

Resilient IoT-Based Heat Treatment Temperature Monitoring System with Offline Synchronization

Ariananda Panduaji Marsailla ^{1*}, Hafiyyan Putra Pratama ^{2*}

* Department of Telecommunication System, Universitas Pendidikan Indonesia
arianandapanduaji044@upi.edu ¹, hafiyyan@upi.edu ²

Article Info

Article history:

Received 2026-08-31

Revised 2026-10-09

Accepted 2026-10-17

Keyword:

*Catch-Up Synchronization,
Internet of Things (IoT),
IoT temperature monitoring,
Network Resilience,
Offline Buffering.*

ABSTRACT

Manual temperature recording during the heat treatment process is prone to human error and does not support continuous real-time monitoring. This research develops and evaluates an IoT-based temperature monitoring system featuring offline buffering, WiFi disruption detection-recovery, and catch-up synchronization mechanisms. The system uses a three-layer architecture (hardware, backend, frontend) based on an ESP32 microcontroller with dual DS18B20 sensors, a PHP/MySQL backend, and an HTML5/JavaScript dashboard. Five key technical designs were validated through quantitative testing on a controlled local network simulating the plant's operational conditions. The measured real-time data latency averaged 0.7 and 0.6 seconds for the two sensors, well under the 2-second target. Offline buffering and catch-up synchronization recovered 99.5% of buffered data (199 of 200 points) in the initial trials, the single failure attributed to a since-remediated firmware defect. Post-remediation trials showed no loss of retained data under near-capacity and overflow conditions and idempotent handling of REST API failures, although data buffered before a restart was lost. Zombie WiFi detection functioned reliably under both critical and self-recovering disruption conditions. REST API uptime reached 99.32% and 99.49% across day- and night-shift simulations, and DataMerger ran without producing duplicate entries or data gaps. An alternating timeout characteristic was additionally observed in the single-threaded ESP32 firmware architecture. The system met the proposed targets under the tested conditions, demonstrating that a layered network resilience approach can improve the reliability and integrity of industrial temperature monitoring. However, validation under actual heat treatment chamber conditions remains a direction for future work.



This is an open access article under the [CC-BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.

I. INTRODUCTION

Unstable network connectivity remains a major challenge to the reliability and scalability of Internet of Things (IoT) deployments, particularly in resource-constrained or inconsistent network environments [1]. This challenge grows even more complex in industrial settings, where cross-technology interference between WiFi and other wireless devices can significantly reduce communication reliability due to differences in transmission power across coexisting wireless technologies [2]. Even among consumer-grade IoT devices, a large-scale reliability study found that 88% of the 51 devices tested failed to meet at least one basic reliability

criterion [3], confirming that network resilience challenges are systemic across IoT device classes. In the industrial heat treatment process examined in this study, temperature recording is still done manually, making it prone to human error and unable to support continuous real-time monitoring.

IoT applications for industrial and environmental monitoring have been applied in a range of domains. In the food supply chain, Lamberty et al. noted that most IoT-based temperature monitoring solutions are designed for a single use case, making them difficult to generalize across domains [4]. Vo et al. identified unstable local networks as a major obstacle to IoT implementation in small- to medium-sized factories in developing countries, with a custom RF communication

solution achieving over 98% transmission success after applying a retry mechanism [5]. Low-cost ESP32-based IoT architectures have also been used for parking availability tracking systems [6] and solar panel monitoring systems [7]. At the same time, van Dinter et al. proposed a reference architecture for IoT- and digital-twin-based predictive maintenance systems in a broader industrial context [8]. In the smart agriculture domain, Li et al. developed an AIoT framework with hybrid communication (LoRa and NB-IoT), reporting average latency of 347–383 ms and mitigating harsh wireless network disturbances, including security evaluations against Denial-of-Service and Man-in-the-Middle attacks [9]. Another precision agriculture IoT system reported a data transmission reliability of 97.2% with a latency of 1.87 seconds in multi-sensor agroclimatic parameter monitoring [10]. From a communication protocol perspective, MQTT is widely used in IoT implementations due to its lower power consumption compared to HTTP [11]. However, some researchers still prefer HTTP/REST for implementation simplicity, as seen in a smart building power control system that reported an average cloud-to-device latency of 686.44 ms and resilience to network failures [12].

Temporary storage approaches for handling disconnection (store-and-forward) have been applied across various domains. Kartikasari et al. applied this approach in wearable healthcare, maintaining biosignal data continuity even when the connection was lost for up to 30 minutes [13], while Marin-de-Yzaguirre et al. implemented a Delay-Tolerant Network protocol on an IoT satellite network for intermittent connectivity conditions [14]. In a high-frequency lake-monitoring system, measurement data is retrieved from the device's local memory whenever wireless transmission fails, thereby preventing data loss [15]. A similar approach has also been applied in an IoT edge computing caching framework, which tracks undelivered data batches for retransmission once connectivity is re-established [16], as well as in an ESP32-based IoT platform for precision agricultural environmental monitoring, where data stored locally during connection disruptions is uploaded automatically after connectivity is restored [17]. In terms of network disruption detection and recovery, an ESP32-based smart greenhouse climate control system demonstrated rapid self-healing in response to network disturbances [18].

Low-cost IoT-based temperature logging systems have also been developed for wildland fire temperature monitoring using the Arduino platform [19]. At the same time, the accuracy of the DS18B20 sensor has been validated through comparison in soil temperature monitoring applications [20]. While the studies discussed above each address one piece of the network-resilience problem, store-and-forward buffering [13]–[17], self-healing disruption recovery [18], or reliable retransmission under unstable links [5], none combine buffer management, disruption-cause differentiation, and post-recovery data integrity into a single system evaluated as one pipeline. In particular, existing store-and-forward implementations typically report whether data survived a

disconnection, but not what happens at the buffer's capacity boundary, under device restart, or when a retried entry risks being recorded twice, gaps this research addresses directly through the circular buffer's capacity/restart behavior and duplicate-handling validation (Section III-C, III-D). The specific contribution of this research therefore lies not in any single mechanism in isolation, each has precedents elsewhere, but in combining circular buffering, dual-layer disruption detection, catch-up synchronization, and duplicate-safe merging into one instrumented system and characterizing its behavior at measurable failure boundaries, an evaluation the reviewed literature, taken individually, does not provide.

Given these issues, the research problem addressed in this study is the design of a temperature monitoring system that keeps data recording reliable and intact throughout the heat treatment process even when network connectivity is disrupted. This research aims to design, develop, and evaluate such a system for an industrial heat treatment process, built around offline buffering, WiFi disruption detection and recovery, and catch-up synchronization, with performance measured across five quantitative parameters covering real-time data latency, offline buffering accuracy, catch-up synchronization integrity, REST API uptime, and data synchronization capability.

The resulting system carries practical value in its potential to replace the manual temperature recording method currently used in this process, reducing the risk of recording errors and data loss caused by network disruptions, while also contributing to the industrial IoT literature through the alternating timeout characteristic identified in single-threaded microcontroller firmware and the layered network resilience mechanism validated as a reference point for similar industrial monitoring systems.

II. METHOD

A. System Development Method

This research was conducted using an iterative prototyping method adapted to the characteristics of embedded-IoT system development. This approach was selected because it enables hardware and software functionality to be validated in stages, so that any error at one stage can be detected and corrected before proceeding to the next. The development stages carried out in this research include:

- 1) **Requirements Analysis and Architecture Design:** At this stage, the functional and non-functional requirements for the system were determined based on the manual temperature recording process used in the heat treatment process at PT Findora Internusa, and the hardware, firmware, and database architecture was designed.
- 2) **Hardware Implementation and Sensor Selection Rationale:** At this stage, an ESP32 circuit was assembled and tested with two DS18B20 sensors (1-Wire protocol, GPIO 4), a TFT ST7735 module (SPI), and a tricolor LED indicator (GPIO 18/19/21), built on a breadboard using jumper wires

for early-stage testing. The ESP32 was chosen as the main microcontroller for its built-in WiFi module, which doubles as a REST API server, its low cost relative to comparable WiFi-enabled microcontrollers given this research's limited budget, and its sufficient GPIO pins and SRAM to run 1-Wire communication, SPI, and circular buffer storage at the same time. Two DS18B20 sensors are used per chamber at PT Findora Intemusa, most plausibly for redundancy and to catch temperature non-uniformity within the chamber, consistent with the inter-sensor consistency check reported later (Section III-H), though the exact rationale from the internship was not formally documented. The prototype evaluated here reflects a single such circuit (one ESP32, two sensors), tested via hot-water simulation due to the field-access limitation noted earlier. DS18B20 was chosen for its 1-Wire interface, which lets multiple sensors share one GPIO pin, and its $\pm 0.5^{\circ}\text{C}$ accuracy across -55°C to 125°C , comfortably covering the facility's $31\text{--}74^{\circ}\text{C}$ range. Absolute sensor accuracy was cross-checked against a commercial reference thermometer only within the upper part of the operating range (Section III-I), and the effect of industrial-environment factors on reading accuracy was not assessed, both noted as constraints of this study rather than resolved here.

3) **Firmware Implementation (C++/Arduino):** Firmware implementation followed an incremental development cycle, beginning with the most basic function, namely temperature reading by the DS18B20 sensor, followed by the development of temperature display and indicators on the TFT display. Each basic function that ran successfully was immediately tested. The results of this testing formed the basis for determining the next function to add. Through this repeated cycle of testing, problem identification, and code updates, several features were gradually added, including NTP time synchronization, a REST API server, circular buffer and offline state machine mechanisms, Zombie WiFi detection with auto-reconnect, and finally catch-up synchronization. Each of these features emerged as a direct response to needs or issues identified during the previous testing stage.

4) **Backend Implementation:** This implementation acts as a bridge between the ESP32 microcontroller and the MySQL database, handling both real-time data and offline buffer results. Values including the measured temperature, sensor identity, transmission type (real-time or buffered), and timestamp, used to preserve the chronological integrity of the data after synchronization are stored in MySQL due to its suitability with fixed tabular data structures, its support for structured queries (time-range filtering, chronological ordering, and duplicate prevention) that directly support the requirements of the catch-up synchronization and DataMerger mechanisms, and its ACID compliance, which helps maintain data integrity when receiving large-volume inserts at once during the post-offline synchronization process.

