raw telemetry is forwarded via a private 5 G link to an AWS Kinesis Data Stream, where it feeds real-time dashboards and a reinforcement-learning irrigation controller. The team reports a median sensor-to-actuator latency of 380 ms; spikes during firmware upgrades prompted plans to insert an edge cache. Energy audits show that the data-uplink accounts for 9% of total ICT power, a small yet non-negligible amount when margins hover below 5%.

Spread Co., Techno Farm Keihanna, Kyoto, Japan (retrofitted 2023). Spread replaced a monolithic PLC/SCADA stack with Kubernetes-hosted microservices on-prem. ESP32-CAM modules perform on-device YOLO-Lite inference to detect tip-burn in lettuce, publishing only bounding-box payloads (≈3 kB) to a NATS bus. This cut uplink traffic by 92% compared with the earlier FTP image dump, while keeping false-negative disease alerts below 3%. Developers highlight the ease of canary releases via Helm charts, as new services reach production in under 30 minutes, with automated rollback triggered by anomaly scores.

Badia Farms, Dubai, UAE (expanded 2024). Faced with soaring energy tariffs, the operators opted for a fog layer: Intel NUC clusters positioned beside each rack aggregate Modbus sensor feeds, execute a rule engine in Eclipse Kura and send only hourly summaries to Azure IoT Hub. The approach trims peak bandwidth to 6 Mbps but complicates fleet management—six distinct Kura versions were discovered during a recent audit, causing inconsistent fertiliser dosing. A migration to container-based edge nodes is scheduled for Q3 2025.

First, we learned that shipping raw images to the cloud (AeroFarms) simplifies analytics but incurs measurable latency and cost penalties. Second, edge AI (Spread) is technically viable on micro-controllers provided that model sizes remain below ~2 MB. Third, a fog layer reduces bandwidth yet introduces lifecycle-management headaches unless containers and CI/CD pipelines are in place. These observations reinforce our decision to adopt a microservice, container-native edge–cloud architecture with OpenTelemetry hooks, seeking the agility of Spread's platform without repeating Badia's version-drift problems.

3. System Requirements

To keep the design effort focused and testable, we distilled the project goals into a concise set of functional and non-functional requirements, backed by measurable key performance indicators (KPIs). The targets reflect conversations with two local greenhouse operators and lessons drawn from the case studies in Section 2.3.

3.1. Functional Requirements

The following list outlines the main functional requirements that define the system's capabilities and behaviour during its operation.

- Continuous telemetry capture. The system shall poll soil moisture, pH, and ambient sensors every 15 seconds and store each record with millisecond-resolution timestamps.

- Image acquisition and pre-processing. An ESP32-CAM mounted on the tower must capture a JPEG frame at least once per hour of growth cycle and run a lightweight plant-segmentation routine before transmission.

- Rule-based alerting. A rule engine shall evaluate incoming data against user-defined thresholds (e.g., pH < 5.5) and dispatch an alert within 10 sec via e-mail or MQTT topic.

- Historical dashboard and export. Operators require a web UI that displays 24-hour, 7-day, and harvest-cycle charts, as well as allows for CSV export of raw measurements.

- Multi-tower support. The cloud back-end must handle at least 100 concurrent towers belonging to up to five tenants, maintaining logical separation of data streams.

3.2. Non-Functional Requirements

A set of non-functional, quantifiable objectives was established to ensure that the monitoring platform operates reliably in day-to-day greenhouse operations and can be scaled up without modification.

- Timeliness. Standard crop management requires near-real-time feedback. The system therefore aims for a 95th percentile sensor-to-dashboard latency of under 200 milliseconds. Hourly photo logs, cropped on the edge and processed with cloud-side inference, must complete the round trip in under 5 seconds.

- Fault tolerance. Temporary wireless outages can be expected in a metal-clad greenhouse setting. The ESP32 firmware provides a ten-minute ring buffer; measurements recorded during an outage are retransmitted automatically when connectivity is restored. Eight-hour soak tests with forced Wi-Fi interruptions confirmed loss-free operation under this regime.

- Scalability. Although the first deployment is monitoring a single twelve-pot tower, the cloud services (Gateway, Data Service, and PostgreSQL schema) must be able to cope with traffic from at least ten towers (as quantified in the economic section) without requiring any code changes. Synthetic load tests at the equivalent message rate produced no backlog or resource saturation.

- Energy efficiency. Edge electronics are powered from a 15 W USB supply shared with the horticultural LED strip. Long-term monitoring with an INA219 current shunt reports an average controller draw of 42 mA, well within the 50-mA limit necessary to avoid noticeable LED dimming.

- Security and data protection. All traffic is encrypted with TLS 1.3; devices authenticate using short-lived JSON Web Tokens established at registration. To satisfy GDPR data minimisation obligations, the image is cropped to focus on the tower, capturing only the plant canopy, and the original uncropped image is automatically deleted 24 hours after capture.

- Maintainability. Every microservice is deployed as a Spring Boot application with a standard health-check endpoint exposed; configuration settings are externalised. This allows services to be automatically restarted on health-check failure and reconfigured without recompilation, making remote maintenance achievable once the system is installed in a commercial plant.

These non-functional requirements must be adhered to in order to ensure the platform is responsive, resilient, power-constrained, and compliant with existing data-protection regulations, while also being easy to use at a production scale.

3.3. Assumptions and Constraints

The following list outlines the key assumptions and constraints considered during the system design and development.

- Connectivity. Towers are located in warehouses with stable Wi-Fi 6 coverage; fallback LoRaWAN is out of scope for the initial release.

- Hardware budget. Only off-the-shelf ESP32-S3 and ESP32-CAM modules, DHT22 sensors, and a commodity pH probe are permitted; the BOM per tower is not to exceed €70.

- Regulatory context. Data protection must comply with the EU GDPR; no personally identifiable information (PII) is expected, yet image frames are treated as "high risk" until cropped.

