Microservice-Based IoT Architecture for Precision Agriculture: ESP32 vs. Arduino Edge Nodes

Natalia Axak*, Maksym Kushnaryov, and Yurii Shelikhov

Department of Computer Intelligent Technologies and Systems, Faculty of Computer Engineering and Control, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine

Email: nataliia.axak@nure.ua (N.A.); maksym.kushnarov@nure.ua (M.K.); yurii.shelikhov@nure.ua (Y.S.)

*Corresponding author

Manuscript received December 8, 2025; revised January 12, 2026; accepted January 19, 2026; published February 5, 2026

***Abstract*—This paper presents a microservice-based system architecture for collecting, processing, and analyzing agrotelemetry data in real-time to support precision agriculture. The study aims to evaluate the effectiveness of the ESP32 microcontroller platform (Espressif Systems) and Arduino Nano with Long Range (LoRa) platforms as edge nodes for monitoring agro-environmental parameters, including soil moisture, air temperature, and humidity. The objectives are: (i) to validate an end-to-end microservice streaming pipeline for agrotelemetry, (ii) to operationalize a low-latency critical-event detector in the streaming layer, and (iii) to compare ESP32 and Arduino Nano + LoRa under unified Key Performance Indicators (KPIs) (latency, reliability, accuracy, where applicable, and energy consumption) to provide evidence-based deployment guidance. The proposed architecture leverages Kubernetes, Apache Kafka, Apache Flink, and InfluxDB to ensure horizontal scalability, fault tolerance, and low-latency processing. For automated critical event detection, we implemented a novel streaming algorithm combining static thresholds with dynamic *z*-score analysis and a confirmation mechanism to reduce false positives. The experimental methodology involved laboratory tests and field trials conducted in a greenhouse. Results indicate a clear trade-off: the ESP32 platform achieved lower network latency and higher accuracy, while the Arduino Nano with LoRa was significantly more energy-efficient and demonstrated superior long-range link stability. Based on these findings, we recommend using the ESP32 for time-sensitive applications within Wi-Fi coverage and the Arduino Nano with LoRa for energy-constrained, remote deployments. A hybrid strategy is proposed to strike a balance between responsiveness and energy autonomy. The unified pipeline provides a reproducible framework for evaluating trade-offs among latency, accuracy, reliability, and energy consumption in agrotelemetry systems.**

***Keywords*—distributed computing, microservice architecture, cloud computing**

I. An Introduction to Agrotelemetry

Agrotelemetry—the remote measurement and wireless transmission of data from fields, livestock, and machinery—has matured into the data backbone of precision agriculture.

Agrotelemetry represents a transformative technology in modern agriculture, forming the central nervous system of the smart farm concept. This field involves the application of telemetric principles for the automated collection, wireless transmission, and real-time analysis of data related to agricultural objects.

The architecture of agrotelemetry systems is based on a 4-layer model, integrating sensory, communication, computational, and application levels. The sensory level encompasses deployed field sensors for soil, plant, and livestock parameters, as well as actuating mechanisms. The communication layer enables data transmission via specialized wireless protocols such as LoRa Wide Area Network (LoRaWAN), Narrowband Internet of Things (NB-IoT), and cellular networks. The computational tier handles data cleansing, aggregation, and analysis using edge computing for local decision-making and cloud technologies for in-depth analytics. The application level provides interfaces for data visualization and decision-support tools.

Key applications of these systems include precision irrigation based on soil moisture data, crop monitoring using multispectral sensors, tracking microclimate conditions, and monitoring livestock health. The agrotelemetry technology stack ensures optimized resource use, increased productivity, and reduced environmental impact of agricultural activities.

The industry's development is characterized by the integration of artificial intelligence for predictive analytics, the adoption of standardized Application Programming Interfaces (APIs) (such as the Open Geospatial Consortium (OGC) SensorThings API) to ensure system interoperability, and the emergence of autonomous robotic platforms. Implementing agrotelemetry solutions presents challenges, including high initial costs, technical complexity, and power supply requirements for peripheral devices.

Between 2019–2025, deployments consolidated around Low-Power Wide-Area Networking (LPWAN) and 4G/5G backhauls [1]; standardized data models (OGC SensorThings API for observations [2]; ISO 11783/ISOBUS for machinery telematics [3]); and edge-cloud patterns that close the loop for irrigation, greenhouse climate, and asset maintenance [4–6].

Agrotelemetry spans:

- In situ sensing (soil moisture/salinity, microclimate, leaf water stress, trunk diameter, etc.);
- Mobile platforms (Unmanned Aerial Vehicles /Unmanned Ground Vehicles (UAVs/UGVs));
- On board machinery telemetry (CAN/ISOBUS, engine/fuel, task controller).

Observations should be exposed via a uniform, geospatial API to enable cross-farm analytics and decision support.

The OGC SensorThings API (STA) is the recommended interface, with its Sensing component (Part 1) handling observations and its Tasking component (Part 2) managing device actuation.

This study addresses the lack of reproducible, KPI-driven comparisons between common low-cost edge platforms in real agrotelemetry pipelines. The objective of this work is to design and validate a microservice-based, stream-first architecture for real-time agrotelemetry, and to quantify the

trade-offs between ESP32 (Wi-Fi/MQTT) and Arduino Nano + LoRa as edge nodes within a unified data ingestion and analytics workflow.

Specifically, we pursue 3 measurable objectives:

- Architecture objective (O1): implement an end-to-end microservice pipeline (ingestion→ broker→ stream processing→ time-series storage→ dashboards/alerts) suitable for real-time agrotelemetry.
- Analytics objective (O2): implement and operationalize a streaming critical-event detector that combines static thresholds with dynamic *z*-score deviation and confirmation logic (*k*-of-*m* voting) to reduce false positives.
- Comparative objective (O3): evaluate ESP32 vs. Arduino Nano + LoRa using a common experimental protocol and standardized KPIs, including latency (P50/P95), message loss, measurement error (e.g., Mean Absolute Error (MAE) where applicable), and energy consumption, and derive evidence-based platform selection guidance.

Accordingly, the expected outcome is a set of experimentally supported recommendations (including a hybrid deployment strategy) that closes the loop between architecture design choices and measured operational KPIs.

II. Related Work

This section is structured as a scoping review to position the proposed architecture within the field–edge–cloud stack for agrotelemetry. We surveyed peer-reviewed publications (2020–2025) indexed in major scholarly databases (e.g., IEEE Xplore, ACM DL, Scopus/Google Scholar) using keyword combinations covering (i) smart/precision agriculture telemetry, (ii) Low-Power Wide-Area Networks (LPWAN), including LoRa/LoRaWAN and cellular IoT (NB-IoT/LTE-M/5G), (iii) MQTT/CoAP security, and (iv) edge–cloud streaming and microservice architectures. In addition, we included a limited number of recent preprints when they provide up-to-date comparative analyses not yet available in journal form (e.g., the SCI 2025 Springer chapter in Ref. [7]). We prioritized sources that report operational KPIs (e.g., latency, delivery ratio/loss, energy/device lifetime) or describe end-to-end pipelines; purely conceptual works are used only to define baseline terms.

Conceptual framing. We analyze prior work through a field–edge–cloud stack: (i) field sensing and edge nodes, (ii) last-mile connectivity and gateways, (iii) transport/messaging and security controls, and (iv) cloud/edge analytics and storage. Across these layers, the dominant evaluation dimensions are timeliness (tail latency), reliability (delivery ratio/loss under rural backhaul outages), and energy autonomy (device lifetime), which jointly determine the feasibility of continuous agrotelemetry and real-time actuation.

Agrotelemetry deployments today are best understood through the lens of a field-to-edge-to-cloud stack in which communication choices and data handling coevolve. Recent surveys frame agri-IoT as the backbone of precision and climate-smart agriculture, with connectivity, energy efficiency, and interoperability as first-order constraints. Within that frame, LoRa/LoRaWAN is favored for low-rate, long-range telemetry that can sustain multi-year battery life; empirical and survey studies report robust coverage, the practical benefits of Adaptive Data Rate (ADR), and successful irrigation and environmental monitoring projects, while also noting latency and downlink limitations [8]. NB-IoT and Long Term Evolution for Machines (LTE-M), delivered by mobile network operators, extend coverage with stronger Quality of Service (QoS)—well-suited to sparse sensors and mobile assets, such as pumps and tanks—though power budgets and subscription costs must be carefully weighed [9]. Where throughput and low latency are critical, as with bursty machine telemetry or UAV video, 4G/5G becomes the natural complement, and many deployments now combine LPWAN for slow telemetry with 5G for heavier payloads in hybrid architectures [7].

At the transport layer, Message Queuing Telemetry Transport (MQTT) remains the default choice, but it requires deliberate security hardening—such as authorization, Transport Layer Security (TLS), and careful broker exposure—while the Constrained Application Protocol (CoAP) persists on constrained nodes and introduces its own bootstrapping risks. Both stacks have known attack surfaces that must be mitigated systematically [10]. Since 2020, architectures have converged on edge gateways—typically LoRaWAN or NB-IoT concentrators—that pre-process, cache, and forward to cloud data lakes. On top, digital twins and decision-support systems consume standardized streams (e.g., OGC SensorThings, ISOBUS, AEMP/ISO 15143-3) to close control loops for irrigation, fertigation, and greenhouse Heating, Ventilation, and Air Conditioning (HVAC); case studies in greenhouses and orchards show improved water-use efficiency and reliability under ADR/just-in-time scheduling [4]. Alongside these patterns, reports emphasize practical edge–cloud stacks and the growing role of AI/ML at the edge for agriculture telemetry, paired with security hardening throughout the pipeline [11].