5) **Web Dashboard Implementation:** The monitoring interface was developed using HTML5/JavaScript with

interactive chart visualization powered by Chart.js and a zoom plugin. Data consistency is maintained through two core components. The first is SyncManager, which handles persistent synchronization using a Fibonacci retry interval strategy and remains session-aware of sensor changes or data deletions. The second is DataMerger, an idempotent slot-alignment-based algorithm that detects and fills data gaps by searching for the nearest value based on timestamp while preventing duplication, making it safe to call repeatedly. The dashboard also includes a data export feature to CSV and Excel formats to support documentation and further analysis.

6) **Integrated Testing and Evaluation:** Once the prototype reached its final stage, end-to-end testing was conducted against the previously established performance metrics, and the results were analyzed to identify necessary adjustments.

Overall, the six development stages of the prototype were carried out iteratively, particularly during the firmware implementation stage, as illustrated in Figure 1.

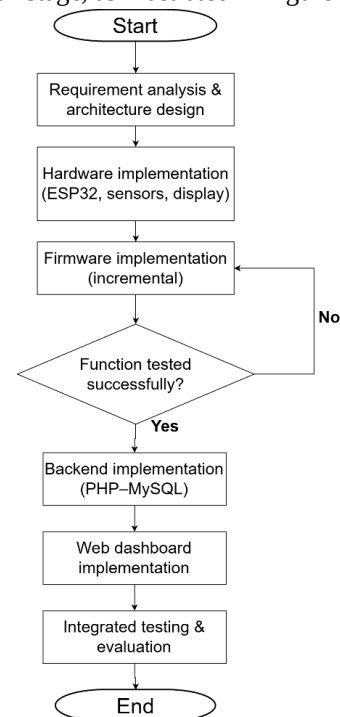


Figure 1 System Development Stages Flowchart

Through this iterative process, each firmware feature, from NTP synchronization to catch-up synchronization, could be progressively evaluated and refined based on the testing results from the previous cycle before the system proceeded to the next stage, namely backend and dashboard implementation.

B. System Architecture

The developed system is built on a three-layer architecture consisting of hardware, backend, and frontend, which are interconnected via the HTTP/REST protocol to ensure

consistent data exchange from the sensor to the dashboard display. On the backend, PHP was chosen as the programming language for its ease of integration with MySQL and suitability for implementing a simple REST API for this research prototype, as summarized in Table I.

TABLE I
SYSTEM ARCHITECTURE

Layer	Technology	Role in the System	Specification
Hardware	ESP32 DevKit	Primary microcontroller featuring WiFi, REST API server, and TFT display	NodeMCU ESP-32S (38-pin ESP32 Dev Module), Xtensa LX6 dual-core @ 240 MHz, 4 MB flash, 520 KB SRAM, programmed in Arduino IDE using Espressif's esp32 board package v2.0.17
Sensor	DS18B20 (×2)	12-bit temperature sensor via 1-Wire protocol	Both sensors share a single 1-Wire bus on GPIO4, addressed via the OneWire (v2.3.8) and DallasTemperature (v4.0.6) libraries, with resolution fixed at 12-bit with blocking conversion enabled, read once per loop iteration.
Display	ST7735 TFT 1.8"	Visualization for real-time temperature indicators & on-device server (NAS) and internet reachability status	SPI interface via Adafruit GFX Library (v1.12.6) and Adafruit ST7735 and ST7789 Library (v1.11.0).
Backend	PHP + MySQL (XAMPP)	Handles temperature log data reception, storage, and delivery	PHP 8.2.12 (mysqli extension), Apache 2.4.58, MariaDB 10.4.32, phpMyAdmin 5.2.1
Frontend	HTML5 / JavaScript	Monitoring dashboard, historical charts, CSV/XLSX export	Built as a single static page with vanilla JavaScript (no framework or build step), so browser compatibility depends only on native ES6 support and the CDN-hosted libraries listed in the Library row below.
Library	Chart.js + chartjs-plugin-zoom	Interactive chart visualization with pan & zoom	Chart.js v4.4.3 for rendering the temperature charts, paired with chartjs-plugin-zoom v2.0.1 for pan-and-zoom interaction and its Hammer.js v2.0.8 dependency for touch/gesture

			handling, plus SheetJS (xlsx) v0.18.5 for the CSV/XLSX export feature.
Protocol	HTTP REST (JSON)	ESP32 ↔ Server ↔ Browser communication	-

Sensor readings are stored in a single MySQL table, `log_temp`, made up of an auto-incrementing `id`, `batch_id`, `sensor`, `temp`, a type field separating real-time from buffered entries, a device-side `recorded_at/recorded_date` pair reflecting when the ESP32 took the reading, and a server-side `received_at` timestamp assigned automatically at the moment of insertion. A composite unique key on (`sensor`, `recorded_date`, `recorded_at`) blocks a retried request from being written twice, which is what allows the REST endpoint to stay idempotent under retry, discussed further in the API failure-handling subsection. One implementation detail worth noting here is that the DS18B20's 12-bit resolution, combined with blocking conversion left enabled in firmware, adds up to 750 ms of conversion time per reading. Because a new reading is requested on every loop iteration, this delay explains why the circular buffer's actual sampling interval slightly exceeded its coded 10-second target rather than matching it exactly.

While Table I outlines the technology stack assigned to each layer, it does not show how a single temperature reading actually moves across those layers. This flow is illustrated in Figure 2. A reading is first acquired from the two DS18B20 sensors by the ESP32 (path 1). Depending on the connection state at that moment, the reading leaves the device through one of two separate HTTP POST channels. Real-time readings are pushed to the backend every 5 minutes while the connection is available (path 2), while readings accumulated in the circular buffer during an outage are sent one by one once the catch-up process begins (path 3). Both channels are received by the same endpoint on the PHP backend, which inserts each record into the MySQL temperature log table (path 4). On the dashboard side, two retrieval channels run independently of one another, with a lightweight polling routine checking the `/data` and `/data1` endpoints every 2 seconds to keep the live connection status and latest readings visible on screen (path 5), while the `SyncManager` separately pulls the stored historical records from the backend in JSON format (path 6). Keeping these two retrieval paths apart is what later allows the `DataMerger` to reconcile real-time, catch-up, and historical entries into one continuous timeline without duplication, as explained in the next subsection.

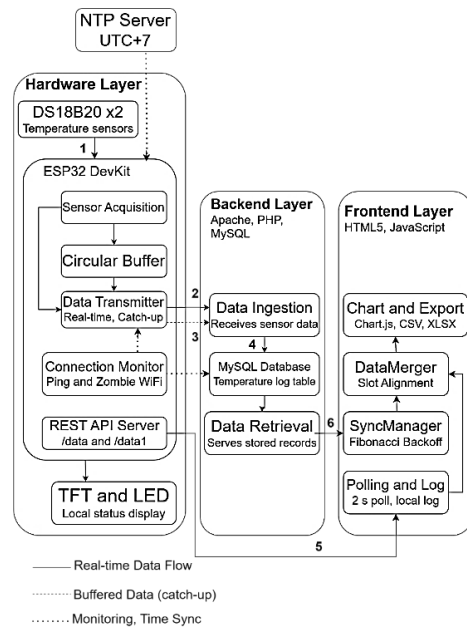


Figure 2 System Architecture and Data Flow Diagram

Data flow alone does not explain why the system behaves differently depending on network condition, so a second diagram is used to describe this behavior more directly. As shown in Figure 3, the device operates in three mutually exclusive states, Online, Offline, and Catch-Up. The system moves from Online to Offline once two consecutive connectivity checks fail, roughly 5 seconds apart (1). While remaining in the Offline state, repeated failures are tracked separately by the Zombie WiFi mechanism, which forces the ESP32 to reconnect after three consecutive failures are recorded (2). Once the server becomes reachable again, the device enters the Catch-Up state (3), where buffered entries are transmitted sequentially, and any transmission that fails within this state is retried up to three times before it is skipped and the next entry is attempted (4). The Catch-Up state only ends, returning the device to Online, after the entire buffer has been emptied (5). Read together, Figure 2 and Figure 3 describe the same system from two different angles, the first shows where data goes, and the second shows under what condition each path is actually taken.

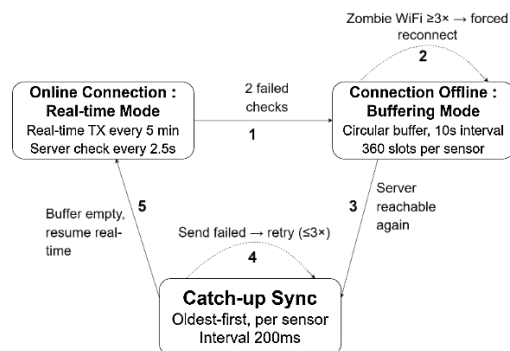


Figure 3 Device Connectivity State Diagram: Online, Offline, and Catch-Up

These two diagrams outline the overall behavior of the system, while the underlying mechanisms behind each transition and pathway, circular buffer, Zombie WiFi detection, catch-up synchronization, DataMerger, and SyncManager, are explained in more detail next.

C. Key Technical Design

To ensure reliable temperature data recording in the face of connectivity disruptions, this research focuses on five interrelated key technical designs, spanning from circular buffer and Zombie WiFi detection on the device side, through to catch-up synchronization, DataMerger, and SyncManager on the backend/dashboard side, as summarized in Table II.