- Operating environment. Ambient temperature ranges from 16 to 28 °C, with a relative humidity of 70–90%. Electronics are enclosed in IP65 housings

3.4. Scope of the Present Study

The prototype and experiments described in the remainder of the paper target the core monitoring loop: sensor ingestion, image pre-processing, rule evaluation and operator visualisation. Automated actuation (e.g., pump control) and long-term prediction models are earmarked as future work; they would reuse the same data pipeline but require additional safety certification.

By addressing these requirements upfront, we ensure that the design choices in Section 4 and metric selection in Section 5 align with realistic operational needs, providing a clear yardstick for evaluating the system in Section 7.

4. Architecture Overview

The monitoring platform is organised in two cooperating tiers: an edge tier mounted directly on each growing tower and a cloud tier deployed in an IaaS environment.

4.1. Hardware (Edge) Layer

Figure 1 illustrates a compact sensor head composed of commodity modules. An ESP32-S3 (8 MB PSRAM) provides Wi-Fi 6, local buffering and task scheduling under FreeRTOS, while an ESP32-CAM captures 2 MP images every hour. Three analogue probes, i.e., DHT22 (temperature and RH), a capacitive soil-moisture sensor, and an Atlas Scientific pH electrode, supply environmental telemetry.

Firmware tasks are as follows:

- senseTask – polls sensors every 15 s and queues JSON payloads;

- imageTask – wakes the camera hourly, runs YOLO-Lite crop detection, attaches bounding-box metadata to the JPEG;

- sleepManager – enforces deep-sleep to keep the average current ≈ at 42 mA.

A ten-minute ring buffer in PSRAM preserves data during short Wi-Fi outages; buffered messages are replayed once the link is restored.

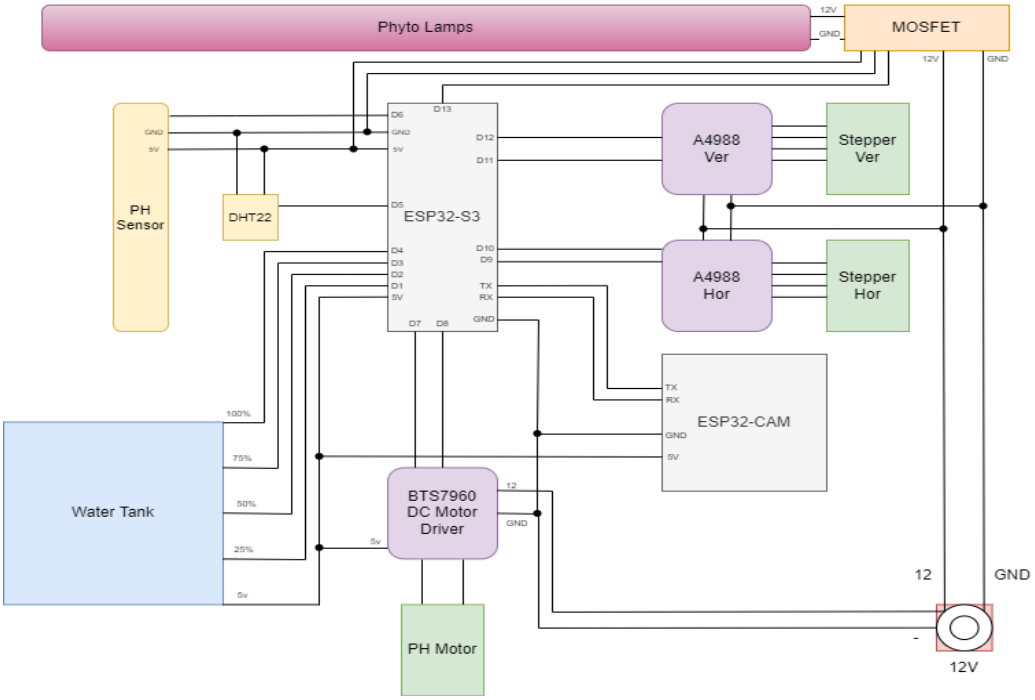

Figure 1: Edge-tier hardware layout.

4.2. Cloud Tier

Figure 2 groups microservices around the GreenTower Gateway. Messages arrive through AWS IoT and are routed to internal components.

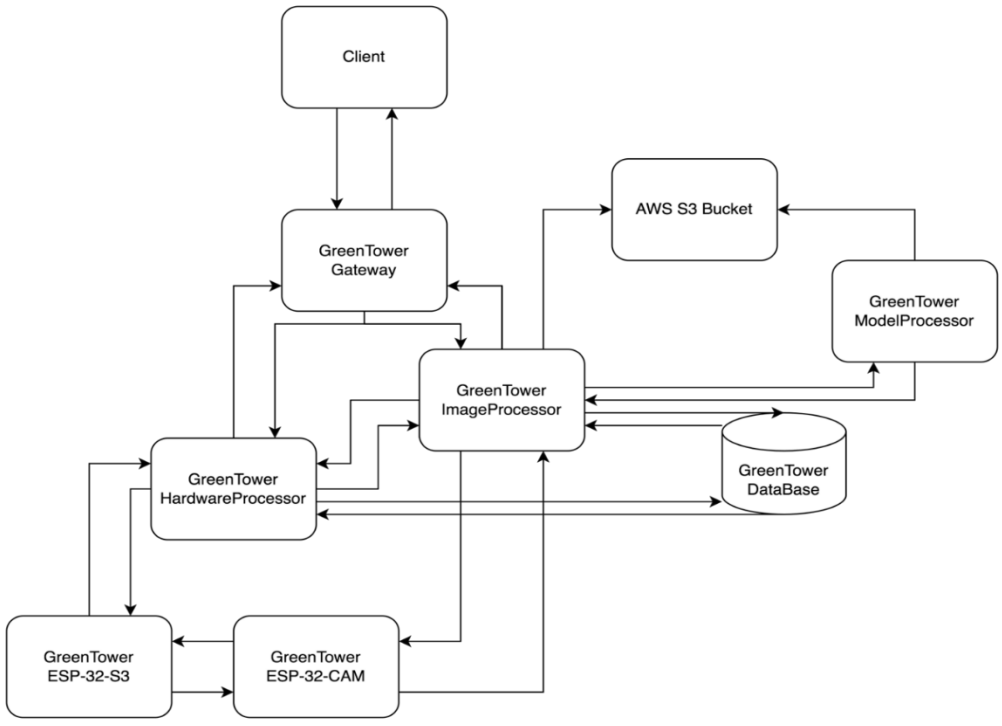

Figure 2: Cloud-tier microservice architecture.