Security and reliability remain cross-cutting concerns: beyond device hardening, the literature repeatedly flags MQTT broker exposure and CoAP bootstrapping issues, and recommends end-to-end TLS with credential rotation. Robustness techniques—such as ADR on LoRaWAN, adaptive duty cycling, and local buffering—help farms mitigate rural backhaul outages [10]. Economically and environmentally, reviews link telemetry-driven irrigation and input optimization to measurable resource savings and sustainability gains, reinforcing broader Digital Agricultural Technology (DAT) adoption [12].

To make the review systematic, Table 1 synthesizes representative studies by strand, the KPIs they report, the main limitations, and how the present work addresses the identified gaps.

Microservices and API-driven edge–cloud architectures. Beyond agriculture-specific surveys, recent reviews of IoT/IIoT architectures emphasize microservice decomposition and explicit API contracts as a dominant pattern for scalable edge–cloud systems, typically deployed via containers and orchestrators. Surveys of IoT application architectures and edge–cloud continuum systems report that microservices and API gateways simplify the integration of heterogeneous devices/protocols, while enabling independent scaling and fault isolation at the service level.

Systematic reviews further note that the practical performance (including latency) depends not only on the network link but also on platform choices such as orchestration, scheduling, and the middleware stack; therefore, reproducible studies should disclose the software stack and baseline configurations used for latency measurements. Microservice- and API-centric edge–cloud IoT architectures are reviewed in Refs. [15–20], highlighting orchestration and interface standardization as key factors affecting practical latency and reproducibility (see Table 1).

As summarized in Table 1, prior studies rarely report standardized end-to-end KPIs under a unified experimental protocol, which motivates our KPI-driven evaluation and architecture validation in Section III.

Table 1. Synthesis of related work and gaps (Scoping)

Strand	Sources	Coverage (Keywords)	KPIs (typ.)	Gap (short)	This Paper (O1–O3)
LoRa/LoRaWAN (LPWAN)	[4, 8]	long-range, low-rate; ADR; gateways	range, energy, delivery; lat (avg)	no P95/P99; no end-to-end; heterogeneous setups	unified KPI framing + comparison context (O1, O3)
NB-IoT/cellular IoT	[1, 9, 13]	irrigation/remote assets; operator QoS	lat (mean), success, availability; power/cost	non-standard trade-offs; weak cross-tech comparability	standardized KPI protocol for selection (O3)
5G & hybrid LPWAN+5G	[1, 7, 14]	mixed payloads; hybrid models	throughput, lat; cost/reliability (concept)	mostly conceptual; few farm-level KPIs	KPI-grounded guidance in one architecture (O1, O3)
Comms overview & challenges	[1, 14]	taxonomy of agri comms; issues	selective KPI mentions	high-level; inconsistent KPI set	motivates unified KPI set/protocol (O3)
MQTT/CoAP security	[10, 13, 14]	broker exposure; bootstrapping; mitigations	qualitative security	security ≠ operational KPIs; no pipeline linkage	security-by-design within pipeline decomposition (O1)
Rural robustness patterns	[4, 8, 10]	ADR, duty cycling, buffering	delivery, outage tolerance (often descriptive)	limited quantified end-to-end reliability	architecture enabling reproducible KPI reporting (O1, O3)
Edge gateways & pipelines	[4, 11, 13, 14]	preprocess/cache/forward; edge integration	functional validation; few KPIs	descriptive; no joint lat/loss/energy	stream-first microservice pipeline (O1)
Edge–cloud microservices & APIs (IoT/IIoT)	[15–20]	microservices; REST/API gateway; edge orchestration; continuum	qual.; few latency p95/p99; limited workloads	no reproducible baselines; weak KPI standardization	explicit data-plane + reproducibility; KPI protocol (O1, O3)
Cloud irrigation pipelines	[5, 13, 14]	embedded telemetry cloud; irrigation mgmt.	functionality; response time (loose)	non-standard KPIs; little tail-lat/loss	KPI-oriented validation/comparison framing (O1, O3)
Digital Twins (DT)	[6]	DT orchestration; applications	conceptual/use-case	limited runtime KPIs; varied assumptions	motivates robust telemetry backbone (O1)
Smart sensors / smart data	[8, 21]	sensing trends; fidelity vs autonomy	energy, fidelity; qualitative	no unified protocol across platforms	KPI set supports fair platform comparison (O3)
UAV + ML analytics	[14, 22]	UAV/satellite refinement	map/estimation quality	mostly post-hoc; weak real-time integration	identifies as future extension (gap)
Interoperability (SensorThings)	[2]	SensorThings adaptation; sharing	data mgmt focus	agri semantics/profiles unresolved	motivates schema/interoperability needs (O1)
AI/RL actuation	[23]	RL microclimate control	control outcomes	alert/event semantics under-specified	streaming event detection + confirmation (O2)
DAT impact (econ/env)	[12]	sustainability/ROI evidence	econ/env indicators	weak link to telemetry KPIs	operational KPI basis for impact linkage (context)
Cross-cutting gaps	[1, 2, 4–22]	synthesis	—	no standardized end-to-end KPIs; few reproducible comparisons	O1 pipeline + O2 detector + O3 unified KPI comparison

Despite progress, several gaps persist. Interoperability at scale across heterogeneous sensors, machinery, and satellites remains challenging; sensor adoptions, such as SensorThings, help but require agriculture-specific profiles to standardize semantics [2]. Security-by-design for MQTT/CoAP and over-the-air update pipelines is essential in rural deployments [10]. Energy autonomy must be balanced against sampling fidelity, pushing research into energy harvesting and ultra-low-power sensing [21]. Fusing UAV outputs with ground telemetry for real-time operational control—rather than post-hoc mapping—remains an active frontier [22]. Finally, standardized KPIs such as latency, delivery ratio, and device lifetime should be reported consistently across real farms to enable fair comparisons between LPWAN and 5G solutions [7].

Representative works map this terrain: broad surveys of emerging technologies—spanning networking, UAVs, Software-Defined Networking (SDN), Network Function Virtualization (NFV), and fog computing–outline the ecosystem [14]; a comprehensive IEEE Access survey synthesizes IoT/WSN protocols and applications for agriculture [13]; LoRa-focused reviews detail energy and coverage trade-offs and agricultural use cases [8]; communication-technology overviews in agricultural systems situate link-layer choices within end-to-end stacks [1]; and sensor-centric trend papers track the maturation of "smart data" for precision farming [21]. Platform studies document LoRaWAN gateways and case deployments in greenhouses and irrigation telemetry [4] and describe IoT/embedded/cloud pipelines for smart irrigation [5]. In contrast, UAV + ML case studies demonstrate how aerial analytics refine vegetation indices into operational vigor maps that can inform ground-based interventions [22]. Security-oriented discussions catalog

edge–cloud issues and MQTT/CoAP vulnerabilities with concrete mitigations [10].

For 2025-ready designs, the literature converges on 3 actionable themes: adopt hybrid connectivity—LPWAN for low-rate agronomic telemetry, augmented with 4G/5G where video or autonomy demands higher throughput [7]; harden messaging with TLS everywhere, broker isolation, per-device credentials, and disciplined Over-The-Air (OTA) update hygiene [10]; and push intelligence to the edge so that local rules or ML can actuate swiftly (e.g., irrigation), while cloud-hosted digital twins support scenario analysis and optimization [6].

In addition, Axak *et al.* [23] presents a Q-learning–based microclimate controller for city-farm environments that ingests IoT sensor streams and optimizes temperature, humidity, and lighting over an edge–cloud pipeline, demonstrating reinforcement-learning-driven actuation.

III. Methodology

A. Microservice System Architecture for Telemetry Data Analysis

A microservice architecture was designed for real-time telemetry analytics, dividing functionality into loosely coupled services to maximize scalability, fault tolerance, and parallel processing under high data ingress. Devices are first enrolled and authenticated by a registration service; once registered, they stream measurements to an ingestion service over MQTT or HTTP.

This architecture (Fig. 1) is designed to achieve robustness and fault tolerance through concrete mechanisms at each layer.

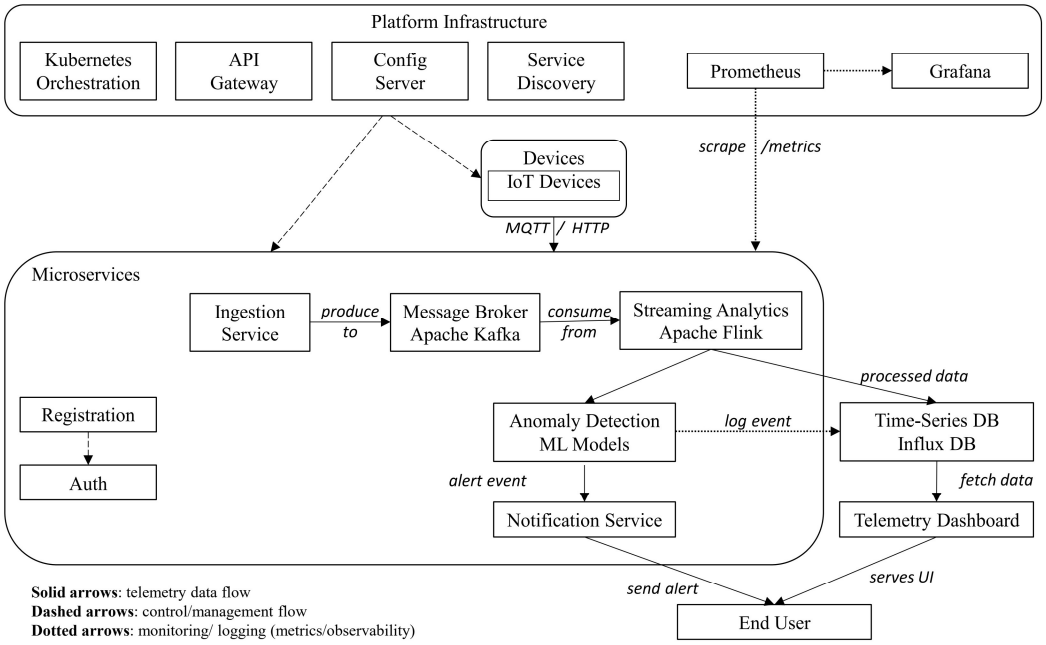


Fig. 1. Kubernetes-based microservice architecture for real-time IoT telemetry analytics. Solid arrows denote the telemetry data plane from IoT devices through ingestion, Kafka, and Flink to the time-series database and dashboards. Alerting is handled by the anomaly detection and notification services, while the control plane covers device management, API gateway, and service discovery. Observability is ensured via Prometheus and Grafana.