TABLE II
KEY TECHNICAL DESIGN OF THE SYSTEM

Technical Design	Description
Circular Buffer ESP32 (360 slot)	Stores temperature data together with NTP timestamps in the ESP32's SRAM during offline mode, sized at 360 slots per sensor sampled on a 10-second tick, giving roughly one hour of local storage before the oldest entry is overwritten. The FIFO implementation ensures the latest data is always retained even when the buffer is full, without requiring external storage media.
Dual-Layer Zombie WiFi Detection	A two-layer reachability check run every 2.5 seconds. The first layer checks the local server (NAS) and declares the device offline after 2 consecutive failures. The second layer checks a public DNS resolver to distinguish a server-only issue from a broader network loss, and 3 consecutive second-layer failures trigger a forced WiFi reconnect.
Catch-Up Synchronization	Once the connection is restored, buffered data from each sensor is sent sequentially and independently, with a 200 ms interval between transmissions, along with data-type and original on-device timestamp parameters to maintain chronological integrity.
DataMerger (Slot-Alignment Idempotent)	An algorithm on the dashboard side that aligns data from multiple sources (real-time + buffer) into a time-based grid, removing duplicates and filling data gaps, and is safe to call repeatedly without side effects.
SyncManager (Fibonacci Backoff)	A persistent synchronization manager using a Fibonacci retry interval strategy, session-aware such that sensor changes or data deletion safely cancel any pending synchronization process.

To further illustrate the explanation in Table II above, the ESP32 firmware's processing logic, which covers connection checking, buffer storage, Zombie WiFi detection, and catch-up synchronization, is represented in the flowchart in Figure 4 below.

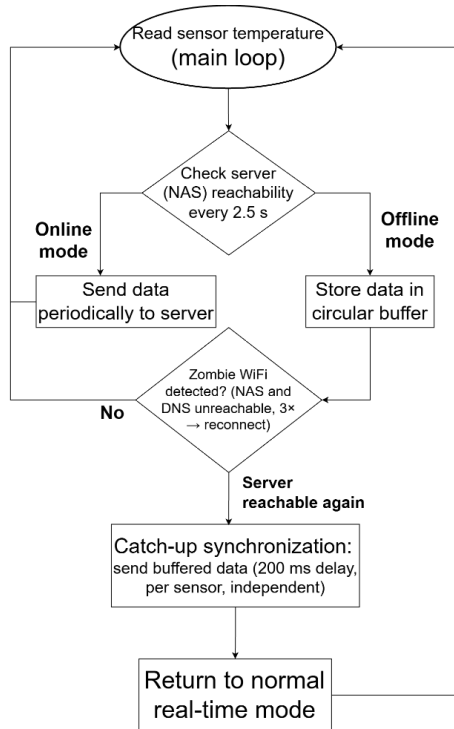


Figure 4 ESP32 Firmware Processing Logic Flowchart

The flowchart illustrates how the system automatically switches between online and offline modes without data loss, with the catch-up synchronization mechanism ensuring that all stored data is eventually transmitted when the connection is restored. Zombie WiFi detection appears in this flowchart as one part of the firmware's overall processing loop, and its internal decision logic is detailed in Figure 5.

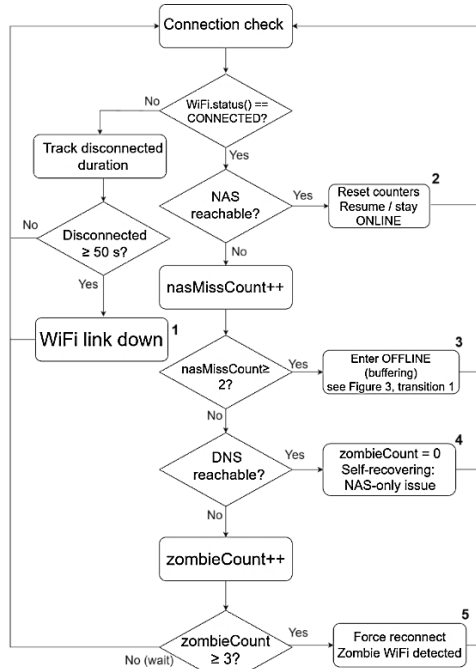


Figure 5 Dual-Layer Zombie WiFi Detection Algorithm

While Figure 4 shows Zombie WiFi detection as one part of the firmware's overall processing loop, the internal decision logic of this mechanism, run on every 2.5-second cycle, is detailed separately in Figure 5. The check begins by confirming whether the ESP32's WiFi radio itself is still connected; if not, the disconnection duration is tracked and a full reconnect is only forced once it has lasted 50 seconds continuously, avoiding unnecessary reconnect attempts during a brief radio-level drop. If the WiFi link is intact, the check moves to a first-layer target on the local network, a reachability check timed out after 1 second; a failure here increments a miss counter, and two consecutive misses are what actually triggers the transition into the Offline/buffering state shown in Figure 3. This alone, however, does not yet count as a critical failure, the algorithm proceeds to a second-layer check against a public DNS resolver (2-second timeout) to determine whether the disruption is confined to the local server or reflects a broader loss of internet connectivity. If the second-layer target remains reachable, the failure is classified as self-recovering, the local server issue is assumed transient, buffering continues, and no forced reconnect is triggered. Only if the second-layer target also fails, and does so on three consecutive cycles, is the disruption classified as critical, at which point the firmware forces a WiFi disconnect-and-reconnect regardless of what the radio itself reports. Table III summarizes the parameters governing this logic.

TABLE III
ZOMBIE WiFi DETECTION PARAMETERS

Parameter	Value
Connectivity check interval	2.5 seconds
First-layer target (local server), timeout	1 second
Offline-state threshold (first layer)	2 consecutive failures
Second-layer target (public DNS resolver), timeout	2 seconds
Zombie/forced-reconnect threshold (second layer)	3 consecutive failures
Pure WiFi-disconnect timeout	50 seconds

This dual-layer structure lets the firmware tell apart two disruption types that would otherwise look identical from the device's perspective. A self-recovering disruption, where the local server is unreachable but the wider internet still is, gets left alone, the device stays on WiFi, keeps buffering locally, and resumes once the server responds, since forcing a reconnect would not fix a problem that has nothing to do with the WiFi link itself. A critical disruption covers the two conditions described above and both trigger the same forced-reconnect response, with the second one, WiFi reporting itself active while both targets stay unreachable, being the actual "zombie" state, a connection that looks alive at the radio level but is functionally dead.

D. Timestamp Handling and Time Synchronization

Each stored reading actually carries up to three separate timestamps, so it is worth being explicit about where they

come from. `recorded_at/recorded_date` is device-side, taken from the ESP32's clock synchronized to `pool.ntp.org` at UTC+7, and is only attached to buffered/catch-up entries, real-time uploads omit this parameter entirely, leaving those fields NULL. `received_at`, by contrast, is assigned independently by the MySQL server at the moment of insertion. Because the two clocks trace back to unrelated time sources, some drift between them is expected, and was in fact observed during REST API failure testing, where the ESP32's own reconnect log and the corresponding database entry differed by about two seconds.

Only the time-of-day of a buffered reading comes from the device, while its calendar date is inferred on the server by comparing that time against the server's current time (more than +12 hours difference is treated as yesterday, less than -12 hours as tomorrow, otherwise today), a rule that stays safe only because the buffer holds at most about one hour of data, well inside that ± 12 -hour margin. Ordering is affected as well. Catch-up entries are sent oldest-first per sensor with their original device timestamp attached, so arrival order at the server does not necessarily reflect capture order once two sensors' buffers interleave. This is why chronological integrity is reconstructed from `recorded_at` rather than assumed from insertion sequence, consistent with how `DataMerger` aligns entries by recorded time slot rather than by arrival, as described earlier. Having covered how the system is built and how its timestamps are kept consistent, the discussion now moves to the environment in which the system was evaluated.

E. Testing Environment and Scheme

This system was designed to address temperature recording issues in the heat treatment process at PT Findora Internusa, which runs two shifts a day, with the night shift estimated to run from midnight until just before sunrise. Manual records of 1,289 HT batches (a separate drying process using the same chamber at a lower temperature range was excluded) show the process operating within 31–74°C, rising on average from 34.8°C to 70.9°C within a single batch, each lasting 1.0 to 1.75 hours (average 1.63 hours), so a shift typically runs several such batches back to back rather than one continuous cycle. No formal tolerance limit was specified, so the within-batch temperature rise (about 36°C, SD 2.7°C) is treated here as the practical operating band the system must track. With on-site field access no longer available during testing, evaluation was carried out instead on a local area network built to approximate both shifts' conditions, temperature rise was simulated by immersing both DS18B20 sensors in hot water, reaching $\pm 27^\circ\text{C}$ to $\pm 61^\circ\text{C}$, while the ESP32 and server (XAMPP) shared the same network segment. An initial functional demonstration using the same hot-water method was also conducted on-site at PT Findora Internusa before this controlled quantitative testing.

The intended deployment places the ESP32, TFT display, and status LEDs in an enclosure outside the chamber, with only the DS18B20 cables routed inside, a planned

arrangement not yet physically installed or validated. The chamber itself is an indoor black steel enclosure surrounded by rattan weaving material. Based on the author's on-site observations during the internship, it stayed consistently warm and looked free of heavy dust, a qualitative impression rather than an instrument-measured metric. Humidity, dust concentration, and electromagnetic interference were not quantitatively assessed and are noted as a limitation of this study.

Network disruption was induced in two ways depending on the failure mode being examined. Device-side disconnection, used for the offline buffering and catch-up trials, came from switching off the WiFi access point, while server-side disruption, used for the REST API failure trials, came from stopping Apache and/or MySQL through the XAMPP control panel, isolating whether a failure originated from the connection or the service behind it. Duration was varied deliberately to stress-test the buffer at and beyond its design boundary, from short 2 to 10-minute baseline outages, to a 20-minute outage combined with device restart, to 55 to 87-minute outages that exercise the buffer's one-hour limit, alongside separate 2 to 5-minute server-down trials, together producing several hundred readings across sixteen scenarios. These durations were not derived from a measured distribution of real outages at PT Findora Internusa, since no such log was available, and the local LAN setup does not reproduce factory-specific conditions like WiFi interference from industrial equipment or the physical distance to the access point. The trials are therefore best read as a controlled characterization across a representative range of disruptions, not a direct reproduction of the facility's actual network conditions, consistent with the broader scope limitation noted earlier.

F. Testing Parameters

System evaluation was carried out quantitatively across five parameters representing the system's core functional aspects, with success criteria based on the targets established during the research proposal stage, as recorded in Table IV.