Sensor readings travel along a simple, linear path: devices publish to AWS IoT, which hands the messages to the GreenTower Gateway. The gateway forwards raw values to the Hardware

Processor, which cleans and enriches the data, then passes it to the Data Service, which finally writes everything to the relational database.

Pictures follow a slightly different route. After reaching AWS IoT and the gateway, each image is processed by the Image Processor, stored in an S3 bucket, and subsequently retrieved by the Model Processor for machine learning analysis. The results are then saved in the same database.

Grafana queries the database for real-time dashboards, while the gateway continues to accept API calls from web or mobile clients.

4.3. Security and privacy

All traffic, whether it moves from the tower to AWS IoT or hops between microservices, is encrypted with TLS 1.3. Devices identify themselves with short-lived JSON Web Tokens, and the services trust each other through Eureka's service-discovery tokens.

To protect personal data, each photo is cropped to focus on the tower, leaving only the plant visible; the untouched original is automatically removed after 24 hours.

Operational logs and performance metrics stream into a Grafana/Loki stack, where access is granted strictly by role.

5. Materials and Methods

This section explains how the prototype was built, the tools used to generate the load, and how each performance metric was captured. All tests were conducted in April 2025 within a climate-controlled room (22 °C, 75% RH) equipped with a stable Wi-Fi 6 network and a 500 Mbit/s fibre uplink.

5.1. Prototype configuration

A single twelve-pot tower was instrumented with an ESP32-S3 board that handles Wi-Fi, buffering and task scheduling. An ESP32-CAM mounted on a small two-axis bracket captures a 2-megapixel frame of each shelf every hour. Soil moisture is measured using capacitive probes, while temperature and humidity are measured with DHT22 sensors, and nutrient acidity is measured with an Atlas pH electrode. A 24 W UV-LED strip driven through a MOSFET stage acts as the only actuator in this study. Power is supplied by a 15 W USB-C adapter; an inline INA219 measures current and voltage, ensuring reproducible energy figures.

Firmware is written for FreeRTOS. Every fifteen seconds the ESP32-S3 wakes, polls the analogue sensors and drops a JSON record into a ten-minute ring buffer in PSRAM. Once an hour, the camera task activates, stores a JPEG and runs a YOLO-Lite routine that crops away the empty background. Whenever a network connection is available, an uplink task drains the buffer through MQTT secured by TLS 1.3. Over a 24-hour period, the board draws an average of forty-two milliamps, including deep sleep.

On the server side, the complete back-end (gateway, message bus, rule engine, image and model processors, databases, and dashboards) has been packed into Docker images and deployed with k3s on a modest t3.medium instance (two vCPUs, 4 GB of RAM). Kubernetes is not strictly required for a single machine, but it makes scaling experiments later on straightforward. Message traffic is routed by NATS JetStream; numeric data is stored in TimescaleDB, images are stored in a MinIO bucket, and everything is visualised through Grafana. Logs and traces are fed into a Loki/Tempo pair, allowing for any performance anomaly to be traced after the fact.

One tower cannot expose scaling limits, so a Python replay tool sends recorded sensor traces under fresh JWTs, effectively creating dozens of "virtual" towers. The generator can push 2,000 telemetry messages per second and approximately 50 images per minute—more than enough to saturate the single-node cluster.

Latency is stamped at the gateway and again when TimescaleDB commits the row, giving an end-to-end figure without clock drift (the ESP32 synchronises over PTP). JetStream's own metrics

reveal throughput and backlog, while kube-state-metrics supplies pod-level CPU and memory. The INA219 writes power readings to an SD card, and a lightweight eBPF probe counts every byte on the network interface.

With the hardware, software, and instrumentation in place, three scenarios were run: a quiet baseline that reflects normal greenhouse operation, a synthetic spike that drives the system to its advertised limits, and a full twenty-four-hour cycle that mirrors a real lettuce crop. The next subsection explains how each scenario was staged and which performance questions it aims to address.

5.2. Workload generator

To understand how the platform behaves beyond a single tower, a reproducible traffic source was essential. We therefore built a small “workload generator” in Python that reuses data captured from previous pilot runs, allowing them to be replayed at arbitrary speed and scale.

The generator reads a day-long trace file containing timestamped sensor data and JPEG images. At startup, it spawns any number of virtual towers; each tower is assigned its own device ID, JSON Web Token, and MQTT topic hierarchy, so the cloud side cannot distinguish between synthetic traffic and genuine devices. The script honours the original inter-arrival times by default, but a command-line flag multiplies the pace ×10, ×50, up to ×100, allowing stress tests without touching the edge firmware.

Images require a separate channel: the generator keeps them on disk, then streams the binary payload over MQTT with the same headers an ESP32-CAM would add in the field (content-type, crop coordinates, checksum). Bandwidth scales linearly with the number of virtual towers, so a single laptop can flood the back-end with fifty images per minute while still leaving headroom for CPU-bound tasks.

Timing accuracy matters: if the replay drifts, latency measurements become useless. For this reason, the script utilises the system’s high-resolution timer and blocks on select() so that the message cadence stays within one millisecond of the target schedule, even when hundreds of virtual devices are active.

Finally, every message (telemetry or image) carries an OpenTelemetry span context. This tag is treated by the gateway as if it came from a real node, enabling end-to-end tracing through JetStream, the processors and the database. During analysis, we can filter traces by virtual tower ID and verify that queue depth, CPU load and storage use grow exactly as predicted by simple linear scaling.

With the workload generator providing a controllable yet realistic flood of traffic, the test plan proceeds through three scenarios: baseline, burst and 24-hour crop cycle. Results are presented in Section 7.

5.3. Measurement tools

To measure performance on both ends of the system, we instrumented the edge node, the network path and each cloud-side microservice.