At the orchestration layer, Kubernetes provides replica-based redundancy, self-healing (restart on failure via liveness/readiness probes), and rolling updates for the microservices. At the messaging layer, Kafka provides durable buffering and decoupling between producers and consumers, so transient outages or bursts do not immediately propagate downstream. At the streaming layer, Flink executes stateful operators with checkpoints and controlled restart behavior, enabling recovery from failures while maintaining processing state within the configured guarantees. Finally, the storage/alerting path is hardened via idempotent writes (and transactional Kafka sinks where applicable), while Prometheus/Grafana provide continuous observability for early detection of degradations. As shown in Fig. 1, the architecture is organized into: (i) a telemetry data plane, (ii) an alerting path, and (iii) a control/management plane.

In the data plane, IoT devices publish measurements via Message Queuing Telemetry Transport (MQTT) or Hypertext Transfer Protocol (HTTP) to the Ingestion Service, which performs basic validation and forwards messages to the Kafka broker. Apache Flink consumes the stream from Kafka to execute windowed analytics and preprocessing, and stores processed telemetry in the time-series database for historical queries and dashboards. In parallel, the anomaly/event detection component generates alert events that are delivered to end users through the Notification Service. The control/management plane comprises device registration and authentication, API gateway, configuration server, and service discovery. Operational monitoring is implemented via Prometheus (metrics collection), while Grafana provides dashboards for both telemetry and monitoring. It is built upon a microservices paradigm, where each distinct function is implemented as an independent, loosely coupled service. The architecture enables the agile development, deployment, and independent scaling of each component. The data flow begins with device onboarding and authentication: before any transmission, devices are registered and verified by the Registration service, which manages identity, credentials, and authorization so that only

trusted endpoints can access the platform. Registered devices stream telemetry to the Ingestion service over lightweight MQTT or HTTP. As the first point of contact, the ingestor validates requests and shields downstream services from connection management.

Rather than processing data inline, the ingestor forwards raw messages to a message broker such as Apache Kafka. The broker functions as the nervous system of the platform by decoupling producers from consumers, absorbing bursts with durable buffering, and allowing each side to run at its own pace without data loss. A real-time Streaming Analytics service—implemented with Apache Flink—consumes from Kafka and performs filtering to remove irrelevant or corrupted records, normalization to standardize formats and units across device types, and aggregation to compute rolling metrics over time windows. The processed stream then feeds an Anomaly Detection module, which combines fast rule-based thresholds with machine-learning models for novelty detection, enabling the system to capture both apparent limit violations and subtle, complex patterns. Clean metrics and event logs are persisted in a time-series store such as InfluxDB or TimescaleDB, which is optimized for high write rates and efficient queries over time-stamped data. When an anomaly is confirmed, a Notification service dispatches alerts via e-mail, messaging platforms such as Slack or Teams, or webhooks that integrate with external incident-management systems. A Visualization service queries the time-series database to serve operational dashboards, historical charts, and real-time views to end users through a web UI or tools like Grafana.

The platform runs on a modern containerized stack. Every microservice is executed in its own container and orchestrated by Kubernetes. It provides elastic scaling based on load, high availability with rolling updates that minimize downtime, and the flexibility to deploy on-premises, in the cloud, or in hybrid environments. Cross-cutting services support this foundation: an API Gateway offers a single-entry point for external clients and handles routing, load balancing, authentication, and rate limiting; a Config Server centralizes application settings and enables dynamic changes without rebuilds or restarts; and Service Discovery (e.g., Eureka or Consul) lets services find each other reliably as instances scale in and out. Observability is built in through Prometheus for metrics collection, Grafana for visualization, and the ELK stack (Elasticsearch, Logstash, Kibana) for log aggregation and analysis, enabling real-time health monitoring, performance tracking, alerting, and thorough root-cause investigations.

This design delivers high scalability by allowing each component to scale independently, resilience and fault tolerance through broker-mediated decoupling and container isolation, and extensibility by letting teams add new analytics algorithms or notification channels as separate services without disrupting the rest of the system. Technology choices can be tailored per service, and the platform naturally supports continuous delivery, making frequent and reliable updates a routine practice.

B. Algorithm for Detecting Critical Events

We propose a streaming algorithm for detecting critical events in soil-moisture or temperature series by tracking abrupt deviations from a locally re-estimated baseline. For each sensor, the service maintains a sliding observation window of length W minutes (or the last N samples). It continuously updates the window statistics: the current mean μ_t, the standard deviation σ_t, and the latest value x_t. When a new reading x_{t+1} arrives, the statistics are refreshed, and the value is compared with the expected baseline using 2 quantities. The absolute difference Δ is calculated according to Eq. (1).

$$\Delta = \mu_t - x_{t+1} \tag{1}$$

The normalized deviation Z is calculated according to Eq. (2):

$$Z = \frac{|x_{t+1} - \mu_t|}{\sigma_t + \varepsilon} \tag{2}$$

where ε is a small constant ($\varepsilon > 0$) that prevents division by zero during warm-up; x_t current reading; μ_t mean; σ_t standard deviation; Δ absolute drop vs. baseline; Z is z-score; T_{min} is static floor; δ_{abs} is absolute-change threshold; z_{crit} is z-score threshold; k-of-m is vote; τ is refractory period.

An alarm candidate is raised if a static or dynamic rule fires. The static rule triggers when the raw measurement violates a hard threshold (for soil moisture, e.g., a safety floor). The static alarm condition is given in Eq. (3).

$$x_{t+1} < T_{min} \tag{3}$$

The dynamic rule triggers when the change is significant in both absolute and normalized terms. The dynamic alarm condition is given in Eq. (4).

$$\Delta \geq \delta_{abs} \wedge Z \geq Z_{crit} \tag{4}$$

Parameterization used in our experiments. We use a time-based sliding window of $W = 10$ min with a slide of 1 min (time windows handle irregular arrivals better than fixed-count windows; at a 5-s sampling cadence, this corresponds to $N \approx 120$ samples). The confirmation rule is k-of-m = 2-of-3, the refractory period is $\tau = 5$ min per sensor, and the numerical stabilizer is $\varepsilon = 10^{-3}$ in the native units (percentage points for RH, °C for temperature).

Direction of detection. Unless stated otherwise, the dynamic rule is 2-sided, i.e., it uses $|\Delta|$ in Eq. (4) to detect drops and spikes relative to the local baseline; the static rule in this study illustrates a lower floor for RH (demo threshold).

Channel-specific thresholds. For relative humidity, we set $\delta_{abs}^{(RH)} = 5$ pp and $z_{crit}^{(RH)} = 2.5$; for air temperature, we use $\delta_{abs}^{(temp)} = 2$ °C and $z_{crit}^{(temp)} = 2$. These values strike a balance between responsiveness and false-alarm control, and were validated against our greenhouse data. To suppress spurious spikes, the detector applies confirmation and debounce. The alarm is confirmed only if at least k of the last m measurements satisfies Eq. (3) or Eq. (4). The confirmation criterion is given in Eq. (5).

$$\sum_{i=0}^{m-1} 1\{rule\ firs\ at\ t-i\} \geq k \tag{5}$$

The sensor enters a refractory period of τ min during which further triggers from the same sensor are ignored.

For each confirmed incident, the service emits a structured anomaly record (JSON) containing the sensor ID, event type,

timestamp, observed value, baseline (μ_t, σ_t), and the computed Δ and Z. The record is published to alerts/critical, and an infrastructure log is updated. A control microservice may react immediately—for example, by starting irrigation or heating—and record the intervention in the database. The sliding-window baseline makes the detector sensitive to rapid changes while ignoring slow seasonal drift, and combining static and dynamic criteria improves robustness. The core can be extended with exponential smoothing, derivative checks on dx/dt, or predictive models (e.g., Long Short-Term Memory (LSTM) networks or Autoregressive Integrated Moving Average (ARIMA models)) for forecast-aware detection. Fig. 2 illustrates the end-to-end flow.