The under-2-second latency target was not derived from a strict physical requirement of the heat treatment process itself, chamber temperature during rattan curing changes gradually over minutes rather than seconds, so sub-second monitoring offers no practical benefit to the process. It was instead set as a dashboard-responsiveness benchmark informed by comparable real-time IoT monitoring systems discussed earlier in this work, which report latencies ranging from roughly 350 milliseconds to under two seconds depending on protocol and network condition [9], [10], [12]. A target in that same range keeps a reading feeling immediate to an operator glancing at the dashboard, without demanding a level of network performance the intended low-cost, WiFi-based deployment could not realistically sustain. The zero-data-loss and above-99% uptime targets follow a different logic entirely, one grounded directly in the operational priority stated by PT Findora Internusa. Under this operational

principle, for a heat treatment log to hold value as a process or quality record, a reading that arrives late is still useful, but a reading that never arrives at all is not. This is also why offline buffering and catch-up synchronization, rather than latency minimization, were treated as the central design problem of this research, and why the system was built to tolerate delay far more readily than it tolerates loss.

TABLE IV
TESTING PARAMETERS AND SUCCESS CRITERIA

No	Parameter	Measurement Method	Target
1	Real-time data latency	Comparison between the NTP timestamp at transmission and the timestamp when displayed on the dashboard	< 2 seconds
2	Offline buffering accuracy	Comparison between data in the ESP32 buffer slots and data in the database after reconnection, assessed through five complementary metrics: data loss rate, synchronization delay, retry count, recovery time, and duplicate count.	Zero data loss
3	Catch-up synchronization integrity	Comparison of sequence and temperature values between local buffer and database per timestamp	Identical sequence & values
4	ESP32 REST API uptime	Automated HTTP polling every 2 seconds for 6 hours on the /data and /data1 endpoints	Uptime > 99%
5	DataMerger synchronization capability	Black-box test, 30-minute offline simulation, reconnection, then dashboard table validation	No duplicates & no gaps

G. Data Inclusion and Exclusion Criteria

A data point was recorded as recovered when its sequence number, timestamp, and value in the database lined up with the matching entry in the ESP32's buffer. In Parameter 4, a poll was marked failed when no response arrived within 3 seconds. Meanwhile, a Zombie WiFi condition was only confirmed after three consecutive second-layer (public DNS) reachability failures (3/3), matching the threshold in Table III. Any buffer gap under the 300-second dashboard interval, such as the 16-second gap in Trial 2 (Section III-G), had no effect on the dashboard display, so we simply logged it without flagging it. The anomalies found during testing, the single S2 upload failure in Section III-B and the transient HTTP connection error in Section III-G, were kept in the final results because neither satisfied the exclusion conditions. Repeated and discarded trials are identified explicitly where they occur, such as the repeat 2-minute trial for dashboard timing (Section III-B), the near-capacity buffer trial repeated after remediation (Section III-C), and the passive-cooling attempt dropped from the reference thermometer comparison (Section III-I).

III. RESULTS AND DISCUSSION

This section presents the testing results for the five parameters established in Section II-F, complemented by additional tests covering circular buffer behavior under extreme conditions, REST API failure handling and idempotency, Zombie WiFi detection, inter-sensor consistency, and cross-validation against a reference thermometer, followed by a discussion of security and operational considerations. Each parameter is presented with the obtained measurement data, technical analysis, and a conclusion on target achievement, followed by a general discussion linking all findings to the overall research objectives. All testing in this section was conducted in the testing environment and scheme described in Section II-E.

A. Real-Time Data Latency

The real-time data latency test measures the time gap between when the sensor reads and transmits temperature data and when that data appears on the web dashboard. Measurement was performed by comparing the NTP timestamp recorded on the ESP32 Serial Monitor (T1, transmission time) with the time the data appeared on the dashboard interface (T2, display time). The NTP timestamp was selected as the reference point because it stays synchronized with an internet time server, making it more accurate than the automatic timestamp in the Arduino IDE, which can be delayed by USB port buffering. For testing purposes, the ESP32's real-time data transmission interval was temporarily reduced from 300,000 ms (5 minutes) to 10,000 ms (10 seconds), then restored to its original value once testing was complete. Testing was performed on each sensor over 10 observations, as shown in Table V.

TABLE V
REAL-TIME DATA LATENCY MEASUREMENT RESULTS

No	T1 Send S1	T2 Display S1	Latency S1	T1 Send S2	T2 Display S2	Latency S2
1	17:45:15	17:45:15	0 s	18:04:31	18:04:32	0 s
2	17:45:56	17:45:56	0 s	18:05:02	18:05:03	1 s
3	17:46:38	17:46:39	1 s	18:05:23	18:05:24	1 s
4	17:55:04	17:55:05	1 s	18:06:05	18:06:06	1 s
5	17:55:45	17:55:46	1 s	18:06:56	18:06:56	0 s
6	17:56:37	17:56:39	2 s	18:07:29	18:07:29	0 s
7	17:57:40	17:57:41	1 s	18:07:59	18:07:59	0 s
8	17:58:47	17:58:47	0 s	18:08:40	18:08:41	1 s
9	17:59:24	17:59:25	1 s	18:10:03	18:10:04	1 s
10	17:59:55	17:59:55	0 s	18:11:05	18:11:06	1 s
Minimum		-	0 s	-	-	0 s
Median			1 s			1 s
Average			0.7 s			0.6 s
Maximum			2 s			1 s
P95			2 s			1 s
P99			2 s			1 s

The measured average latency was 0.7 seconds for Sensor 1 and 0.6 seconds for Sensor 2, with a maximum latency of 2 seconds recorded on Sensor 1. The distribution of results indicates that 40% of observations fell below 1 second, 60%

fell within the 1–2 second range, and no observation exceeded 2 seconds. This measurement variation was relatively small, with standard deviations of 0.67 seconds for Sensor 1 and 0.52 seconds for Sensor 2, suggesting that the system's latency performance remained consistent across observations. These low results were partly attributable to the dashboard-side polling mechanism, which uses a `setInterval` of 1,000 ms, theoretically limiting dashboard-side latency to at most 1 second in the worst case.

Summary of Parameter 1. The Real-Time Data Latency parameter was successfully met, meeting the proposal target of less than 2 seconds, with average latencies of 0.7 seconds (S1) and 0.6 seconds (S2), and no observations exceeding the threshold. Reporting the average alone leaves the spread of these measurements unexamined, so the same ten observations per sensor were also reduced to their minimum, median, maximum, and 95th/99th percentiles. Both sensors' latency stayed tightly clustered around their median of 1 second, with S1 ranging from 0 to 2 seconds and S2 from 0 to 1 second; because timestamps in this trial were only resolved to whole seconds and each sensor was sampled only ten times, the 95th and 99th percentiles coincide with the observed maximum for both sensors rather than sitting meaningfully below it, a known property of percentile estimation at small sample sizes rather than an indication that the true tail behavior has been fully characterized. All observed values, including this upper bound, remained within the proposal's 2-second threshold, though the limited sample size means this margin should be interpreted as an early indication of stability rather than a statistically robust guarantee, larger-scale latency testing is noted as a direction for future work.

B. Offline Buffering Accuracy and Catch-Up Synchronization Integrity

Both parameters were tested together because they are interdependent, offline buffering only holds value if the stored data can be recovered completely and in order through catch-up. Testing cut the ESP32's WiFi while both sensors sat in hot water simulating the HT process's dynamic temperature change, then reconnected and tracked recovery across three durations, 2, 5, and 10 minutes.

Dashboard update time for the 2-minute scenario wasn't recorded in the initial test, a repeat trial of similar duration filled this gap at roughly 10 seconds. In Trial 2, one S2 data point (14:08:56, 56.94°C) failed to upload, an S2 success rate of 96.67% (29/30). Of the four possible explanations, delay, duplication, buffer-capacity limitation, or permanent loss, only permanent loss fits, the entry never resurfaced, no duplicate appeared elsewhere, and the buffer held just 30 of its 360-slot capacity, ruling out overflow. This trial ran on the same pre-correction firmware later found, during extreme buffer testing (Section III-C), to silently accept a failed insert as a false success without retrying, the same signature diagnosed and fixed there. No equivalent failure occurred in the other trials, pointing to an isolated, since-corrected defect rather than a recurring flaw.

TABLE VI
OFFLINE BUFFERING & CATCH-UP TRIAL COMPARISON SUMMARY

Parameter	Trial 1 (2 minutes)	Trial 2 (5 minutes)	Trial 3 (10 minutes)
Disconnection time	13:33:27	14:04:00	20:57:02
Reconnection time	13:35:24	14:09:02	21:07:37
Actual offline duration	1 min 57 s	5 min 2 s	10 min 35 s
Buffered data S1	12	30	58
Buffered data S2	12	30	58
Uploaded data S1	12 (100%)	30 (100%)	58 (100%)
Uploaded data S2	12 (100%)	29 (96.67%)	58 (100%)
Zombie WiFi detected	No	No	Yes
Dashboard update post-catch-up	±10 seconds (from additional replication)	±2 minutes	±40 seconds
Overall result	SUCCESS	SUCCESS (with 1 transient anomaly on S2)	SUCCESS

Catch-up integrity was verified by comparing buffer sequence and timestamps against the database post-reconnection, no out-of-order entries or duplicate timestamps appeared in any trial, consistent with Catch-Up Synchronization's use of the device's own timestamp rather than server reception time. The sync-to-database step and the dashboard's chart update also turned out to be two distinct stages, the former completed almost instantly per the Serial Monitor, while the latter's duration didn't scale with data volume (±2 minutes for 30 points in Trial 2 vs. ±40 seconds for 58 points in Trial 3), suggesting network conditions during catch-up matter more than data volume itself. This dashboard timing is a supporting observation, not a formal success criterion (Table IV), so its variation doesn't affect the validity of the core result.

Trial 3 also triggered Zombie WiFi detection before reconnecting (Section III-E), yet all 58 points from both sensors were still fully recovered, showing the mechanism holds even under a more complex disruption.

Summary of Parameters 2 & 3. Across the three scenarios, 199 of 200 buffered points (99.5%) were fully recovered with consistent ordering, the single S2 failure was transient and did not recur. A separate 20-minute trial**, run after the circular buffer correction, also recovered all 119 buffered points for both sensors with zero loss.