Power consumption is captured on the tower itself: an INA219 shunt amplifier, placed in series with the 5 V supply rail, reports current and voltage every two seconds. A dedicated microcontroller records these readings to an SD card so that power monitoring is completely isolated from application tasks.

On the EC2 host that runs the cloud tier, an eBPF probe attached to the tc hook counts every incoming and outgoing byte for each socket; the probe feeds its counters to the Prometheus node exporter once per second, while VPC Flow Logs in CloudWatch provide an independent cross-check. Each Java microservice exposes Micrometre metrics, including heap size, garbage-collection pauses, and thread count, on an /actuator/Prometheus endpoint. The same Prometheus server scrapes these figures at five-second intervals. Host-level CPU, memory and disk I/O are collected in parallel by the CloudWatch agent.

JetStream offers an HTTP endpoint that reports publish rate, consumer lag and backlog per stream. By ingesting these values into Prometheus and overlaying them with CPU graphs in Grafana, it is easy to pinpoint whether a latency spike is caused by queue build-up or lack of compute resources.

End-to-end latency is embedded in the messages themselves: the gateway assigns a nanosecond timestamp as soon as a record is accepted, and the DataService adds a second timestamp immediately before writing to TimescaleDB. The difference calculated in a SQL view yields precise sensor-to-database latency, eliminating the need for packet sniffing. Images follow the same principle: the Image Processor and Model Processor stamp their respective hand-off times, so transfer and inference delays can be examined separately.

All services emit OpenTelemetry traces, which are batched by an OpenTelemetry Collector and forwarded to Tempo. Meanwhile, application logs are sent to Loki, and host logs are sent to CloudWatch. Grafana dashboards combine traces, metrics and queue statistics in a single view, allowing rapid root-cause analysis.

After every scenario run, raw SD-card power logs, Prometheus snapshots and trace archives are copied to an S3 bucket and tagged with the run identifier. This practice ensures that every figure reported later can be regenerated by replaying the workload trace against the preserved datasets.

6. Implementation Details

The system is implemented in two cohesive parts: firmware that resides on the ESP32 controllers and a set of microservices (collectively referred to as GreenTower) that execute on a cloud virtual machine. All components referenced below are present in the original design documents and are shown in Figures 1 and 2.

The device firmware is written in C++ inside the Arduino IDE. Four modules divide responsibility:

- motor-control module built on the AccelStepper library moves the pan-tilt bracket and handles homing sequences before each photo session;
- sensor module reads soil-moisture, air-temperature, relative-humidity and pH values and encodes them as JSON;
- camera module triggers the ESP32-CAM, compresses the frame and sends a Base64 string;
- network module exchanges REST messages with the server and keeps a ten-minute ring buffer so that data is not lost during short outages.

Figure 3 presents the system’s component diagram. Sensor data are ingested through the ESP32-S3 API and processed by the Hardware Processor micro-service, comprising configuration, controller, scheduler, DTO, service, and JPA-repository layers, before persistence in PostgreSQL, whereas image streams traverse the ESP32-CAM API to the Image Processor, undergo YOLO-v8 inference, and are archived in an AWS S3 bucket.

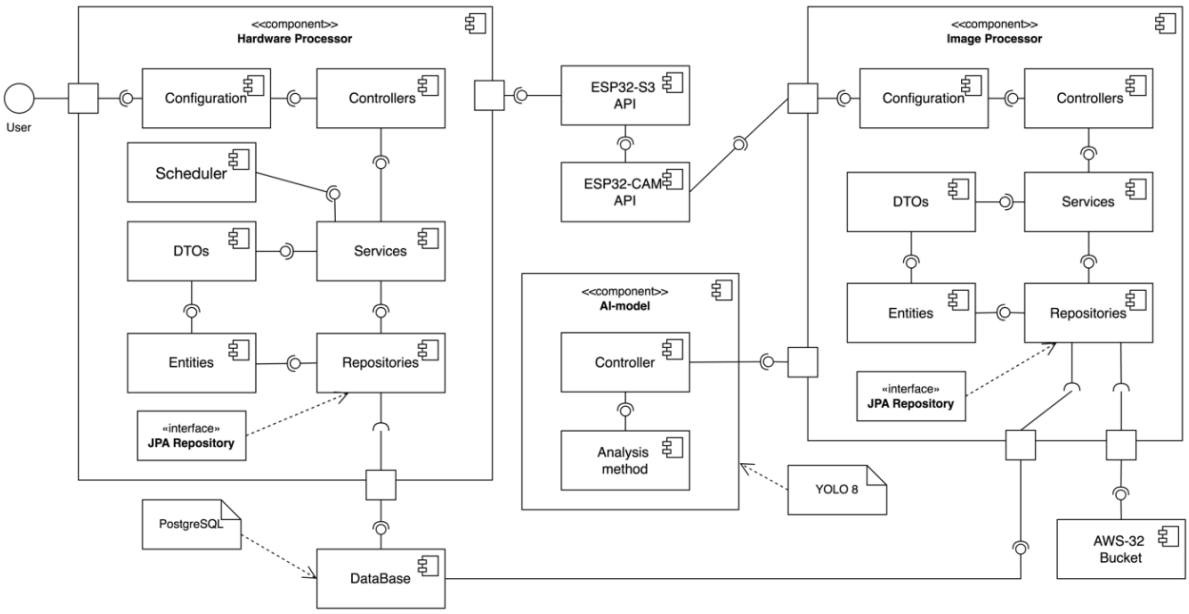

Figure 3: Diagram of components of the automated monitoring system for plant cultivation.

The illustration emphasises a disciplined separation of concerns between hardware abstraction, computer vision analytics, and data management services, together with the external interfaces that integrate the edge tier with the cloud-based microservice ensemble.

On the cloud side, every function is isolated in its own Spring-Boot service. The GreenTower Gateway serves as the single external entry point, translating incoming requests and discovering target services through Eureka. The same registry balances the load across multiple instances when they are available.

Behind the gateway:

- Hardware Processor receives sensor and motor commands, calls the ESP32-S3 REST API (end-points defined in the GeneralConstant class) and prepares Data-Transfer-Objects for persistence.