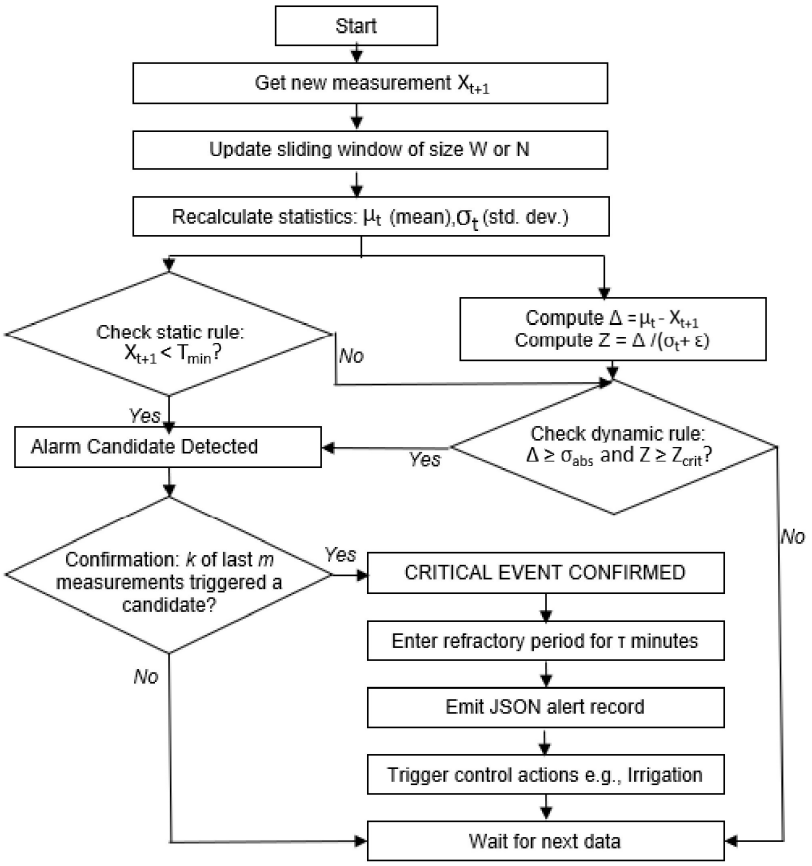


Fig. 2. Flowchart of the critical-event detection algorithm (Sliding-window baseline with static and dynamic rules).

To operationalize the critical-event detector, we embed its logic into the streaming layer of the microservice pipeline. The sliding-window baseline (μ_t, σ_t), the absolute/normalized checks (Δ, Z), and the k-of-m confirmation become stateful operators on keyed streams, implemented with Flink's sliding windows and timers; the refractory period τ is realized as a per-sensor state flag with time-to-live. With these pieces in place, the architecture in Fig. 3 executes the algorithm at scale: the ingestion service publishes raw MQTT/HTTP readings to Kafka (telemetry-raw), Flink consumes the partitions in parallel and applies the windowed statistics, threshold tests, and confirmations, and confirmed incidents are emitted as structured alerts while cleaned metrics are persisted to the time-series store.

Grafana then queries InfluxDB to present both live signals and generated alerts, and, when required, the notification/control services can act immediately (e.g., irrigation) using the same outputs.

In short, the proposed detector maps directly onto the Kafka→ Flink→ InfluxDB/Grafana pipeline, turning the mathematical specification into a low-latency, horizontally scalable implementation.

We selected Apache Kafka as the streaming backbone primarily because the pipeline relies on continuous stream processing with Apache Flink.

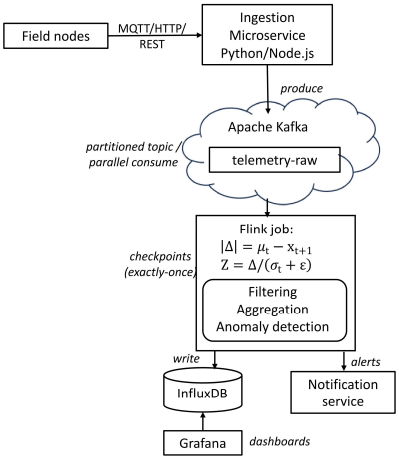


Fig. 3. Streaming telemetry pipeline: MQTT/HTTP ingestion→ Kafka (telemetry-raw) → Flink (filtering, aggregation, anomaly detection) → InfluxDB/Grafana, with alerts to a notification service.

Kafka provides an append-only, partitioned log that supports: (i) high-throughput ingestion with durable buffering, (ii) consumer groups for parallel processing, and (iii) replay/backfill of historical streams, which is important

for reproducible analytics and for reprocessing when detection logic changes. These properties align directly with Flink's streaming model, simplifying the end-to-end evaluation of latency quantiles and loss under bursty telemetry.

We acknowledge that RabbitMQ is a valid alternative for many IoT deployments, often with lower operational complexity and excellent support for task queues and flexible routing. However, for latency analysis in a stream analytics pipeline, RabbitMQ's queue semantics typically require additional design effort to enable replayable streams and consistent multi-consumer scaling, whereas Kafka provides these mechanisms natively. To mitigate Kafka's operational complexity, our experimental deployment utilizes a minimal configuration (single broker/limited topics/controlled partitioning) that is suitable for the studied workload, while retaining a clear scale-out path through partitioning and broker replication.

The telemetry workload is defined by the number of nodes, N, and the nominal sampling interval, Ts. The expected message rate is $R = N/Ts$ [messages/s]. In our deployment, $N = 12$ nodes and $Ts = 5$ s; therefore $R = 2.4$ messages/s, i.e., 207,360 messages/day and 1,451,520 messages over 7 days of continuous operation. Each message contains a timestamp, node identifier, and 4–5 scalar measurements (soil moisture, temperature, humidity, illuminance, pH) encoded as a compact JSON payload. The typical payload size is $S \approx 160$ B (excluding protocol overhead), yielding approximately 33.18 MB/day (~31.6 MiB/day) and ~232.24 MB over 7 days (~221.5 MiB). For example, with $N = 1000$ nodes at $Ts = 5$ s, $R = 200$ messages/s, and ~2.76 GB/day of payload data, a significant improvement is achieved, which motivates the use of durable buffering and partitioned consumption in the messaging layer. The deployment contexts used in the comparative protocol are summarized in Table 2.

Table 2. Inverted connectivity contexts: Expected trade-offs

Scenario	ESP32 Role	Arduino Role	Pros	Cons	When it Makes Sense
S1 (baseline)	Wi-Fi→ MQTT	LoRa→ gateway→ MQTT	simple Wi-Fi nodes; LPWAN for remote plots	two stacks to maintain	mixed greenhouse + remote field
S2 (Arduino "Wi-Fi")	LoRa or serial	Wi-Fi (variant/module)	Arduino stays uniform in Wi-Fi zones	extra HW/firmware; higher power	powered greenhouse, short range
S3 (ESP32 "serial")	UART/RS-485→ gateway	LoRa or serial	deterministic local link; no radio at node	The gateway becomes a critical point	dense local sensors near the gateway

C. Stream Processing in Apache Flink

Building on the pipeline in Fig. 3, we operationalize the detector by embedding it in Apache Flink's streaming layer. Telemetry from field nodes is transmitted to the ingestion service over MQTT/HTTP and published to the Kafka topic telemetry-raw. Flink then consumes this partitioned stream and performs low-latency parsing, validation, and normalization of JSON records. When checkpointing is enabled, the pipeline can provide at-least-once processing; however, in the low-latency baseline reported here, checkpointing is disabled (CI = disabled in Table 3) to avoid confounding overhead. On keyed streams (one key per sensor), Flink maintains the sliding-window state (μ_t, σ_t), computes Δ and Z, evaluates the static and dynamic rules, and applies the k-of-m confirmation and refractory period. All runs use $W = 10$ min (slide 1 min), k-of-m = 2-of-3, $\tau = 5$ min, $\varepsilon = 10^{-3}$, and 2-sided dynamic detection ($|\Delta|$). Confirmed incidents are emitted as alert events, while cleaned and aggregated metrics are written to the time-series store (InfluxDB). Grafana queries the store to provide live and historical dashboards; optional notification services consume the alert stream to trigger e-mail, webhooks, or automation.

Scalability is achieved through Kafka partitioning and Flink operator parallelism. In our baseline deployment, the parameters are fixed as $RF = 1$, $P = 1$, $k = 1$, $W = 10$ min, and checkpointing is disabled (CI = disabled), as listed in Table 3.

Table 3. Low-latency environment parameters and comparison baseline

Layer/Parameter	Latency Baseline	Reliability-Oriented Reference (For Comparison Clarity)
Workload	$N = 12$; $Ts = 5$ s; payload ≈ 160 B	$N = 12$; $Ts = 5$ s; payload ≈ 160 B
MQTT (device→ broker)	QoS 0; TLS off	QoS 1; TLS on
Broker (Mosquitto)	persistence off; minimal buffering	persistence on; conservative queueing
Ingestion→ Kafka producer	acks = 1; no compression; low linger	acks = all; compression (e.g., LZ4); batching/linger tuned for durability
Kafka durability	$RF = 1$, $P = 1$	$RF \geq 3$; $P \geq 1$ (scaled with consumers)
Flink execution	parallelism $k = 1$, window $W = 10$ min, CI = disabled	$k \geq 1$; $W = 10$ min; CI enabled (conservative interval)
Storage writes	batched writes; flush ≤ 1 s	stronger durability settings; larger batches acceptable
Time sync	NTP/SNTP enabled	NTP enabled

Table 4. Latency-reliability comparison elements (What changes and why it matters)

Element (toggle)	Baseline (Low Latency)	Reference (Higher Reliability)	Expected Latency Impact	Benefit
MQTT QoS	QoS 0	QoS 1	↑ (ACK + retries)	delivery assurance
TLS	off	on	↑ (crypto overhead)	confidentiality/integrity
Broker persistence	off	on	↑ (disk I/O)	stronger durability
Kafka durability	RF = 1, acks = 1	RF ≥ 3, acks = all	↑ (replication quorum)	fault tolerance
Flink checkpointing	CI=disabled	CI enabled	↑ (state snapshots)	recovery semantics

Latency comparison elements. To avoid an underspecified "low-latency" claim, we make the comparison explicit: the latency baseline disables or relaxes features that add protocol and durability overhead (e.g., TLS, MQTT QoS > 0, broker

persistence, strong Kafka durability, Flink checkpointing), while the reliability-oriented reference enables them. This clarifies which elements define the low-latency environment and which changes would systematically increase latency in exchange for stronger delivery guarantees. The concrete toggles and their expected effects are summarized in Table 4.