C. Circular Buffer Behavior Under Extreme Conditions

The buffer was also pushed toward its design limits, near-capacity (±55 min), overflow (±87 min), and interruption via restart or power loss. Table VII summarizes the two capacity trials.

The first run recovered only 81.4% of buffered points, most of the untraceable loss came from the backend silently returning HTTP 200 on failed inserts combined with the firmware never retrying missed entries. Fixing both issues brought a repeat trial to 100% recovery with zero duplication.

The overflow trial filled all 360 slots per sensor and recovered every one, with the earliest ~23 minutes overwritten, confirming the buffer sacrifices the oldest unsent data rather than rejecting new readings once full. Because readings kept arriving every 10 seconds while no synchronization was possible, the overflow trial also covers the case where incoming data outpaces synchronization.

TABLE VII
CIRCULAR BUFFER CAPACITY TEST RESULTS

Trial	Offline Duration	Buffered per Sensor	Synced to Database	Data Loss	Recovery Time
Scenario 1: near-capacity (pre-remediation)	±55 min	307	250 (81.4%)	57 (18.6%)	±6 min 54 s
Scenario 1: near-capacity (post-remediation)	±54 min 51 s	309	309 (100%)	0%	±3 min 29 s
Scenario 2: overflow	±87 min	360 (100% of retained buffer)	360 (100% of retained buffer)	0%*	±4 min 33 s

*Scenario 2: the 0% loss figure applies to the 360 retained slots, roughly 23 minutes of the earliest data was overwritten per the FIFO policy, as discussed in the text.

Restart behavior, summarized in Table VIII, was tested separately since it interrupts the buffer differently than simply filling it.

TABLE VIII
CIRCULAR BUFFER BEHAVIOR UNDER DEVICE RESTART

	Soft Restart (NTP pre-remediation)	Hard Restart (NTP pre-remediation)	Soft Restart (NTP post-remediation)	Hard Restart (NTP post-remediation)
Offline start	09:45:10	10:22:08	22:00:11	22:35:11
Reconnect	Invalid ("Time Error")	Invalid ("Time Error")	22:21:53 (±21 min 42 s)	23:00:27 (±25 min 16 s)
Pre-restart buffered data recovered	0	0	0	0
Post-restart data synced	4/sensor	26/sensor	4/sensor	20/sensor
Post-restart data loss	0%	0%	0%	0%
Duplication	None	None	None	None
Valid timestamp on recovery	0/4	0/26	*1/4	0/20

*One of four post-restart entries recovered a valid timestamp (22:21:47), coinciding with a logged "NTP re-sync success" event, confirming the retry

mechanism worked when the interval between reconnection and resumed catch-up allowed at least one cycle to complete. The other three preceded this resync and stayed invalid.

Data buffered before a soft restart or hard power loss did not survive any of the four runs, consistent with the buffer sitting in volatile SRAM, while data recorded after each restart synced with zero loss and no duplication, confirming the earlier fix held under restart too. A separate NTP issue also surfaced, a restart before WiFi reconnected could permanently lose timestamping for that session, a retry loop resolved this for soft restarts but only partially for hard restarts, given the narrower recovery window.

D. REST API Failure Handling, Idempotency, and Synchronization Metrics

Beyond the recovered-percentage figures reported earlier, the same trials were re-examined against five complementary metrics, data loss rate, synchronization delay, retry count, recovery time, and duplicate count, using the post-remediation REST API failure trials as the reference dataset, since these were collected after the catch-up retry logic had already been corrected. Table IX summarizes the results across three such trials.

TABLE IX
REST API FAILURE TRIAL METRICS

Metric	Trial 1 (±4 min 45 s server-down)	Trial 2 (±2 min)	Trial 3 (±2 min)
Data loss rate	0% (28/28)	0% (12/12)	0% (12/12)
Synchronization delay (reconnect → last entry synced)	±20 s (56 points, both sensors)	±8 s (24 points)	±8 s (24 points)
Recovery time (server back up → detected)	not logged	3 s	2 s
Retry count	0	0	0
Duplicate count	0	0	0

Across three server-down trials totaling 104 buffered points, nothing was lost, no retry was required, and no duplicate row appeared. Synchronization delay ranged from 8 to 20 seconds depending on the number of queued points, and the server was detected reachable again within 2 to 3 seconds. The absence of retries here fits the short offline duration involved, retries only surfaced under the near-capacity buffer condition reported earlier (Table VII). Duplicate prevention also isn't purely behavioral, the sensor-time uniqueness constraint at the database level structurally rejects a repeated insert no matter how many times an entry is retried, making the zero-duplicate result a guarantee by design rather than a lucky outcome.

A separate concern is what happens when the same reading is sent more than once, whether from a retry after a false failure or any other repetition. The backend originally had no protection against this, three identical requests sent deliberately produced three identical rows, differing only in insertion timestamp. This was fixed with a composite unique constraint on sensor, date, and device-recorded time, paired with backend logic that returns HTTP 200 rather than an error on a detected duplicate, letting the ESP32 treat a rejected duplicate as a successful save and move on. Table X summarizes the validation of this mechanism.

TABLE X
IDEMPOTENCY VALIDATION

Identical payloads sent	Rows accepted	Rows rejected as duplicate	Response code
3× (16:52:10)	1	2	HTTP 200 (all)
2× (16:52:20)	1	1	HTTP 200 (all)
1× (16:52:30)	1	-	HTTP 200
1× (16:52:40)	1	-	HTTP 200
3× (16:52:50)	1	2	HTTP 200 (all)
Total: 10 requests	5 unique rows	5 duplicates rejected	0 errors

Note: auto-increment ID gaps are expected here, since a rejected duplicate still consumes an ID. Row count and the sensor-date-time combination, not ID continuity, indicate duplication.

This guarantee only applies to buffered/catch-up data, the only path ever retried in this system, real-time uploads are never retried and never hit this duplication path at all. At-least-once delivery on transmission combined with deduplication on storage produces an effectively exactly-once outcome, valid as long as no sensor produces two readings in the same second and buffered data is recovered within 12 hours, both true throughout testing.

A related failure mode is a server that accepts the connection but cannot actually store anything, tested by disabling only the database while the web server and WiFi stayed active. Under the original design the firmware had no way to detect this, real-time uploads kept failing silently with HTTP 500 that was never retried or buffered, losing every reading sent during the outage. A gradual database recovery made this worse, buffered catch-up data was retried against a still-unready database, each failed attempt taking over four seconds to time out, stalling the main loop (sensor reading, dashboard polling, everything) for over eight minutes while the firmware falsely reported success. Table XI compares this behavior before and after remediation.

TABLE XI
DATABASE-DOWN FAILURE HANDLING (PRE VS POST REMEDIATION)

Aspect	Pre-Remediation	Post-Remediation
Reachability check	Web server (Apache) only	Web server and database, via a dedicated health-check endpoint
Realtime data lost during DB outage	6/6 (100%)	0/0 (buffered instead of attempted)
Catch-up during gradual recovery	0/36 recovered, reported as "Succeed"	60/60 (100%) recovered
Catch-up duration under recovery	512 s (8 min 32 s), firmware unresponsive throughout	22 s, 0 failed requests
HTTP request timeout	None (unbounded)	1500 ms connect / 2000 ms response, explicit

The remediation didn't change the retry-then-skip policy itself, that was rarely reached anyway, it changed what counts as "reachable." The health-check endpoint now returns success only when both the web server and the database behind it are functioning, so the device withholds catch-up until data can actually be stored. An unplanned secondary WiFi disruption mid-catch-up during regression testing also completed successfully (24/24 per sensor, zero loss, zero duplication), incidental evidence that the retry and buffer-preservation logic holds under layered disruption too.

These trials were each run once or twice under controlled, single-fault conditions, evaluated independently rather than combined, and don't cover every HTTP error class (only 500-level failures and timeouts occurred in practice). Broader fault injection across more status codes remains future work.

E. Zombie WiFi Detection and Recovery

The dual-layer Zombie WiFi Detection mechanism (Table II) was tested using a two-layer reachability check against the local server (NAS) and a public DNS resolver, with a 3-consecutive-failure threshold on the second layer to trigger a forced reconnect. Throughout testing, the mechanism exhibited two distinct behavior patterns, both relevant for assessing the accuracy of its calibration, as summarized in Table XII.

TABLE XII
TWO OBSERVED ZOMBIE WiFi DETECTION BEHAVIOR MODES

Aspect	Trial 3: Offline Buffering (10 minutes)	Trial 2: DataMerger (± 31 minutes)
Trigger condition	Second-layer target (public DNS) unreachable, reaching the full 3/3 failure count	Second-layer target briefly unreachable but stayed below 3/3
System response	Automatic forced reconnect initiated	No forced reconnect (self-recovered condition)
Post-event outcome	Reconnection successful; 58/58 (100%) upload for both sensors	No disruption; buffer remained 175/175 (100%)
Interpretation	Mechanism activates fully when conditions reach the critical threshold	Mechanism is sufficiently sensitive to record borderline anomalies without over-triggering reconnects

Both observed behavior modes show that the detection mechanism adapts, being sensitive enough to detect problematic network conditions while avoiding overly aggressive reconnect behavior when conditions have not yet become genuinely critical. In Trial 3, the better catch-up outcome (100% upload and a faster dashboard update) compared to Trial 2 in Section III-G coincided with an automatic forced reconnect. However, given that the two trials differed in duration and data volume, the cause-and-effect link between the forced reconnect and the resulting connection quality is still only a preliminary observation and requires further validation through controlled testing under equivalent conditions.

Summary of Zombie WiFi Detection and Recovery. The Zombie WiFi detection and recovery mechanism was validated in two real-world scenarios of varying severity, responding correctly in both cases without requiring operator intervention.

F. ESP32 REST API Uptime

REST API uptime was tested through automated HTTP polling every 2 seconds on the /data and /data1 endpoints over a 6-hour period, corresponding to the approximate duration of a single operational shift at PT Findora Internusa, during which multiple HT batches (each averaging 1.63 hours) are processed sequentially. Uptime is defined here as the proportion of polling attempts receiving an HTTP 200 response within a 3-second per-request timeout, a timeout, connection error, or non-200 response counts as a failure. Testing ran twice at different times to represent the two operational shifts (day and night), a combined 12 hours and 14,464 polling attempts. The results are summarized in Table XIII.