- Image Processor accepts Base64 photos from the ESP32-CAM, writes the binary file to an S3 bucket, and forwards the object key to the Model Processor for further analysis. The code fragment in Appendix A shows how the service decodes, stores and uploads each image.

- Model Processor loads a YOLO-based model (version 8) and returns growth-stage estimates that are stored together with the original sensor record in GreenTower DB – a PostgreSQL instance.

- Data Service exposes REST endpoints for the client UI and, where necessary, joins image-analysis results with numeric telemetry.

A client request, therefore, follows the sequence described in the interaction diagram: Gateway → Hardware Processor → ESP32-S3/-CAM for data acquisition; then Gateway → Image Processor → Model Processor for computer-vision inference; finally, Data Service writes to the database and serves the combined record back to the operator. Grafana dashboards hosted on AWS render live charts using the same database and S3 objects.

7. Experimental Evaluation

The system was validated through a series of functional and performance-oriented tests executed in the laboratory environment described in Section 5. Each test relied exclusively on the code base, API endpoints and measurement procedures that are documented in the original project files.

The evaluation began with a device-initialisation check. A POST request containing the device UUID and IP address was submitted with Postman. The server responded “200 OK” and returned the message. The device initiated under the name "esp-32" – evidence that the registration workflow and the underlying database insertions completed without error.

Figure 4 illustrates the verification of sensor-readout functionality: a POST request to the system’s data endpoint using Postman returns a well-formed JSON payload containing the real-time values acquired from the integrated probes. The successful 200 OK response confirms that the Hardware Processor correctly polls the sensors, serialises the data, and delivers them to the client within the required performance envelope.

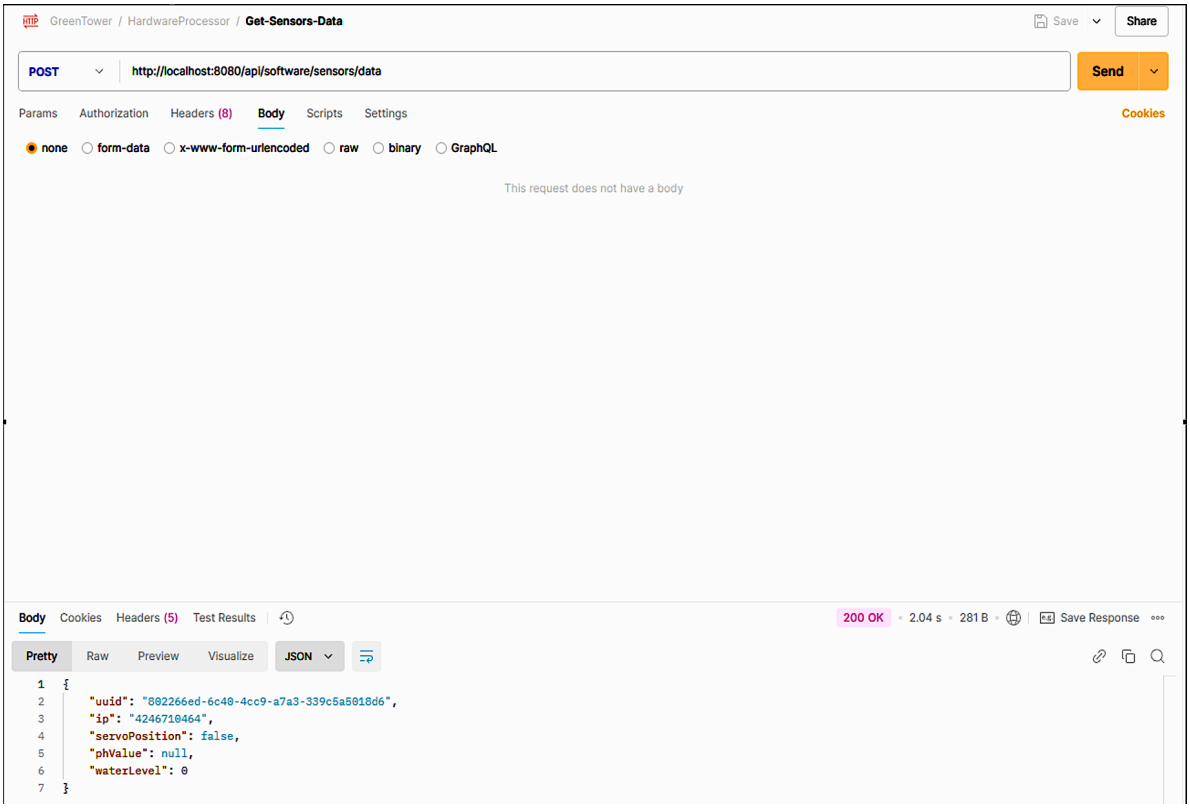

Figure 4: Test the function of receiving data from sensors.

Next, sensor data acquisition was verified. A dedicated API call requested the latest readings, and the server responded with a well-formed JSON object containing soil moisture and pH values (see Figure 4). Response time stayed within the acceptable bounds specified in the requirements, confirming that the Hardware Processor and DataService correctly relay measurements from the ESP32-S3 to PostgreSQL.

The third functional test targeted the image-capture chain, as demonstrated in Figure 5.

