Anais do Congresso Brasileiro de Iniciação Científica
(https://periodicos.unisanta.br/COB)

ISSN 2317-1316

DEVELOPMENT AND EVALUATION OF RESILIENT OVER-THE-AIR UPDATES FOR IoT DEVICES

DESENVOLVIMENTO E AVALIAÇÃO DA RESILIÊNCIA DE ATUALIZAÇÕES OVER-THE-AIR EM DISPOSITIVOS IoT

Eric Denser Alencar, Vitor Kenzo Martins Ochida, Matheus Cardoso Kroeff, Fabio Kawaoka Takase, Leandro Carlos Fernandes, Fabio Silva Lopes, Bruno da Silva Rodrigues

Mackenzie Presbyterian University – School of Computing and Informatics
E-mail para contato: eric.alencar@mackenzie.br

RESUMO – A crescente adoção da Internet das Coisas (IoT) exige mecanismos capazes de garantir a atualização remota e a disponibilidade de dispositivos distribuídos geograficamente. Este trabalho apresenta e avalia uma arquitetura de atualização de firmware por meio de Over-the-Air (OTA), integrada a uma plataforma de gerenciamento de estações meteorológicas. A solução incorpora mecanismos de recuperação e rollback automático de firmware para aumentar a resiliência diante de falhas de comunicação e software. Testes experimentais avaliaram o comportamento do sistema sob condições adversas, considerando métricas de latência e throughput. Os resultados demonstraram a capacidade da arquitetura de recuperar dispositivos por meio de mecanismos de rollback e manter a continuidade operacional.

Palavras-chave: IoT; OTA; Cidades Inteligentes; Recuperação de falhas; Continuidade Operacional.

ABSTRACT – The growing adoption of the Internet of Things (IoT) requires mechanisms to ensure remote updates and the availability of geographically distributed devices. This work presents and evaluates an Over-the-Air (OTA) firmware update architecture integrated into a weather station management platform. The solution incorporates recovery mechanisms and automatic firmware rollback to improve resilience against communication and software failures. Experimental tests evaluated system behavior under adverse conditions, considering latency and throughput metrics. The results demonstrated the architecture's ability to recover devices through rollback mechanisms while maintaining operational continuity.

Keywords: IoT; OTA; Smart Cities; Fault Recovery; Operational Continuity.

ISSN 2317-1316

1 INTRODUCTION

As the Smart Cities paradigm continues to evolve, the IoT has become increasingly important for optimizing urban infrastructure by enabling the real-time collection and analysis of data to support decision-making (GANESAN, 2024). However, the widespread adoption of IoT has also introduced significant challenges related to the deployment, connectivity, and management of large-scale heterogeneous devices operating across a wide range of application domains (MIETTINEN, 2018).

With more than 29 billion smart connected devices expected to be deployed worldwide by 2030 (MAURYA et al., 2025), IoT environments may comprise dozens or even hundreds of geographically distributed nodes. In such scenarios, relying on manual intervention to perform tasks such as device reconfiguration, credential updates, and software patching becomes increasingly impractical due to constraints in logistics. Moreover, manual operations are inherently susceptible to configuration errors, which can potentially compromise the security and integrity of the entire trust domain (MIETTINEN, 2018).

A logistic solution for updating geographically distributed IoT devices without requiring physical access is performing OTA updates (De Simone, 2025). In addition to enhancing security and compliance, this practice also reduces operational and maintenance costs (SUDHARSAN, 2022). However, the execution of the OTA process itself introduces availability and fault-tolerance challenges. As noted by Sudharsan (2022), unexpected interruptions during the transmission or flashing of the binary caused by power fluctuations, battery failures, loss of connectivity, or premature reboots can corrupt the system image and render the device unusable (bricking). Furthermore, the firmware must incorporate defensive mechanisms to safely handle unforeseen internal logic errors. To ensure operational continuity and environment robustness, implementing rollback mechanisms becomes worth considering, thereby allowing the device to restore a previous functional state should any anomaly occur during the update procedure.

In this context, this work aims to propose and experimentally evaluate a solution for the remote update and recovery of firmware on IoT devices through OTA mechanisms, contributing to the operational continuity of geographically distributed devices in the presence of communication and software failures. The evaluation focuses on the resilience of the update process, the behavior of the system under fault conditions, the effectiveness of rollback

ISSN 2317-1316

mechanisms, and performance metrics such as update latency, report latency, and download throughput. The proposed solution is integrated into the governance and management system for IoT-based weather stations developed as part of an extension project at the Software Production Environment Studies Laboratory (MackLEAPS) at Mackenzie Presbyterian University (UPM). The weather stations are deployed within the Campus Living Lab initiative.

To present this project, the paper is organized as follows. Section 2 describes the materials and methods used in the study. Section 3 presents the results and discussion. Finally, Section 4 presents the conclusions of the study.