TABLE XIII
REST API UPTIME COMPARISON: RUN 1 VS RUN 2

Parameter	Run 1 (Day)	Run 2 (Night)	Average
Execution time	11:58–17:58	23:05–05:05	-
Total polls	7.101	7.363	7.232
Uptime S1	98.73% (FAIL)	99.90% (PASS)	99.32% (PASS)
Uptime S2	99.04% (PASS)	99.95% (PASS)	99.49% (PASS)
Failures S1	90 times	7 times	48.5 times
Failures S2	68 times	4 times	36 times
Average latency S1	363.4 ms	330.5 ms	346.9 ms
Average latency S2	617.2 ms	598.6 ms	607.9 ms
Network condition	Day: active	Night: quiet	-
Conclusion	S1 FAIL / S2 PASS	S1 PASS / S2 PASS	PARAMETER 4 PASSED

In Run 1, S1 uptime of 98.73% missed the 99% target by 0.27%, just 90 of 7.101 requests. Of the 117 total failures, 76 (65%) followed an alternating pattern (one sensor failing while the other kept responding) and 41 (35%) failed simultaneously on both. The alternating pattern traces to the ESP32 firmware's single-threaded architecture, specifically its synchronous WebServer library, which processes one client per server.handleClient() call inside the same loop() that also runs blocking operations like sensor conversion (up to 750 ms) and buffer management. Whichever endpoint's request arrives while the loop is busy elsewhere queues behind it and risks timing out, while the other, arriving once the loop frees up, gets served normally, producing an alternating rather than simultaneous pattern. This blocking is confined to dashboard-facing polling and doesn't touch outbound data transmission, which runs on a separate, client-initiated path, consistent with no temperature data being lost despite these polling failures. The same blocking window could in principle delay a connectivity check too, though the 2-consecutive-failure offline threshold (Section II-C) offers some margin against a single transient delay. The simultaneous failures, by contrast, more likely reflect WiFi congestion during working hours, consistent with prior findings that cross-technology interference on unlicensed bands reduces reliability in device-dense environments [2]. Recovery was mostly immediate, 43 of 60 S1 failure clusters and 30 of 46 for S2 were single-poll misses recovering on the next 2-second cycle, while the worst clusters reached 34 seconds (S1) and 28 seconds (S2), still well short of anything that would exceed the buffer's recovery window.

Run 2, at night, ran under a much quieter network and showed higher uptime for both sensors (S1: 99.90%, S2: 99.95%), with failures dropping sharply (7 vs. 90 for S1). This points to WiFi conditions, not the single-threaded architecture, as the dominant factor, though the alternating pattern still appeared briefly early in Run 2, confirming the mechanism stays active even on a cleaner network. Both runs are reported without adjustment, since each represents real

production conditions for its shift and neither is more representative than the other.

Summary of Parameter 4. Averaged across both runs, S1 reached 99.32% and S2 99.49% uptime, both meeting the >99% target. Run 1's marginal miss for S1 (0.27 points, 90 of 7,101 requests) has a clearly traceable cause and no impact on data integrity, since the circular buffer keeps capturing readings regardless of API request success.

G. DataMerger Synchronization Capability

Parameter 5 was tested using a black-box scenario, disconnecting the network for about 30 minutes, injecting dynamic temperature changes toward the end of the offline period, then verifying the outcome in the database and on the dashboard. Testing was run twice, independently and on different days, as a replication (Table XIV).

Integrity during synchronization rests on two components sharing the work. DataMerger identifies each data point by its position on a fixed time grid (5-minute dashboard slots) instead of a unique ID, reconciling real-time and buffered entries that land in the same slot and discarding a repeat for a slot already filled. This works because Catch-Up Synchronization attaches the device's original timestamp to every buffered entry rather than its arrival time (Section II-D), so the two mechanisms depend on each other. Since the alignment depends only on the current state of the data, DataMerger can be run repeatedly, even after an interrupted update, without producing a different or duplicated result, the idempotent property noted in Table II. SyncManager keeps the dashboard's periodic backend requests alive under unstable conditions, retrying on a Fibonacci-interval backoff (1, 1, 2, 3, 5 seconds, and so on) so repeated failures progressively reduce request frequency rather than flooding a struggling connection. It is also session-aware, meaning a sensor change or data deletion cancels any sync attempt in progress instead of letting it write stale data afterward.

TABLE XIV
DATAMERGER TEST RESULTS: TRIAL 1 VS TRIAL 2

Parameter	Trial 1 (July 4, 2026)	Trial 2 (July 5, 2026)
Offline duration	30 min 32 s	31 min 13 s
Offline confirmed time	15:11:34	20:14:33
NAS reconnected time	15:42:06	20:45:50
Buffer S1	170/170 (100%)	175/175 (100%)
Buffer S2	170/170 (100%)	175/175 (100%)
S2 HTTP Failed	None	1 time, transient (self-recovered)
Buffer data gap	0 gaps	1 gap (16 seconds)
Zombie WiFi detected	No	Yes, below reconnect threshold
Duplicates in database	None	None
Gaps in dashboard	None	None
Dashboard slot interval	5 minutes, exact	5 minutes, exact
DataMerger result	SUCCESS	SUCCESS

Verification of the core criteria, no duplication, no interval gaps, upload completeness, and chronological ordering, is presented in Table XV.

TABLE XV
DATAMERGER CRITERIA VERIFICATION

Verification Criteria	S1	S2
No duplicates in database (unique recorded_at)	Confirmed	Confirmed
No interval gaps on dashboard (exact 5 minutes)	Confirmed	Confirmed
All buffer data uploaded	100%	100%
Buffer timestamps in correct chronological order	Confirmed	Confirmed

In Trial 1, nine dashboard slots at 5-minute intervals (15:01 to 15:41) landed exactly 300 seconds apart, including at the transition between real-time data (before offline) and buffered data (offline period), which is by design the point most vulnerable to duplication. The last slot of the buffer period showed a significant temperature rise on both sensors (HOT condition), confirming the buffer captured actual temperature changes during the outage rather than static values.

Trial 2, as a replication, produced the same outcome (175/175 for both sensors, no duplication or gaps), along with two minor anomalies the system handled without affecting the result. A transient HTTP failure on S2 (error code -11, connection failed to establish) was immediately followed by the next transmission, keeping data intact at 175/175. A 16-second gap in the S2 buffer timestamp, possibly from an overlapping connectivity-check cycle, stayed far below the 5-minute slot resolution and had no effect on the display. As an operational note, phpMyAdmin also needs a max_input_vars adjustment for full-scale use (about 576 rows/day at PT Findora Internusa's rate of 8 processes/day), which doesn't affect data integrity itself.

Summary of Parameter 5. The DataMerger algorithm was validated across two independent 30-31 minute trials with no duplication or gaps on the dashboard, including at the real-time-to-buffer transition point, and the minor anomaly in Trial 2 did not affect final data integrity. DataMerger and SyncManager were also exercised indirectly across every other synchronization-related test in this study, since any data reaching the dashboard, from the near-capacity and overflow buffer trials (Section III-C) to the server-down, database-down, and duplicate-payload trials (Section III-D), passes through the same slot-alignment and retry logic. No duplication, gap, or ordering error appeared in the dashboard across any of these scenarios, extending the evidence well beyond the two dedicated trials.

H. Inter-Sensor Consistency and Independence

Beyond the five main parameters in the proposal, an additional test was conducted to validate the consistency and independence of the two DS18B20 sensors that share the same 1-Wire bus, since both communicate over a single physical line. Under stable room-temperature conditions

(± 26 – 27°C , 12 measurement points), the two sensors showed an average difference of 0.168°C (maximum 0.250°C , minimum 0.120°C), well below the DS18B20's rated tolerance of $\pm 0.5^{\circ}\text{C}$ [20].

Under dynamic heating conditions, the reading gap briefly increased during the fast temperature-rise phase, then narrowed back toward baseline once the temperature stabilized, as presented in Table XVI.

TABLE XVI
SENSOR READING DIFFERENCES UNDER DYNAMIC CONDITIONS

Metric	Trial 2 Rising Phase	Trial 2 Stable Phase	Trial 3 Rising Phase	Trial 3 Stable Phase
Average S1 vs S2 difference	3.316°C	1.260°C	0.544°C	0.443°C
Maximum difference	7.880°C	-	4.060°C	-
S1 peak temperature	59.38°C	-	52.88°C	-
S2 peak temperature	61.50°C	-	53.56°C	-

This pattern can be reasonably explained by physical factors: differences in the speed or position at which the two sensors were immersed in the hot water may have caused one sensor to be thermally exposed slightly sooner, but once the temperature equalized, the difference returned to a range of 0.12 – 0.19°C , close to the stable-condition baseline. In Trial 2, a single point of exact convergence was also observed (at 14:07:34), where both sensors read an identical value of 58.94°C . To validate the independence of the two sensors on the same 1-Wire bus, an additional trial was conducted in which only Sensor 1 was heated (from 25.81°C to 35.19°C). At the same time, Sensor 2 was left at room temperature (stable at 25.62 – 25.69°C), demonstrating that the two sensors' readings remain independent despite sharing the same physical bus. In addition, both sensors successfully buffered and uploaded all 12/12 data points (100%), indicating that data acquisition remained unaffected even when one sensor experienced a significant temperature change.

Summary of Inter-Sensor Consistency and Independence. Both DS18B20 sensors were shown to be consistent (within manufacturer specifications) and electrically and logically independent on the same 1-Wire bus, supporting the reliability of the dual-sensor architecture for simultaneous use.

I. Cross-Validation Against an Independent Reference Thermometer

To address the limitation that agreement between the two DS18B20 sensors alone does not establish accuracy against true temperature, both sensors were cross-validated against two independently sourced commercial digital thermometers (TPM-10, Unit A and Unit B), immersed together with S1 and S2 in the same water medium. Since the TPM-10 units carry no formal, traceable metrology certification, this is described as cross-validation against a reference-grade commercial

instrument rather than calibration. Both units were first checked against a known physical reference, boiling water (≈ 99 – 100°C), Unit A read 0.47°C low and Unit B 1.00°C low, both within the manufacturer's $\pm 1^{\circ}\text{C}$ claim though Unit B sat right at that boundary; the two units also disagreed with each other by 0.53°C , a systematic offset rather than noise, later used as a secondary reliability check in place of a freezing-point verification that was not practically achievable.