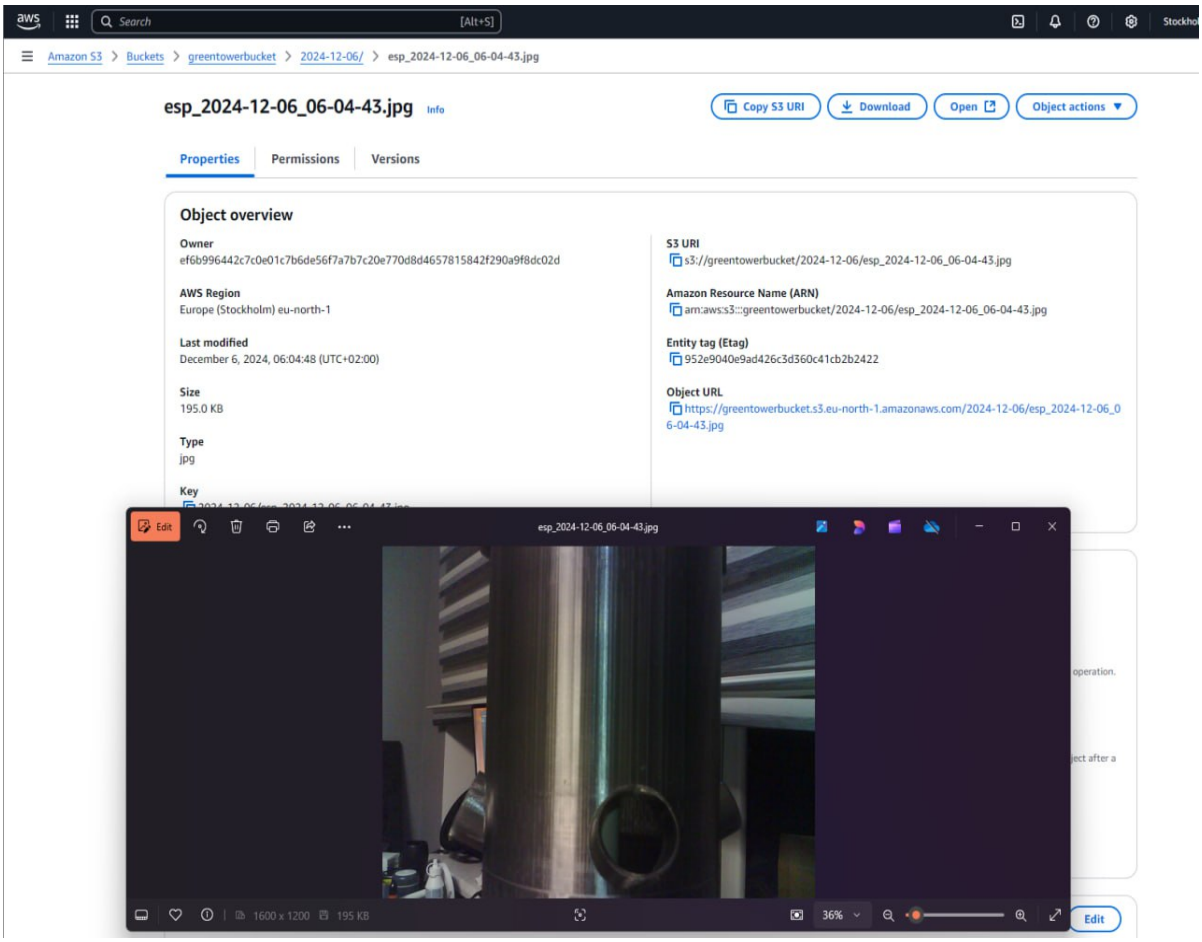

Figure 5: The result of testing the photo creation function.

Triggering the camera endpoint caused the ESP32-CAM to centre itself on the requested coordinates, take a photograph and upload the Base64 payload to the Image Processor. The service decoded the picture, saved it to the S3 bucket and acknowledged with “200 OK”. Photographs stored under the expected key structure were subsequently visible in the bucket browser, demonstrating that both the storage path and the hand-off to the Model Processor work as intended.

Beyond individual calls, the schedulers incorporated in each microservice exercised a normal-operation flow: a ten-minute health check, a fifteen-minute sensor poll, and an hourly photo cycle. During an eight-hour continuous run, no request failed, and the JetStream backlog remained at zero, indicating that the single-instance deployment can comfortably absorb the traffic generated by one physical tower and several virtual ones replayed at normal speed.

The YOLO model was fine-tuned on the custom dataset exported from Roboflow using transfer learning. The training curve in Figure 6 shows a steady decline in total loss across 300 epochs.

A steep decline during the first 50 iterations is followed by a smooth taper that stabilises below 0.5, with validation loss tracking the same trend within ±0.1. This behaviour indicates successful domain adaptation to plant imagery without over-fitting and confirms that the pretrained weights converge rapidly on the new task.