2 MATERIALS AND METHODS

This study is characterized as an experimental development research study with a descriptive approach, whose objective is to design and implement a firmware update system for IoT devices using OTA technology.

Regarding the methodology employed, an exploratory study was initially conducted using scientific databases such as the CAPES Journal Portal, ScienceDirect, and IEEE Xplore, with the aim of identifying research studies, articles, and other publications related to firmware updates for IoT devices. This stage aimed to understand the main concepts and technologies involved in the process of updating IoT devices through OTA.

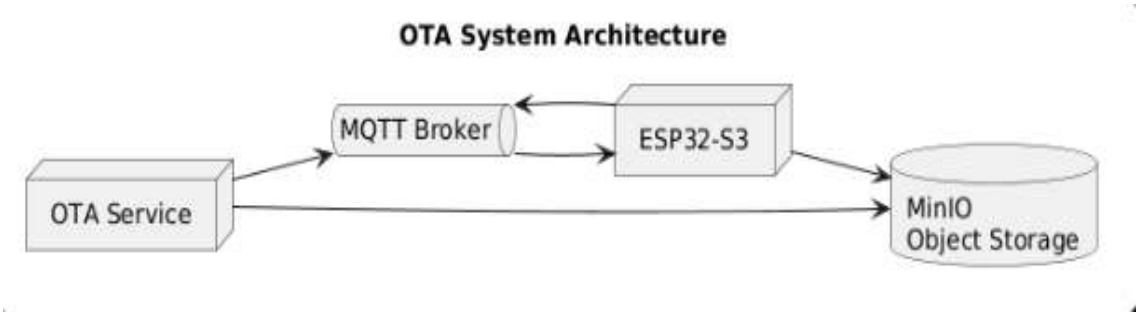
Following the literature review, the hardware to be used in the developed IoT solution, the system architecture, and the experiments to be conducted were defined. The development and experimentation methodology, as well as the resources employed, are detailed in the subsequent sections.

2.1 System Architecture

The proposed OTA update pipeline consists of three main components: (i) an OTA service used to upload, manage, and trigger firmware updates; (ii) a Message Queuing Telemetry Transport (MQTT) broker responsible for distributing commands to the target devices; and (iii) a MinIO object storage instance that stores the firmware binaries and makes them available over HTTP. Figure 1 presents the overall architecture and data flow between these components.

ISSN 2317-1316

Figure 1: OTA System Architecture UML diagram



The infrastructure presented in Figure 1 was deployed using Docker Compose on premises. Eclipse Mosquitto 2 was used as the MQTT broker on port 1883, while MinIO (RELEASE.2025-09-07) was used as an object storage open-source S3-compatible storage server for firmware distribution. Finally, the OTA service was implemented in Java 21 using the Spring Boot framework and communicates with the MQTT broker through the Eclipse Paho MQTT client library.

The update process starts when the OTA service publishes a JSON command to the broker using a device-specific topic following the pattern "command/<device_id>". The message contains the target firmware version, its SHA-256 hash, and a download URL signed by MinIO pointing to the corresponding stored .bin file. The embedded device, permanently subscribed to its designated command topic, receives the payload and autonomously executes the download-and-flash procedure. The complete sequence is illustrated in the sequence diagram presented in Figure 2.