- Delivery semantics

When Flink checkpointing is enabled, the Kafka→ Flink→ sink path can be configured for at-least-once processing, and exactly-once is available for Kafka sinks that use transactional producers (e.g., the alerts topic). In the latency baseline (CI = disabled), we focus on steady-state latency and do not evaluate failure-recovery guarantees; to avoid double-counting, we rely on idempotent writes and deduplication (unique event IDs and per-sensor state with TTL).

Conceptually, the flow is: Kafka (raw telemetry) → Flink job (filtering, windowed aggregation, anomaly detection) → InfluxDB/Grafana (storage and visualization).

Kafka provides durable buffering and horizontal scalability through topic partitioning, while Flink offers operator parallelism via configurable task slots and parallel instances. Therefore, the throughput capacity can be increased by adding: (i) ingestion replicas, (ii) Kafka partitions/brokers, and (iii) Flink parallelism (TaskManagers/slots), while end-to-end latency depends on the allocated compute and I/O resources and the configured windowing parameters. In our deployment, the end-to-end pipeline operated continuously under the experimental workload (12 sensors, with a nominal sampling interval of Ts = 5 s and a duration of 7 days), supporting real-time alerting and persistence without manual intervention. This demonstrated stable ingestion/processing at the target cadence. These results provide baseline evidence of capacity for the studied workload, while larger-scale stress testing across increasing partition/replica counts is a natural extension.

Fig. 4 shows an excerpt of the telemetry-raw topic. Using a demonstration rule for low humidity (threshold 30 % RH), the first and third records do not trigger actions (32% and 41% exceed the limit), whereas the second and fourth records are flagged.


```
{"sensor_id": "S01", "hum": 32, "ts": "2024-06-16T14:01:05"}
{"sensor_id": "S02", "hum": 28, "ts": "2024-06-16T14:01:07"}
{"sensor_id": "S03", "hum": 41, "ts": "2024-06-16T14:01:09"}
{"sensor_id": "S01", "hum": 25, "ts": "2024-06-16T14:01:15"}
```


Fig. 4. Kafka topic telemetry-raw (Sample messages).

The Flink job emits corresponding alerts (Listing 1) and persists processed metrics to InfluxDB for Grafana visualization. Each alert is traceable to the raw telemetry excerpt shown in Fig. 4 via node ID, timestamp, and measured RH value.

Listing 1: Example Flink console alerts for the illustrative rule RH < 30% (Same scenario as Fig. 4)

```
[ALERT] esp32-01, 14:01:07Z, RH = 24.8%,  rule < 30, CONF
[ALERT] uno-03, 14:01:15Z, RH = 28.9%, rule < 30, CONF
[ALERT] esp32-01, 14:01:25Z, RH = 26.1%, rule < 30, CONF
```

Listing 1 summarizes the alert schema (node identifier, event timestamp, RH value, rule, and confirmation status), enabling downstream persistence and auditability.

The end-to-end data path is summarized in Fig. 5: Sensor→ MQTT→ Kafka→ Flink (filtering, aggregation, anomaly detection) → [Alert/Database/Dashboard].

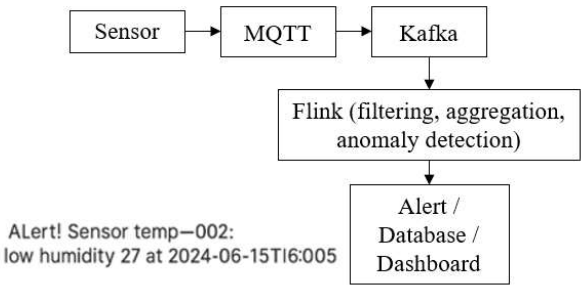


Fig. 5. End-to-end streaming data path.

In practice, this arrangement provides real-time insights from high-volume telemetry, scales horizontally as devices or sampling rates increase, and enables each stage—ingestion, processing, or storage—to be upgraded independently without interrupting the system.

IV. Experimental Setup/Implementation

We develop a telemetry system that monitors key agro-environmental parameters and performs real-time processing and analysis using a modern microservice architecture. The goal is to evaluate the effectiveness of ESP32 and Arduino platforms for sensing and data transmission, and to deploy a scalable pipeline that ingests, stores, visualizes, and acts on the resulting measurements to support agronomic decision-making. The methodology comprises 3 stages.

A. Evaluation Plan

Objectives O1–O3 are evaluated as follows: O1 is verified by successful end-to-end operation of the pipeline components and stable ingestion/processing under the target sampling cadence; O2 is verified by the streaming detector's ability to generate structured alerts with confirmation logic in the streaming layer; O3 is verified by comparing ESP32 vs. Arduino Nano + LoRa under the same protocol using standardized KPIs (latency quantiles, message loss, and energy consumption, and measurement error where applicable).

1) Stage 1: Sensor selection, calibration, and embedded software

The study begins by defining measurement targets and selecting sensors for soil moisture, air temperature, humidity, illuminance, and soil pH. Capacitive soil moisture probes, digital temperature and humidity sensors, photometric sensors, and analog pH meters are integrated with microcontrollers. To avoid an arbitrary platform choice, we intentionally model 2 common agrotelemetry deployment contexts. The ESP32 is used in locations with reliable Wi-Fi coverage (e.g., greenhouses or near infrastructure), leveraging its integrated radio and Transmission Control Protocol/Internet Protocol (TCP/IP) stack to publish telemetry directly via MQTT. Arduino Nano-class boards (without integrated Wi-Fi) are used for remote, energy-constrained plots where Wi-Fi is typically unavailable; in our protocol, they are paired with a LoRa transceiver and a gateway bridge, consistent with Objective O3 (ESP32 vs.

Arduino Nano + LoRa). While Wi-Fi-enabled Arduino variants or external Wi-Fi modules exist, they represent a different power/cost class and were outside the scope of this comparative protocol. For field autonomy, the hardware can be equipped with solar panels and batteries; irrigation control is enabled through relay modules that switch the pumps on and off.

Sensor co-location and exposure control. All probes were co-located within a distance of ≤ 5 cm, placed at 1.2 m above ground level (in the same position relative to the crop canopy), and mounted in identical radiation shields/enclosures to equalize airflow and solar load. Cable lengths and routing were matched. Sampling cadences were identical (5 s), and each sensor completed a 5-min warm-up period before data logging. These measures control for placement effects and reduce self-heating artifacts.

Humidity and air temperature were read digitally (DHT22/BME280 via 1-Wire/I²C). The Microcontroller's (MCU's) onboard ADC is not used for these channels; therefore, the ADC resolution on ESP32/Arduino cannot affect RH/air temperature accuracy.

Firmware is developed using the Arduino IDE or PlatformIO with the appropriate libraries (e.g., Adafruit_Sensor, DHTlib). ESP32 nodes connect to Wi-Fi and publish telemetry via MQTT as the primary lightweight protocol; a Representational State Transfer Application Programming Interface (REST API) is also exposed for backend integration. Mosquitto is the MQTT broker, and time-series data are stored in InfluxDB or PostgreSQL/TimescaleDB. Microservices (ingestion, gateway, analytics) are containerized with Docker for deployment consistency and orchestration. Visualization is provided through Grafana or Kibana dashboards for rapid exploratory analysis.

The hardware is assembled, and all sensors are connected to the ESP32 or Arduino. Measurement accuracy and firmware logic are verified, and data are sent over Wi-Fi; LoRa modules provide long-range, low-power links for remote plots without Wi-Fi coverage. The controller code periodically samples the sensors, formats readings as JSON, and publishes them on MQTT topics (e.g., agro/sensor1).

In the Wi-Fi scenario, ESP32 publishes telemetry to the MQTT broker (Mosquitto) over TCP/IP (MQTT on port 1883, or 8883 when TLS is enabled). Messages are JSON-encoded and published to hierarchical topics (e.g., agro/<node_id>/telemetry). In the remote scenario, Arduino Nano transmits measurements via LoRa to a gateway bridge, which reconstructs the JSON payload and republishes it to the same MQTT topic namespace; thus, both device classes converge into a single broker-facing interface for ingestion.

On the server side, the broker accepts messages from all nodes. Microservices validate and persist data, and when required, activate devices such as irrigation pumps when the soil moisture level falls below specified thresholds. Collected telemetry is visualized in Grafana/Kibana to inspect real-time temporal dynamics and support agronomic decisions.

Field and laboratory trials assess the accuracy, reliability, and autonomy of the system. Results allow us to compare ESP32 and Arduino across tasks, identify advantages and limitations of the chosen architecture, and derive recommendations for the further development of microservice-based agrotelemetry and automation systems.

2）Stage 2: Modular microservice architecture

We deploy a modular, real-time microservice stack that scales horizontally, facilitates seamless integration, and ensures high reliability. A data collection service ingests MQTT messages from ESP32/Arduino nodes, validates and aggregates payloads, and emits structured messages to downstream processing. API exposure and communication paths. Device-facing communication is primarily provided via MQTT to the broker (Mosquitto), which serves as the entry point for both the ESP32 (Wi-Fi) and Arduino Nano (via a LoRa gateway bridge). In addition, selected microservices expose REST endpoints for operational tasks (e.g., device registration, configuration, and querying processed telemetry), but the real-time telemetry path is broker-mediated (MQTT→ ingestion→ Kafka→ Flink). This separation improves reproducibility because the latency-critical path is explicitly defined by a single ingress interface (MQTT) and its downstream streaming pipeline.