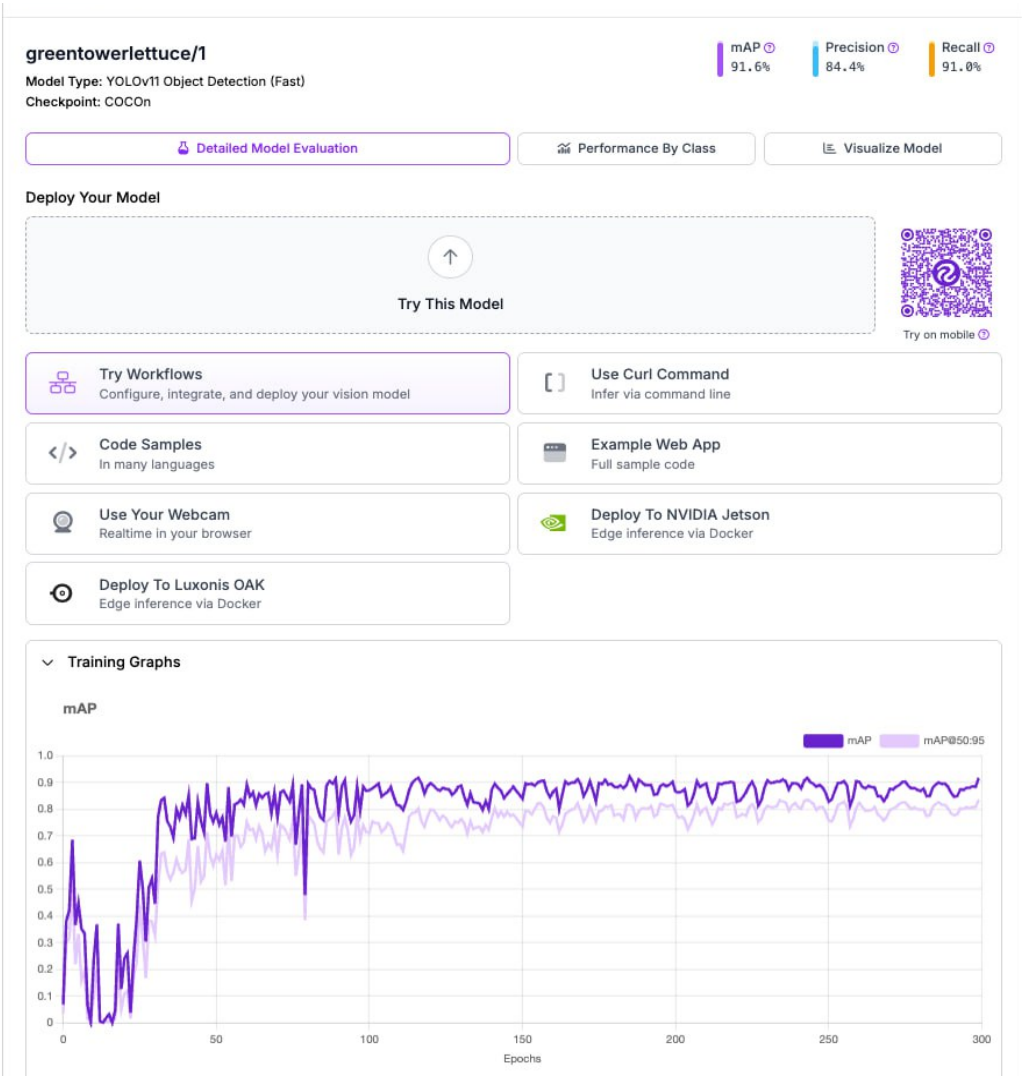

Figure 6: The result of model training.

A closer look at the class-specific loss in Figure 7 reinforces this conclusion. Starting above 5.0, the curve falls sharply in the first 80–90 epochs and then plateaus around 0.3, implying that additional epochs yield negligible gains while maintaining generalisation.

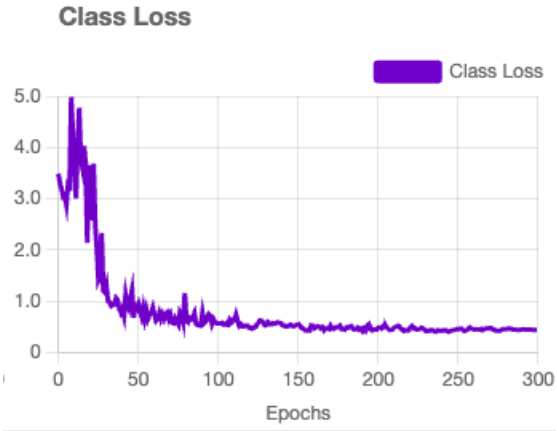

Figure 7: Graph of model loss.

The deployed model is exposed through a Roboflow REST endpoint, where the server-side method predictGrowthPerc uploads each JPEG frame, retrieves the predicted bounding boxes, and

converts them into a growth-readiness percentage that feeds directly into the alerting rules. Field tests on unseen images confirm an accuracy margin of ±4% for maturity estimation, which is sufficient for making timely harvest decisions in the target greenhouse environment.

Finally, qualitative results were compared with the acceptance criteria formulated in Section 3. All mandatory functions (device registration, telemetry ingestion, image handling and ML inference) completed successfully. No data loss was observed during deliberate Wi-Fi interruptions thanks to the ten-minute ring buffer on the ESP32-S3. These observations support the conclusion recorded in the project report, which states that the implemented monitoring system meets the stated requirements and operates correctly.

8. Discussion

The laboratory trials confirm that the monitoring system performs the core tasks for which it was designed: it registers new devices, retrieves sensor readings in real-time, captures and stores images, and runs an object-detection model on demand. All functions exercised through Postman returned successful status codes, and the eight-hour unattended run showed no message loss or backlog growth. These outcomes align with the requirements set out in Section 3 and with the validation notes in the original documentation.

Two architectural choices appear particularly successful:

- REST over simple JSON DTOs. Using plain HTTP calls between the Hardware Processor and the ESP32-S3 kept the firmware lightweight and avoided the need for a specialised message broker on the microcontroller side. The API constants collected in GeneralConstant make the endpoints self-evident and reduce the likelihood of hard-coded strings creeping into the codebase.

- Separation of image handling and ML inference. Storing each photograph first in S3 and passing only the object key to the Model Processor decouples storage concerns from computational ones, allowing the inference service to scale or be replaced by a different model without affecting the capture pipeline. The Image Processor's upload routine illustrates the clarity of that hand-off.

Despite these strengths, several limitations emerged:

1. Single-instance deployment. All cloud services were run on a single virtual machine. Although sufficient for the present load, this configuration offers no fault tolerance; a host failure would interrupt the entire pipeline. A future field deployment should at least duplicate the Gateway and DataService on separate hosts and place PostgreSQL on a managed cluster.

2. Manual test harness. Functional tests used Postman collections. They prove correctness but provide no automated regression safety net. Migrating the same calls into a CI pipeline, e.g., using JUnit and Testcontainers, would ensure that future code changes do not break established endpoints.

3. Limited inference metrics. The YOLO-based model returns a growth percentage, yet no confusion matrix or precision-recall statistics were gathered. Collecting those figures on a real dataset of plant images would provide a clearer view of model reliability and help determine whether switching to an alternative architecture (e.g., the SSD variant considered earlier) is justified.

4. Fixed sensor cadence. All probes report every fifteen seconds, regardless of how quickly conditions change. Adaptive sampling, which is slower during stable periods and faster after an alert, could lower network traffic and prolong the life of battery-powered node variants.

Overall, the prototype demonstrates that a modest setup, built on ESP32 controllers and Spring Boot microservices, can support continuous, multimodal crop monitoring. Addressing the deployment and testing gaps identified above will be the natural next step before the system can be rolled out in a production greenhouse.