An initial attempt at cross-validating S1 and S2 by letting the water cool passively was discarded: the gap between DS18B20 and TPM-10 shrank steadily from $+2.00^{\circ}\text{C}$ at 74°C to -0.35°C at 50°C , inconsistent with a fixed accuracy bias and consistent instead with thermal lag, confirmed directly when TPM-10 dropped to 35°C while DS18B20 still read 40°C five seconds after both probes were lifted from the water at $\approx 45^{\circ}\text{C}$. This delay explains why the circular buffer's actual sampling interval slightly exceeded its coded 10-second target, averaging about 10.7 seconds per cycle rather than matching it exactly.

The trial was repeated using active temperature-holding, yielding three stable points (60.75 – 79°C); lower temperatures could not be stabilized within the time available for this self-conducted test. Table XVII summarizes the result.

TABLE XVII
DS18B20 BIAS RELATIVE TO TPM-10 REFERENCE (STABLE-STATE READINGS)

Stable Point	Bias S1 ($^{\circ}\text{C}$)	Bias S2 ($^{\circ}\text{C}$)	TPM-A vs TPM-B difference ($^{\circ}\text{C}$)
$\sim 79^{\circ}\text{C}$	-0.50	-0.30	0.60
$\sim 70.45^{\circ}\text{C}$	+0.25	+0.05	0.30
$\sim 60.75^{\circ}\text{C}$	+0.05	-0.45	0.50
Mean	-0.067	-0.233	0.467
MAE	0.267	0.267	-
RMSE	0.324	0.314	-

Both sensors' bias stayed within -0.50°C to $+0.25^{\circ}\text{C}$, inside the $\pm 0.5^{\circ}\text{C}$ manufacturer specification, with no consistent direction, suggesting random variation rather than systematic offset; TPM-10 inter-unit disagreement (0.467°C) closely matched the boiling-point result (0.53°C), reinforcing that this is a real but modest discrepancy between reference units. Coverage was limited to 60.75 – 79°C , the lower operating range (31 – 50°C) could not be stabilized with this method, each point was recorded once rather than three times as planned, and TPM-10's own $\pm 1^{\circ}\text{C}$ accuracy contributes to the combined uncertainty, so these results reflect accuracy relative to a verified but non-traceable commercial reference rather than an absolute metrological guarantee.

J. Security and Operational Considerations

As a research prototype rather than a production system, several security and operational safeguards were intentionally left out of scope. None of the REST API endpoints require authentication, CORS is fully open, and the log-clearing endpoint can truncate the entire temperature table without

access control, worth flagging since this data may later serve as a quality or compliance record. Input validation is minimal, arbitrary values are accepted for fields like batch identifier and data type, and an empty or malformed reading gets stored as 0.0 instead of rejected. The system also runs over plain HTTP without TLS, exposed internal server file paths in error responses observed during testing (Section III-D), and the dashboard has no login or access restriction. The one exception is duplicate protection, already enforced by the sensor-time uniqueness constraint (Section III-D), introduced to solve a synchronization problem rather than a security one but happening to reject a resent or replayed entry too, a reminder that resilience against unintended repetition is not the same guarantee as resilience against deliberate misuse, and only the former was addressed here.

A production deployment would therefore need, at minimum, API authentication, server-side input validation, encrypted transport, restricted database credentials in place of the default root account, disabled verbose error output, dashboard access control, and a scheduled database backup, since the buffer's protection against data loss ends once a reading reaches the database, with no safeguard against loss of the database itself. These are noted as future work rather than implemented features, since security hardening fell outside this study's focus on network-layer resilience. The most critical of these, authentication, input validation, encrypted transport, and database backup, are revisited as concrete development directions in the General Discussion below.

K. Summary of Testing Results and Proposal Targets

Table XVIII summarizes how each testing parameter performed including the additional tests.

TABLE XVIII
TESTING RESULTS RECAP

Testing Aspect	Proposal Target	Research Result
Real-time data latency	< 2 seconds	Average 0.6–0.7 s; maximum 2 s; P95/P99 = 2 s, n = 10 per sensor (target met)
Zero data loss while offline (short outages, 2–20 min)	0 data loss	199/200 (99.5%) in the 3 initial trials on pre-remediation firmware; the single loss traced to the since-remediated silent-HTTP-200 defect
Zero data loss, buffer capacity limits	0 data loss	Near-capacity (± 55 min): 81.4% pre-remediation $\rightarrow$ 309/309 (100%) post-remediation. Overflow (± 87 min): 360/360 retained slots synced; oldest ± 23 min overwritten by FIFO by design
Buffer behavior on restart/power loss	Not a proposal target	Data buffered before restart lost in 4/4 runs (volatile SRAM); post-restart data 0% loss, 0 duplicates; valid timestamps on recovery only

		1/4 (soft restart) and 0/20 (hard restart)
Catch-up sequence integrity	Ordered & complete	Met in all trials; 0 duplicates and no ordering errors, including after the idempotency fix (32/32)
Duplicate protection (idempotency)	No duplicates	Before remediation, 3 identical payloads gave 3 rows. After remediation, 10 requests gave 5 unique rows and 5 rejected, all HTTP 200
REST API failure handling (server down / DB down)	Not a proposal target	Server-down: 104/104 points, 0 retries. DB-down realtime: 6/6 lost $\rightarrow$ 0 lost. Staged recovery: 0/36 $\rightarrow$ 60/60, 512 s $\rightarrow$ 22 s
Zombie WiFi detection & recovery	Automatic, no intervention	Confirmed across two behavior modes (full force-reconnect and adaptive near-threshold)
Inter-sensor consistency	Within $\pm 0.5^\circ\text{C}$	Average difference 0.168°C (stable); up to 7.88°C during rapid heating, back to baseline once stable
Accuracy vs reference thermometer	Not a proposal target	Bias -0.50 to $+0.25^\circ\text{C}$. MAE 0.267. RMSE 0.32 (S1) / 0.31 (S2); only 60.75–79°C covered
REST API uptime (6 hours)	> 99%	Average of 2 runs: S1 99.32%, S2 99.49%. S1 in the busy-hours run was 98.73%
DataMerger (idempotent, gap-free)	No duplicates/gaps	Met in 2 trials (30–31 min); no duplication or gaps in any other scenario either

L. General Discussion

Overall, the testing results in Sections III-A through III-G address the research problem established at the proposal stage. The system's ability to read and present data from both sensors in real time through the REST API and the local display was confirmed through Parameter 1. The offline buffering mechanism, based on the circular buffer and the catch-up synchronization algorithm, was confirmed by Parameters 2 and 3, with an aggregate success rate of 99.5% across the three offline duration scenarios tested. The dual-layer Zombie WiFi detection mechanism was shown to function under two distinct behavior modes, both when conditions reached the critical threshold and when borderline conditions self-recovered. The DataMerger- and SyncManager-based dashboard was confirmed to be free of duplicates and gaps across two independent trials. Overall system performance is presented in Table XVIII. Under the tested conditions, the proposal targets for latency, catch-up integrity, Zombie WiFi recovery, inter-sensor consistency, REST API uptime, and DataMerger integrity were achieved, either directly or via the average of two independent test runs. The zero-data-loss target was met for outages within the

buffer's capacity once the identified defects were corrected, but not unconditionally: readings buffered in volatile SRAM are lost on a restart or power loss, and an outage longer than the buffer's capacity (± 60 minutes) overwrites the oldest entries by design.

Against earlier IoT monitoring studies, the system's 0.6–0.7-second real-time latency sits above the 347–383 ms reported for edge-based AIoT inference [9], but still comes in under the 1.87-second latency of a multi-sensor agroclimatic system [10] and the 686.44 ms cloud-to-device latency of an HTTP/MQTT smart building system [12], placing its responsiveness roughly on par with these deployments despite the different network topologies involved. For offline recovery, the 99.5% success rate (199 of 200 points) achieved in the initial trials slightly exceeds the 98.62% transmission success reported for a custom RF-based factory monitoring system [5] and the post-remediation trials recovering all retained data lines up with store-and-forward approaches in wearable healthcare systems that held continuity through disconnections up to 30 minutes [13]. The average REST API uptime of 99.32–99.49% also stands well above consumer IoT reliability more broadly, where 88% of tested devices failed to meet at least one basic reliability criterion [3], a gap that reflects the practical benefit of the layered resilience mechanisms built into this system.

1) *Research Contributions:* This research makes technical contributions by documenting and validating a circular buffer algorithm for offline data persistence on memory-constrained microcontrollers, a SyncManager–DataMerger architectural pattern that offers a structured approach to widespread IoT data synchronization issues, and a dual-layer Zombie WiFi detection and recovery strategy that adapts to the severity of network disruptions. The study also uncovered an alternating timeout characteristic in the single-threaded ESP32 firmware architecture when handling multiple endpoints simultaneously. This finding could prove useful for developers of other ESP32-based IoT systems with similar requirements.

2) *Research Limitations:* These results carry several boundaries worth stating plainly. Quantitative testing was confined to a local network simulating the facility's conditions, with hot-water immersion peaking at $\pm 61^{\circ}\text{C}$ rather than the historical maximum of 74°C , and it did not reproduce chamber-level humidity, dust, or electromagnetic interference. A preliminary functional showcase on-site confirmed the system runs as intended in the field, but it does not substitute for validation inside an operating chamber, and the planned enclosure and cable-mounting arrangement (Section II-E) remains unrealized. Buffer-capacity and restart behavior were likewise characterized only through simulated disconnection rather than a genuine multi-hour chamber outage, and each parameter was tested across just 2 to 4 trials, which is sufficient for functional validation but limited for statistical generalization. The zero-data-loss target held at 99.5% (199 of 200 points) rather than absolutely, and its one

transient failure was traced to a firmware defect since corrected (Section III-C). Authentication and transport encryption were left out of scope by design, and the cause of the simultaneous dual-sensor failures behind 35% of Run 1's uptime losses (Section III-F) remains unconfirmed, with WiFi congestion standing as an untested hypothesis.