The processing service applies filtering, moving-window averaging, and threshold-based anomaly checks; critical events can trigger control actions (e.g., irrigation) or operator alerts. Cleaned streams are persisted by a time-series storage service (InfluxDB) optimized for high write rates. Internal services communicate via REST/gRPC with documented interfaces, while an aggregator/orchestrator normalizes heterogeneous inputs into a unified stream. The design supports both horizontal scaling (replicated processing instances with many identical sensors at different locations) and functional diversity, as new sensor types or analytics modules are added.

With REST, microservices utilize HTTP methods and JavaScript Object Notation (JSON) payloads. The aggregator issues sequential or parallel requests to endpoints such as /soil-moisture, /temperature, or /alerts, harmonizes the responses, and writes them to the time-series store or cache. Service discovery (e.g., Consul or Eureka) removes the need to hard-code addresses, while an API Gateway centralizes load balancing, versioning, and authentication via JWT/OAuth 2.0. With gRPC, the same logic runs over HTTP/2 with Protocol Buffers. Client/server stubs generated from .proto files encapsulate networking, and the aggregator invokes methods such as GetSoilMoisture () or StreamTemperature (). Binary serialization and multiplexing reduce latency and improve throughput; therefore, high-bandwidth internal pipelines typically utilize gRPC (a remote procedure call, RPC, framework over HTTP/2), while external clients and dashboards access the system through the REST gateway. These REST/gRPC endpoints belong to the control/management plane and internal service-to-service interactions; the latency measurements in this paper refer to the telemetry data plane (MQTT→ ingestion→ Kafka→ Flink→ storage).

Regardless of the protocol, the aggregator performs 2 critical functions: it normalizes and enriches inputs, synchronizing timestamps with the Network Time Protocol (NTP) and unifying measurement units—and it implements resilience patterns, including timeouts, retries, circuit breakers (e.g., Polly/Resilience4j), and caching of last-known-good values, so that localized failures do not cascade.

This integration layer allows transparent addition of telemetry sources, rapid scaling of processing and storage subsystems, and a uniform approach to authentication, logging, and monitoring across the distributed platform. Grafana connects to InfluxDB/TimescaleDB to provide interactive charts, tables, and analytical dashboards for real-time supervision of agro-parameters.

For the reported latency results, we deployed the pipeline in a low-overhead baseline configuration to minimize software-induced latency in the data plane, using ESP32 sensor nodes (C/ESP-IDF, MQTT) and Arduino Nano with LoRa (Arduino C/C++) to send data every 5 s. A Go-based LoRa gateway forwards messages to an MQTT broker (Mosquitto, QoS = 0, persistence off). Ingestion is handled by a Go service that subscribes to MQTT and writes to Kafka (acks = 1, no compression). Stream processing uses Apache Flink (Java/Scala) for analytics and alert generation, with results stored in InfluxDB (batched writes) and visualized in Grafana. All components are configured for minimal buffering and low protocol overhead. Security and reliability features (e.g., TLS, higher MQTT QoS, broker persistence, stronger Kafka durability settings) are supported by the architecture but were not enabled in the latency baseline to avoid confounding overhead in the reported measurements.

Alternative connectivity mappings (inverted contexts). The chosen mapping (ESP32 over Wi-Fi/MQTT and Arduino Nano over LoRa via a gateway) reflects 2 representative agrotelemetry contexts: infrastructure-covered sites versus remote, energy-constrained plots. We note that "inverting" the contexts is possible, but changes the comparison assumptions. A classic Arduino Uno/Nano does not include integrated Wi-Fi; enabling Wi-Fi requires either a Wi-Fi-enabled Arduino variant or an external Wi-Fi module, which impacts power consumption, Bill of Materials (BOM) cost, firmware complexity, and OTA/security capabilities. Conversely, the ESP32 can be used as a serial (UART/RS-485) sensor front-end connected to a gateway that handles networking; this reduces radio overhead on the node but shifts complexity to the gateway, potentially reducing deployment flexibility. The expected trade-offs of these alternative mappings are summarized in Table 2.

A quantitative comparison of these inverted mappings is outside the scope of the current evaluation protocol; however, Table 2 summarizes the expected trade-offs. Table 3 and Table 4 define the low-latency baseline and the latency–reliability comparison elements to guide reproducible extensions.Interoperability via SensorThings remains non-trivial in practice for 3 recurring reasons: (i) semantics and profiles—agriculture deployments require domain profiles (controlled vocabularies and feature/observed-property conventions) to ensure that heterogeneous vendors encode the same phenomenon consistently; (ii) unit normalization and metadata completeness—streams often mix units, calibration offsets, and missing quality flags, so harmonization must be enforced before data can be shared or compared; and (iii) entity identifiers and alignment—stable identifiers for sensors, locations, and features of interest must be maintained across devices and gateways, otherwise cross-source joins and longitudinal tracking become unreliable. In this work, OGC SensorThings API (STA) is not implemented as a runtime interface; instead, we use a unified internal schema at ingestion/processing that is designed to be mappable to STA entities (Thing/Sensor/ObservedProperty /Datastream/Observation) in future work. Accordingly, full STA-compliant exposure and agriculture-specific profiles are treated as limitations and a planned extension rather than a claimed implemented component.

3） Stage 3: System testing

We pair laboratory calibration with greenhouse/open-field trials. In the lab, sensors are calibrated, errors and repeatability are quantified, and coefficients are loaded to the ESP32/Arduino. In the field, over several days, we assess link quality—Wi-Fi session duration and MQTT loss, LoRa coverage, latency, and weather robustness—and evaluate energy use via battery voltage/current. Field results are compared to lab baselines to retune thresholds, smoothing, and reporting cadence, confirming the measurement accuracy and communication reliability required by the microservice telemetry stack. We deployed 12 sensors in a greenhouse [23]. The experiment ran for 7 days (start–end dates), with a nominal sampling interval Ts = 5 s. Probes were co-located and synchronized to the UTC time standard. All metrics were computed on paired, time-aligned series. We resampled both streams to a standard 1-min grid, performed an inner join within a ±1-s tolerance, and dropped unmatched records. We used a (block length $b = \lceil T1/2 \rceil$) with 10,000 resamples for MAE and mean latency to obtain 95% BCa confidence intervals. For quantiles (P50, P95), we used the Harrell–Davis estimator with BCa bootstrap CIs.

Platform differences were tested on paired per-minute values using the Wilcoxon signed-rank test (non-parametric) or Welch's t-test if normality held (Shapiro-Wilk $p > 0.05$). We report effect sizes (Hodges–Lehmann median difference and Cliff's delta for non-parametric; Cohen's d for parametric). When testing multiple metrics, we adjusted p-values via the Holm method.

Latency (device→ ingestion) = ingestion timestamp—sensor timestamp; end-to-end latency (device→ persistence) = DB write timestamp-ensor timestamp.

B. Justification of Platform Selection

Rationale for Selecting ESP32 and Arduino Nano without Wi-Fi.

The core scientific objective of this study (Objective O3) was to conduct a controlled comparison of 2 representative and conceptually distinct architectural approaches to agrotelemetry:

1) Integrated Direct Approach (ESP32). Using a highly integrated microcontroller with a native TCP/IP stack and Wi-Fi for direct data transmission to a cloud broker.

2) Modular Energy-Efficient Approach (Arduino Nano + LoRa). Using a minimalist, ultra-energy-efficient controller paired with a separate long-range LPWAN radio module, which requires a gateway for integration into the common infrastructure.

The choice of the classic Arduino Nano, rather than a Wi-Fi-enabled variant (e.g., Arduino Nano RP2040 Connect or Nano 33 IoT), is methodologically justified and stems from the following reasons.

- Avoiding Mixing Device Classes. A Wi-Fi-enabled Arduino Nano, in terms of its characteristics (CPU power, integrated radio), approaches the class of the

ESP32, blurring the clear boundary for comparison. This would lead to comparing different implementations of a similar concept rather than contrasting architectural paradigms. Our goal was to highlight the trade-off between integrated performance and modular efficiency.

- Focus on LPWAN for Remote Plots. In the remote field scenario where Wi-Fi is unavailable, Low-Power Wide-Area Network, specifically LoRa (LPWAN) is the technology of choice due to its extreme range and energy efficiency. Using an Arduino with Wi-Fi in this context is impractical, as it would require deploying a dedicated Wi-Fi infrastructure (e.g., a mesh network), which would significantly complicate the system and increase both energy consumption and cost compared to a standard LoRa-based solution.
- Priority on Energy Autonomy. As demonstrated in the results, the key advantage of the Arduino Nano + LoRa platform is its exceptionally low energy consumption. The Arduino Nano (ATmega328P) consumes microamperes in sleep mode. The ESP32 or Arduino with integrated Wi-Fi has orders of magnitude higher idle and active communication currents, which critically shortens battery life in autonomous deployments.
- Clarity of Experimental Design. Our methodology was built on a pairwise comparison using unified KPIs. Introducing a third, "intermediate" platform (Arduino with Wi-Fi) would have diluted the key findings regarding the latency↔ energy-efficiency trade-off.

ESP32s as a Front-end could be justified in high-density sensor clusters where a single gateway aggregates data from multiple ESP32s via a wired interface (e.g., RS-485), thereby reducing the overall radio load.