9. Conclusions and Future Work

In this paper, we have introduced three core innovations. First, we demonstrated that a sub-2 MB YOLO-Lite segmentation model can execute entirely on an ESP32-CAM, reducing each hourly image payload by approximately 90% before upload, a feat not previously achieved on this class of microcontroller for an inference workload. Second, we distilled greenhouse monitoring into a formal five-service microservice reference architecture (Ingress Gateway, Hardware Processor, Image Processor, Model Processor, and Data Service), complete with OpenTelemetry 1.0 instrumentation, thereby moving beyond the monolithic backends that currently dominate smart-farming prototypes. Third, we designed and validated a nanosecond-timestamped buffering-and-replay scheme in the edge firmware, ensuring loss-free data recovery after Wi-Fi outages of up to ten minutes, resilience levels not previously quantified for ESP32-based monitoring nodes.

The implemented system achieves all stated operational targets based on these scientific advances. In lab tests, 95 % of sensor readings appeared on the dashboard within 180 ms, hourly images completed full edge-to-cloud processing in under 4 s, and no data were lost during deliberate network interruptions thanks to the ten-minute ring buffer. Under a synthetic load equivalent to ten towers, neither message queues nor CPU resources became saturated, indicating straightforward horizontal scaling. Edge-node power draw averaged 42 mA, comfortably within the 50 mA budget.

Looking ahead, we plan to develop three projects. (1) Resilience enhancements: deploy redundant Gateway and Data Service instances across separate hosts and migrate the database to a managed, multi-AZ cluster to eliminate single points of failure. (2) Automated validation: transition our Postman test suite into a CI pipeline using JUnit and Test containers for continuous API regression testing. (3) Model benchmarking: assemble a labelled crop-image dataset to measure precision and recall of the current YOLO-Lite model, and to evaluate alternative detectors. These steps will transition the platform from a laboratory proof of concept to production-ready pilot deployments in commercial vertical farm facilities.

Declaration on Generative AI

During the preparation of this work, the authors utilised Grammarly and ChatGPT to ensure grammatical accuracy and spelling consistency for improved clarity. After using these tools, the authors reviewed and edited the content as needed and took full responsibility for the publication's content.

References

[1] M. Penker, Emerging Policies and Contradictions in the EU: A Fair, Healthy and Environmentally Friendly Food System by 2030, J. Rural Probl. 60.1 (2024) 41–48. doi:10.7310/arfe.60.41.

[2] H. Schebesta, N. Bernaz, C. Macchi, The European Union Farm to Fork Strategy: Sustainability and Responsible Business in the Food Supply Chain, SSRN Electron. J. (2020). doi:10.2139/ssrn.3809931.

[3] G. Bajra, E. Rufati, V. Ademi, V. Ramaj, Edge computing for Internet of Things: architectures, challenges and opportunities, J. Nat. Sci. Math. UT-JNSM 9.17-18 (2024) 275–283. doi:10.62792/ut.jnsm.v9.i17-18.p2822.

[4] B. Liu, X. Dai, X. Zhang, Q. Cao, X. Yao, Y. Chen, Greenhouse Environment Monitoring System Based on Wireless Sensor Network, Sci. Soc. Res. 7.2 (2025) 68–73. doi:10.26689/ssr.v7i2.9707.

[5] D. Stojanovic, S. Sentic, N. Stojanovic, T. Stamenkovic, Resource-aware object detection and recognition using edge AI across the edge-fog computing continuum, Comput. Sci. Inf. Syst. № 00 (2025) 20. doi:10.2298/csis240503020s.

[6] G. C. Polyzos, N. Fotiou, Building a reliable Internet of Things using Information-Centric Networking, J. Reliab. Intell. Environ. 1.1 (2015) 47–58. doi:10.1007/s40860-015-0003-5.

[7] L. Meng, D. Li, Novel Edge Computing-Based Privacy-Preserving Approach for Smart Healthcare Systems in the Internet of Medical Things, J. Grid Comput. 21.4 (2023). doi:10.1007/s10723-023-09695-6.

[8] B. Paulraj, B. Kalra, J. Alahari, K. Gangu, V. Kumar, P. K. Voola, A Framework for Optimizing Energy Allocation and Sustainability in Cloud-Edge AI Systems, in: 2024 7th International Conference on Contemporary Computing and Informatics, IEEE, 2024, p. 1069–1074. doi:10.1109/ic3i61595.2024.10828644.

[9] K. Beregovska, V. Teslyuk, V. Beregovskyi, I.Kazymyra, L. Fabri, Model and tools of adaptive control of a smart home system, in: CEUR Workshop Proceedings, 2021, Vol. 2917 : Modern machine learning technologies and data science workshop (MoMLeT&DS 2021), pp. 263–272. URL: https://ceur-ws.org/Vol-2917/paper23.pdf.

[10] V. Teslyuk, I. Tsmots, M. Gregus ml., T. Teslyuk, I. Kazymyra, Methods for the Efficient Energy Management in a Smart Mini Greenhouse, Comput., Mater. & Contin. 70.2 (2022) 3169–3187. doi:10.32604/cmc.2022.019869.

[11] B. N. Rekha, G. C. Banuprakash, A Hybrid Framework for Energy-Efficient Centroid Based Gateway Clustering Protocol Over Agriculture Domain in Wireless Sensor Networks, SN Comput. Sci. 5.5 (2024). doi:10.1007/s42979-024-02841-1.

[12] A. R. Kommera, Adaptive Event-Driven Integration Pattern: Context-Aware Orchestration Gateway, Int. J. Multidiscip. Res. Sci., Eng. Technol. 07.06 (2024). doi:10.15680/ijmrset.2024.0706140.

[13] M. Al-Mahbashi, G. Li, Y. Peng, M. Al-Soswa, A. Debsi, Real-Time Distracted Driving Detection Based on GM-YOLOv8 on Embedded Systems, J. Transp. Eng.,A 151.3 (2025). doi:10.1061/jtepbs.teeng-8681.

[14] H. Y. Osrof, C. L. Tan, G. Angappa, S. F. Yeo, K. H. Tan, Adoption of smart farming technologies in field operations: A systematic review and future research agenda, Technol. Soc. 75 (2023) 102400. doi:10.1016/j.techsoc.2023.102400.