3) *Future Development Directions:* Closing these gaps points toward several concrete next steps. Chief among them is validation under real chamber conditions and real operating temperatures, with more repetitions per parameter for stronger statistical grounding and the physical mounting finally realized. The alternating-timeout pattern behind Parameter 4 points toward either a FreeRTOS-based multitasking rewrite, shown elsewhere to cut transmission intervals by up to 37.5% in comparable architectures [21], or a more targeted swap of the synchronous WebServer library for an asynchronous alternative such as ESPAsyncWebServer. The unexplained simultaneous-failure cases would need separate mitigation such as WiFi channel optimization or migration to a wired connection like W5500, which would also remove the need for Zombie WiFi detection altogether [2]. Production readiness would further call for the safeguards discussed in Section III-J, among them authentication, input validation, encrypted transport, restricted database credentials, access control, and scheduled backup, along with migrating buffer storage and dashboard hosting to a cloud server. The cloud and wired-network directions reflect direct input from the facility's IT team during the author's internship.

IV. CONCLUSION

This research developed an IoT-based temperature monitoring system for the heat treatment process at PT Findora Internusa, built on a three-layer architecture (hardware, backend, frontend) connected through HTTP/REST, with five key technical designs to maintain data recording reliability amid connectivity disruptions, namely circular buffer, Zombie WiFi detection and recovery, catch-up synchronization, DataMerger, and SyncManager.

Based on the five testing parameters, the system met the proposal targets under the tested conditions, except that zero data loss held only for outages within the buffer's capacity (about one hour) and without a device restart. Offline buffering and catch-up synchronization recovered 199 of 200 buffered points (99.5%) in the initial trials, the single failure being attributed to a since-corrected firmware defect (Section III-C), and all buffered data in the later capacity trials. Zombie WiFi detection responded correctly to both critical and self-recovering disruptions. REST API uptime averaged 99.32% (Sensor 1) and 99.49% (Sensor 2) across the day and night shifts, DataMerger produced no duplication or gaps in any scenario, and inter-sensor consistency stayed within $\pm 0.5^{\circ}\text{C}$ under stable conditions. Additional tests on thermometer cross-validation, extreme offline buffering, and REST API

failure and idempotency handling (Section III) supported these results.

Beyond the five parameters, this research uncovered alternating timeouts in the single-threaded ESP32 firmware when handling multiple endpoints. The system's real-time latency, offline recovery rate, and REST API uptime (Section III-L) match or exceed results reported for comparable industrial, agricultural, and consumer IoT systems, especially in device-level reliability. Besides solving a practical problem for the company, this work contributes a tested network resilience pattern combining circular buffering, dual-layer disruption detection, catch-up synchronization, and the SyncManager-DataMerger architecture, as a starting point for similar industrial IoT systems.

Testing remained confined to a local network rather than an actual heat treatment chamber, buffer capacity and restart behavior were tested only through simulated disconnection, and repetitions per parameter are still limited for strong statistical generalization. Further development should include validation inside an actual chamber, FreeRTOS-based handling of concurrent requests, authentication and transport encryption, and migration to a cloud server and wired (Ethernet) connectivity as suggested by the company's IT team.

These findings hold within a scope bounded by a simulated network, a hot-water proxy for chamber heat, and a limited number of trials per parameter. Closing that gap through chamber-level validation, FreeRTOS-based concurrency handling, and stronger security and infrastructure remains the clearest path toward field deployment.

REFERENCES

- [1] G. Cangondo, R. P. O. Pinto, and B. M. C. Silva, "Incentive-based reputation and cooperation mechanisms for intermittent IoT networks: A survey," *Ad Hoc Netw.*, vol. 190, p. 104276, Sep. 2026, doi: 10.1016/j.adhoc.2026.104276.
- [2] S. Hassan *et al.*, "Real-Time Investigation of Cross-Technology Interference in Heterogeneous IoT Networks," *IEEE Access*, vol. 11, pp. 112223–112235, 2023, doi: 10.1109/ACCESS.2023.3321221.
- [3] S. Lazzaro, V. De Angelis, A. M. Mandalari, and F. Buccafurri, "A black-box assessment of authentication and reliability in consumer IoT devices," *Pervasive Mob. Comput.*, vol. 110, p. 102045, May 2025, doi: 10.1016/j.pmcj.2025.102045.
- [4] A. Lamberty, H. Sundmaeker, G. Große Hovest, M. Dechert, and J. Kreyenschmidt, "Development of a generic implementation concept for autonomous IoT-based temperature monitoring in multi-stakeholder food supply networks," *Sustain. Futur.*, vol. 11, p. 101890, Jun. 2026, doi: 10.1016/j.sftr.2026.101890.
- [5] C. D. Vo, T.-T. Nguyen, and D. A. Dang, "A low-cost IoT architecture with RF communication and data analytics for real-time productivity monitoring in garment manufacturing," *iScience*, vol. 29, no. 1, p. 114536, Jan. 2026, doi: 10.1016/j.isci.2025.114536.
- [6] A. K. Trivedi, P. Jain, and G. S. Chhabra, "ParkSync: IoT-Based Smart Parking Availability Tracker & Billing System," *Procedia Comput. Sci.*, vol. 282, pp. 2500–2508, 2026, doi: 10.1016/j.procs.2026.06.020.
- [7] C. M. Nkinyam, C. O. Ujah, C. O. Asadu, B. Anyaka, and P. A. Olubambi, "Development of a low-cost monitoring device for solar electric (PV) system using internet of things (IoT)," *Results Eng.*, vol. 28, p. 107324, Dec. 2025, doi: 10.1016/j.rineng.2025.107324.
- [8] R. Van Dinter *et al.*, "Reference Architecture for IoT-Based Predictive Maintenance Systems Using Digital Twins," 2024, SSRN. doi: 10.2139/ssrn.5014879.
- [9] L. Li, C. Z. Liu, K. Cao, Z. Zhou, and M. Alrowaily, "Artificial intelligence of things for smart agriculture with optimal embedded edge intelligence," *Comput. Electr. Eng.*, vol. 133, p. 111039, May 2026, doi: 10.1016/j.compeleceng.2026.111039.
- [10] S. S. A. Begum, S. Mande, S. Jameer, D. Veeraiah, Md. Amanatulla, and P. Eswaraiyah, "An IoT and Sensor-Based Intelligent Framework for Data-Driven Precision Agriculture and Sustainable Resource Management," *Frankl. Open*, p. 100726, Aug. 2026, doi: 10.1016/j.fraope.2026.100726.
- [11] M. Kashyap, A. K. Dev, and V. Sharma, "Implementation and analysis of EMQX broker for MQTT protocol in the Internet of Things," *E-Prime - Adv. Electr. Eng. Electron. Energy*, vol. 10, p. 100846, Dec. 2024, doi: 10.1016/j.prime.2024.100846.
- [12] R. K. Sharma and G. Verma, "Design and analysis of an automated power control system for smart buildings using IoT communication protocols," *Results Eng.*, vol. 30, p. 110299, Jun. 2026, doi: 10.1016/j.rineng.2026.110299.
- [13] D. P. Kartikasari, A. Tobing, A. Bhawiyuga, A. Kusyanti, and N. Hevy, "A store-forward method for biosignal acquisition in smart healthcare system utilizing wearable IoT device," *Kinet. Game Technol. Inf. Syst. Comput. Netw. Comput. Electron. Control*, vol. 6, no. 1, pp. 51–58, Feb. 2021, doi: https://doi.org/10.22219/kinetik.v6i1.1154.
- [14] M. Marin-de-Yzaguirre, O. Fuste, and J. A. Ruiz-de-Azua, "Study to integrate Delay-Tolerant Network protocols in IoT LEO constellations for flood prevention," *Acta Astronaut.*, vol. 225, pp. 968–977, Dec. 2024, doi: 10.1016/j.actaastro.2024.09.004.
- [15] R. Tiberti *et al.*, "Automated high frequency monitoring of Lake Maggiore through in situ sensors: system design, field test and data quality control," *J. Limnol.*, vol. 80, no. 2, Jun. 2021, doi: 10.4081/jlimnol.2021.2011.
- [16] I. Zyrianoff, L. Gigli, F. Montori, L. Sciallo, C. Kamienski, and M. Di Felice, "CACHE-IT: A distributed architecture for proactive edge caching in heterogeneous IoT scenarios," *Ad Hoc Netw.*, vol. 156, p. 103413, Apr. 2024, doi: 10.1016/j.adhoc.2024.103413.
- [17] L. Castro García *et al.*, "I-Canopy: A resilient IoT platform for adaptive environmental monitoring at the edge," *Smart Agric. Technol.*, vol. 12, p. 101611, Dec. 2025, doi: 10.1016/j.atech.2025.101611.
- [18] L. Zeping, N. Abdullah, and M. K. Ishak, "Smart greenhouse climate control with real-time fault detection and energy-aware automation," *Smart Agric. Technol.*, vol. 13, p. 101707, Mar. 2026, doi: 10.1016/j.atech.2025.101707.
- [19] N. Chamara, Y. Ge, and S. Russo, "FireLog: An open-source, low-cost system for temperature logging during wildland fires with high spatial and temporal resolution," *HardwareX*, vol. 24, p. e00722, Dec. 2025, doi: 10.1016/j.ohx.2025.e00722.
- [20] M. Todd, A. J. E. Gallant, A. Wang, J. Plucinski, and V. N. L. Wong, "Quantifying inter- and intra-sensor variability in low-cost soil moisture and soil temperature sensors: A comparative study," *Smart Agric. Technol.*, vol. 12, p. 101186, Dec. 2025, doi: 10.1016/j.atech.2025.101186.
- [21] E. Alves, N. Silva, W. Júnior, A. Moares, N. Silva, and A. Balieiro, "FreeRTOS Application in a Real-Time Air Quality Monitoring Architecture for Smart Campus," *IEEE Lat. Am. Trans.*, vol. 23, no. 4, pp. 266–273, Apr. 2025, doi: 10.1109/TLA.2025.10930347.