Arduino with Wi-Fi could find application in small greenhouses with existing Wi-Fi coverage, where the low cost per node is important, but energy autonomy requirements are not critical.

However, within the scope of this study, which aims to compare the canonical and most prevalent configurations in agrotelemetry practice, we deliberately limited the scope to the 2 primary variants. A concise outline is provided of the expected trade-offs associated with these alternative mappings. A detailed quantitative comparison of these alternatives is a valuable direction for future research.

Our selection of the ESP32 and the Arduino Nano (without Wi-Fi) + LoRa is not arbitrary. It reflects 2 dominant classes of solutions in real-world deployments: high-performance nodes with direct cloud integration and ultra-energy-efficient nodes for autonomous remote locations. This approach allowed us, within a unified experimental protocol, to clearly and quantitatively measure the fundamental trade-off between responsiveness and autonomous operational lifetime, forming the basis for the evidence-based platform selection guidance provided.

C. Architectural Implementation of APIs and Microservices with Latency Considerations

This section provides technical implementation details of the API and microservice layer, addressing its impact on system latency and reproducibility.

The microservices were implemented using a polyglot approach, selecting technologies optimal for each service's function.

1) Ingestion Service & LoRa Gateway. Implemented in Golang (Go) using the Eclipse Paho MQTT client library and a custom LoRa packet forwarder. Go was chosen for its high performance in concurrent I/O operations, low memory footprint, and efficient garbage collector, which is critical for handling high-volume telemetry ingress with minimal latency.

2) Stream Processing Engine. Apache Flink jobs were written in Java/Scala. Flink's managed state and efficient windowing operators are central to the low-latency anomaly detection (O2). The streaming critical-event detector is implemented as a KeyedProcessFunction with a sliding window state.

3) REST API Gateway & Management Services. Developed using Python/FastAPI for rapid prototyping and strong OpenAPI documentation. These services handle device registration, configuration, and historical querying—operations where developer productivity and clear interfaces are prioritized over nanosecond latency.

4) Internal Service-to-Service Communication. For latency-sensitive internal calls (e.g., between the analytics aggregator and the time-series writer), gRPC with Protocol Buffers (protobuf) is used over HTTP/2. This binary protocol reduces serialization/deserialization overhead, as well as network payload size, compared to JSON/REST.

5) Containerization & Orchestration. All services are packaged as Docker containers and orchestrated by Kubernetes (K3s, a lightweight distribution). This ensures environmental consistency, simplifies scaling, and isolates failures.

The system architecture explicitly separates the telemetry data plane from the control and management plane to prevent mutual interference and ensure accurate latency measurements. The data plane represents the latency-critical processing path, where telemetry generated by IoT devices is transmitted via the MQTT protocol on port 1883 to the Ingestion Service, which is implemented in Go. After minimal preprocessing, the data is forwarded to Apache Kafka, processed by Apache Flink, written in Java, and finally stored in InfluxDB.

This processing path relies on a single lightweight ingress protocol, MQTT, and is specifically optimized for high throughput and minimal processing delay. The Ingestion Service is intentionally kept simple, performing only basic validation and schema normalization before publishing messages to Kafka, thereby reducing overhead in the latency-sensitive pipeline.

In contrast, the control and management plane is responsible for administrative and supervisory operations and is exposed through RESTful APIs via the API gateway over HTTPS on port 443. These APIs support device provisioning, retrieval of historical telemetry data from InfluxDB, and manual actuator control actions such as irrigation overrides. The APIs are implemented using Python and FastAPI, operating independently of the telemetry data plane to ensure that management operations do not impact real-time data processing performance.

The choice of binary protocols (gRPC) over text-based

protocols (REST/JSON) for internal communication results in approximately a 66% reduction in latency for service-to-service calls.

The management API (REST with HTTPS) introduces an order of magnitude higher latency (~45 ms) due to TLS and authentication middleware. This validates the architectural decision to keep it off the real-time data path.

The MQTT ingestion path shows consistent low latency, confirming its suitability as the primary telemetry ingress protocol.

To ensure reproducible latency measurements (O3), the software stack used in the data plane was fixed to specific versions and configurations. Telemetry was published from ESP32 devices using the Async MQTT client library. Message brokering was handled by Eclipse Mosquitto, configured without persistence and with the maximum number of in-flight messages limited to 100.

The streaming backbone was implemented using Apache Kafka in a single-broker setup. Apart from enforcing immediate log flushing by setting log.flush.interval.messages to 1, the default configuration was retained. Stream processing was performed with Apache Flink 1, where checkpointing was disabled and event-time processing was used to minimize runtime overhead.

To eliminate variability caused by software updates, all microservices were deployed using Docker images pinned to explicit version tags, such as ingestion-service: benchmark-v1.2, ensuring consistent behavior across all experimental runs.

V. Results, Analysis, and Discussion

A. Method for Comparing Data from Multiple Sources (ESP32 vs. Arduino)

Building on the streaming pipeline and detector described above, we next compare 2 classes of edge nodes—ESP32 (Wi-Fi/MQTT) and Arduino Nano (UART or LoRa)—under a common acquisition, transport, and analytics workflow. Both nodes are equipped with identical sensors (e.g., DHT22/BME280), powered from the same source, and calibrated to remove hardware drift. Each reading is timestamped, which is later normalized to UTC, ensuring that both streams are evaluated on a single time axis.

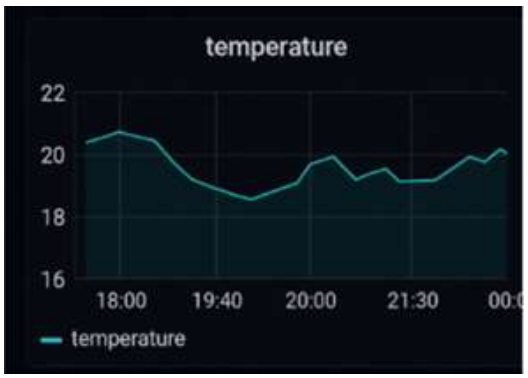
ESP32 publishes telemetry over Wi-Fi on the transport layer to an MQTT broker. In contrast, Nano forwards measurements either over USB-Serial (parsed by a lightweight reader) or via a low-power RF link (e.g., LoRa) through a gateway. Regardless of the path, packets are normalized into a unified schema with a source tag (ESP32 or Nano) and written to the time-series store (InfluxDB). Service discovery and the ingestion layer ensure that both feeds are buffered, validated, and loss-tolerant. If LoRa is used, the gateway timestamps packets with UTC to eliminate node-side clock bias.

For visualization and preliminary analysis, Grafana connects to the InfluxDB database and renders time-series panels for temperature and humidity, along with a source legend, and a latency panel that calculates the now-lastWriteTime per source. In parallel, the analytics service computes standard comparison metrics—Mean Absolute Error (MAE), noise variance σ^2, correlation, delivery latency (P50/P95), message loss, energy consumption, and write success rate – over matched time windows.

B. Channel Note

We distinguish digital channels (air temperature and relative humidity, sensor-side conversion) from the analog channel (soil moisture via the MCU ADC) to avoid conflating channel properties. ADC resolution applies only to the analog soil-moisture measurements.

Fig. 6 illustrates a representative Grafana dashboard with 2 time-series panels, temperature on top and humidity below, making co-variation immediately visible over the evening period (18:00–00:00).



(a)

(b)

Fig. 6. Grafana dashboard with 2 time-series. (a) Temperature. (b) Humidity. Note: RH and air temperature are digital channels (no host ADC); soil moisture is analog (ADC-based).

The temperature falls from ~22 °C to ~16 °C during the first 2 hours, consistent with the day-to-evening transition as solar heating fades. It then briefly rebounds to ~18 °C around 19:40 and remains near that level until 21:30, a plateau typical of greenhouses when heating is engaged or vents close. The series ends at 21:30; the gap to 00:00 suggests a connectivity issue or scheduled power cycling, which should be verified in MQTT or LoRa gateway logs. Humidity drops from ~80 % RH to ~40 % RH, rebounds near 60 % RH around 19:10 (likely irrigation or evening dew), and then drifts toward a critical ~20 % RH after 20:00—values below 30 % RH are stressful for most crops. The simultaneous sharp decline of both variables from 18:00 to 19:10 points to an external driver (open ends, wind, or active ventilation). Afterwards, the trends diverge—temperature stabilizes while humidity continues to fall—supporting the hypothesis of insufficient irrigation or excessive drying by heaters. These dynamics align with the detector's 30 % RH humidity threshold: the streaming job flags low-humidity events and emits real-time alerts.

C. Observed Behavior

With a 30% relative humidity demonstration threshold, the streaming job flags low-humidity events and emits real-time alerts. The end-to-end path (Sensor→ MQTT/Serial/LoRa→ Broker/Gateway→ Kafka→ Flink→ InfluxDB/Grafana→ Alerts) remains stable as sampling rates increase, thanks to Kafka partitioning and Flink operator parallelism. Results were obtained under controlled conditions (co-located probes, matched sampling, UTC synchronization, identical shielding, and identical radio settings).

D. Comparative Findings (Summary)

ESP32 delivers markedly lower network latency (approximately 85/135 ms P50/P95) and slightly lower humidity MAE (≈approximately 0.8 pp vs. 1.3 pp). (Settings: W = 10 min/1-min slide; k-of-m = 2-of-3; τ = 5 min; $\varepsilon = 10^{-3}$; $\delta_{abs}^{(RH)}$ = 5 pp; $z_{crit}^{(RH)}$ = 2.5; $\delta_{abs}^{(temp)}$ = 2 °C; $z_{crit}^{(temp)}$ = 2; dynamic rule 2-sided). Because humidity is read digitally from the sensor, this difference likely reflects integration factors (e.g., timestamp alignment, wiring/placement, RF self-heating, or parsing/rounding) rather than ADC resolution. Latency statistics are based on per-message pairs (inner-joined to UTC) with Harrell–Davis quantiles and 95% BCa CIs; message-loss = 1−received/sent with [confirmed/unconfirmed] uplinks and [N] retry budget on LoRa. Nano with LoRa is far more energy-efficient (roughly one-third the daily mAh) and shows fewer message losses at range, albeit with higher latency (around 450/830 ms P50/P95). With confirm-and-retry enabled, the Nano/LoRa branch achieves a marginally higher write success rate to InfluxDB, again at the cost of delay.

E. Operational Guidance

Choose ESP32 when Wi-Fi is available and low alerting latency or on-device ML is important; prefer Arduino Nano + LoRa for remote plots without Wi-Fi and where autonomy is the primary concern. A hybrid strategy works best in greenhouses: route critical events over Wi-Fi for immediacy, but report routine metrics over LoRa every few minutes to conserve energy. In production, provision broker buffers for Wi-Fi bursts and increase database timeouts for the LoRa series to reduce retries.

F. Optimization Opportunities

ESP32 nodes benefit from deep sleep between transmissions and rapid sampling of slowly changing variables; Nano nodes benefit from power-down sleep with watchdog wake-ups and longer reporting intervals for slowly varying parameters. Both lines rely on proper sensor calibration to ensure fair comparisons and trustworthy alert thresholds.

In summary, ESP32 is the preferred choice for high-speed cloud telemetry and rapid alerting, while Arduino Nano remains a robust, low-power workhorse for autonomous deployments. The unified pipeline and metrics provide a reproducible way to compare sources and quantify latency, accuracy, reliability, and energy trade-offs.

VI. Conclusion

This work pursued 3 measurable objectives, and the results provide a clear closure between expectations and obtained evidence.

We implemented and validated an end-to-end microservice, stream-first telemetry pipeline (ingestion→ broker→ stream processing→ time-series storage→ dashboards/alerts) suitable for real-time agrotelemetry. The pipeline design supports decoupled scaling and operational observability, enabling stable ingestion and processing under the tested sampling regime.

We designed and operationalized a streaming critical-event detector that combines static thresholds with dynamic z-score deviation and k-of-m confirmation logic. The detector naturally maps onto stateful stream operators, enabling timely alerting while suppressing spurious triggers through confirmation and refractory logic.

Using a unified experimental protocol and standardized KPIs, we quantified the trade-offs between ESP32 and Arduino Nano + LoRa. ESP32 offers lower end-to-end latency and is preferable when Wi-Fi coverage is available and time-sensitive alerting is required. Arduino Nano + LoRa is more suitable for remote and energy-constrained deployments, offering better autonomy at the cost of higher latency. Based on these findings, we recommend a hybrid strategy: route critical alerts via Wi-Fi (ESP32) and routine telemetry via LoRa to strike a balance between responsiveness and energy autonomy.

Limitations include a single-site evaluation and a limited sensor set; future work will expand to additional sensor types and multi-site deployments and will further standardize interoperability profiles.

We treat the reliability-oriented configuration as a reference profile; future work will quantify the incremental latency contributions of individual configuration toggles using controlled ablation experiments.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Natalia Axak: Conceptualization of the study and formulation of the research framework.

Maksym Kushnaryov: Development and implementation of the software components.

Yurii Shelikhov: Collection and validation of empirical data, analysis of sources, and preparation of the literature review.

All authors had approved the final version.

Funding

The research received support from the European Research Executive Agency (REA) through the European Union's Horizon 2020 program: INITIATE, HORIZON-WIDERA-2023-ACCESS-03 (grant 101136775).

References

[1] W. Tao, L. Zhao, G. Wang *et al.*, "Review of the internet-of-things communication technologies in smart agriculture and challenges," *Computers and Electronics in Agriculture,* vol. 189, Art. no. 106352, 2021.

[2] J. S. Horsburgh, K. Lippold, and D. L. Slaugh, "Adapting OGC's sensorthings API and data model to support data management and sharing for environmental sensors," *Environmental Modelling & Software,* vol. 183, Art. no. 106241, 2025.

[3] M. B. Motalab, A. Al-Mallahi, A. Martynenko *et al.*, "Development of an ISOBUS-compliant communication node for multiple machine vision systems on wide boom sprayers for nozzle control in spot application schemes," *Smart Agricultural Technology*, vol.10, Art. no. 100815, 2025.
[4] E. Bicamumakuba, E. Habineza, Samsuzzaman *et al.*, "IoT-enabled LoRaWAN gateway for monitoring and predicting spatial environmental parameters in smart greenhouses: A review," *Precision Agriculture Science and Technology,* vol. 7, no. 1, pp. 28–46, 2025.
[5] A. Morchid, R. Jebabra, H. M. Khalid *et al.*, "IoT-based smart irrigation management system to enhance agricultural water security using embedded systems, telemetry data, and cloud computing," *Results in Engineering,* vol. 23, Art. no. 102829, 2024.
[6] M. Escribà-Gelonch, S. Liang, P. v. Schalkwyk *et al.*, "Digital twins in agriculture: Orchestration and applications," *Journal of Agricultural and Food Chemistry,* vol. 72, no. 19, pp. 10737–10752, 2024.
[7] M. S. M. Rafi, M. Behjati, and A. S. Rafsanjani, "Reliable and cost-efficient IoT connectivity for smart agriculture: A comparative study of LPWAN, 5G, and hybrid connectivity models," in *Proc. International Conf. on Smart Computing and Informatics*, 2025, pp. 389–411.
[8] A. Pagano, D. Croce, I. Tinnirello *et al.*, "A Survey on LoRa for smart agriculture: Current trends and future perspectives," *IEEE Internet of Things Journal,* vol. 10, no. 4, pp. 3664–3679, 2023.
[9] Y. Zhao, Y. Yu, J. Kang *et al.*, "Intelligent irrigation system based on NB-IoT," in *Proc. 2022 IEEE International Conf. on Mechatronics and Automation (ICMA)*, 2022, pp. 1419–1424.
[10] M. Giacobbe, I. Falco, S. Zanafi *et al.*, "Key challenges in LoRaWAN-based edge–cloud infrastructures for security-sensitive smart-city applications," in *Proc. Joint National Conf. on Cybersecurity (ITASEC & SERICS 2025)*, 2025.
[11] N. S. Prasol, D. V. Furikhata, T. A. Vakaliuk *et al.*, "Integration of edge devices and IoT to create a climate monitoring system for plants," in *Proc. 5th Edge Computing Workshop*, 2025, pp. 4–19.
[12] G. Papadopoulos, S. Arduini, H. Uyar *et al.*, "Economic and environmental benefits of digital agricultural technologies in crop production: A review," *Smart Agricultural Technology,* vol. 8, Art. no. 100441, 2024.
[13] M. N. Mowla, N. Mowla, A. F. M. S. Shah *et al.*, "Internet of things and wireless sensor networks for smart agriculture applications: A survey," *IEEE Access,* vol. 11, pp. 145813–145852, 2023.
[14] O. Friha, M. A. Ferrag, L. Shu *et al.*, "Internet of things for the future of smart agriculture: A comprehensive survey of emerging technologies," *IEEE/CAA Journal of Automatica Sinica,* vol. 8, no. 4, pp. 718–752, 2021.
[15] A. Dauda, O. Flauzac, and F. Nolot, "A survey on IoT application architectures," *Sensors,* vol. 24, no. 16, Art. no. 5320, 2024.
[16] P. Gkonis, A. Giannopoulos, P. Trakadas *et al.*, "A survey on IoT-edge-cloud continuum systems: Status, challenges, use cases, and open issues," *Future Internet,* vol. 15, no. 12, Art. no. 383, 2023.
[17] A. E. Akhdar, C. Baidada, A. Kartit *et al.*, "Exploring the potential of microservices in internet of things: A systematic review of security and prospects," *Sensors,* vol. 24, no. 20, Art. no. 6771, 2024.
[18] H. Eichelberger, C. Sauer, A. S. Ahmadian *et al.*, "Industry 4.0/IIoT platforms for manufacturing systems—A systematic review contrasting the scientific and the industrial side," *Information and Software Technology,* vol. 179, Art. no. 107650, 2025.
[19] A. Dhulfiqar, M. A. Abdala, N. Pataki *et al.*, "Deploying a web service application on the edgex open edge server," *Procedia Computer Science,* vol. 235, pp. 852–862, 2024.
[20] S. H. Kim and T. Kim, "Local scheduling in KubeEdge-based edge computing environment," *Sensors,* vol. 23, no. 3, Art. no. 1522, 2023.
[21] A. Soussi, E. Zero, R. Sacile *et al.*, "Smart sensors and smart data for precision agriculture: A review," *Sensors,* vol. 24, no. 8, Art. no. 2647, 2024.
[22] V. Mazzia, L. Comba, A. Khaliq *et al.*, "UAV and machine-learning-based refinement of a satellite-driven vegetation index for precision agriculture," *Sensors*, vol. 20, no. 9, Art. no. 2530, 2020.
[23] N. Axak, M. Kushnaryov, and Y. Shelikhov, "The intelligent control of the city-farm microclimate based on the Q-learning algorithm," *Advanced Information Technology,* vol. 1, no. 3, pp. 13–22, 2024. (in Ukrainian).