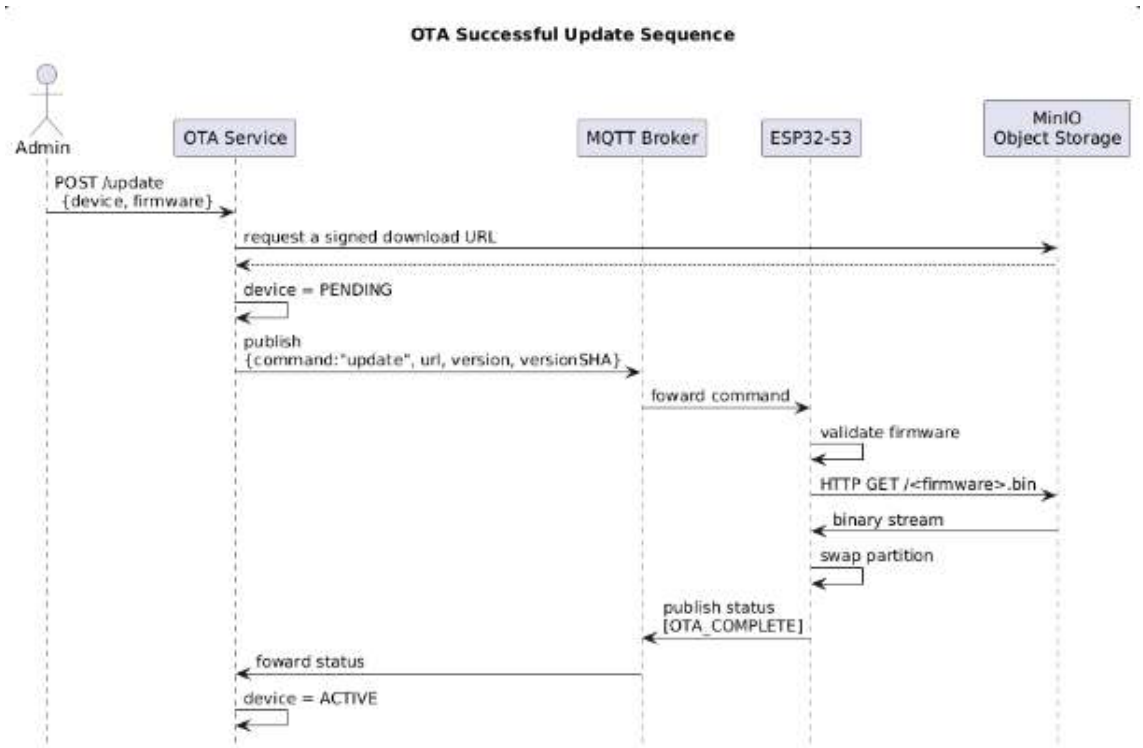
2.2 Hardware and Firmware

All experiments were conducted using an Espressif ESP32-S3 microcontroller featuring a dual-core Xtensa LX7 processor operating at up to 240 MHz, 512 KB of on-chip SRAM, 8 MB of embedded PSRAM, and 16 MB of flash memory. The device provides integrated Wi-Fi 802.11 b/g/n and Bluetooth 5 Low Energy connectivity. The flash memory was partitioned using the ESP-IDF dual-OTA layout, providing two independent application partitions (ota_0 and ota_1) of equal capacity to support firmware updates and rollback mechanisms. One partition stores the active firmware, while the other is used to stage new updates and support rollback. The firmware was developed in C++/C atop the ESP-IDF framework. Additionally, a watchdog timer was employed to detect firmware lockups and automatically reset the device

ISSN 2317-1316

whenever the application became unresponsive, contributing to fault tolerance and system recovery (MEHALAINE et al., 2024).

Figure 2: Successful OTA update sequence diagram



2.3 Test procedure and metrics

To evaluate the reliability and resilience of the proposed pipeline, a set of controlled experiments was performed using three identical ESP32-S3 units, connected on the same network and distributed across the university campus.

Each unit underwent a sequence of OTA update cycles under nominal conditions, followed by a series of fault-injection scenarios targeting the system's recovery mechanisms. All evaluated test scenarios are described in Table 1.

ISSN 2317-1316

Table 1. Scenarios, fault injected and system behavior evaluated during the tests.

ID	Scenario	Fault Injected	Expected Outcome
1	Nominal Update	Without fault injection	*OTA success, new firmware active*
2	Network Interruption	Stop Wi-Fi mid download	*Transfer aborted, session cleaned*
3	SHA-256 blacklisted	Update banned firmware	*Rejected pre-download*
4	Oversized Binary	Image exceeds partition	*Rejected pre-download*
5	Defective Firmware	Infinite loop	*Watchdog trigger and rollback*
6	Guru Meditation	Null-pointer access	*Core panic and rollback*
7	Power cut	Unplug device power source	*Transfer aborted, session cleaned*

Based on the scenarios described in Table 1, the quantitative metrics detailed in Table 2 were analyzed for each scenario.

Table 2. Quantitative metrics analyzed and descriptions.

Metric	Description
Success rate (%)	Ratio of completed firmware installations to total update attempts.
Update latency (s)	Elapsed time from init firmware transfer to complete transferring.
Report latency (s):	Time between the command sent from the OTA Service and the report status published by the device.
Download throughput (KB/s)	*Mean binary transfer rate from MinIO to the device during the flashing phase.*
Rollback latency (s)	*Measures the time the firmware took to become operational again after a rollback.*

To support the experimental evaluation, three Python scripts were developed for data collection and monitoring purposes. These scripts ran concurrently throughout the experiments, recording relevant events in JavaScript Object Notation Line (JSONL) format with Unix timestamps. The collected data were subsequently aggregated into Comma-Separated Values (CSV) files to facilitate analysis and comparison of the results.

The first script, referred to as the MQTT collector, subscribes to the broker topics (commands/#, status/#, error/#) and logs each message with its reception timestamp, providing the temporal reference for latency calculations. The second script, the broker log parser, extracts device connection and disconnection events from the Mosquitto log, classified by cause

ISSN 2317-1316

(socket_error, timeout, reconnect_takeover). Finally, the MinIO log parser extracts GetObject operations from MinIO audit logs, retrieving bytes transferred and request duration for each firmware download, from which throughput is computed directly as tx_bytes / 1024 / duration_s (KB/s).

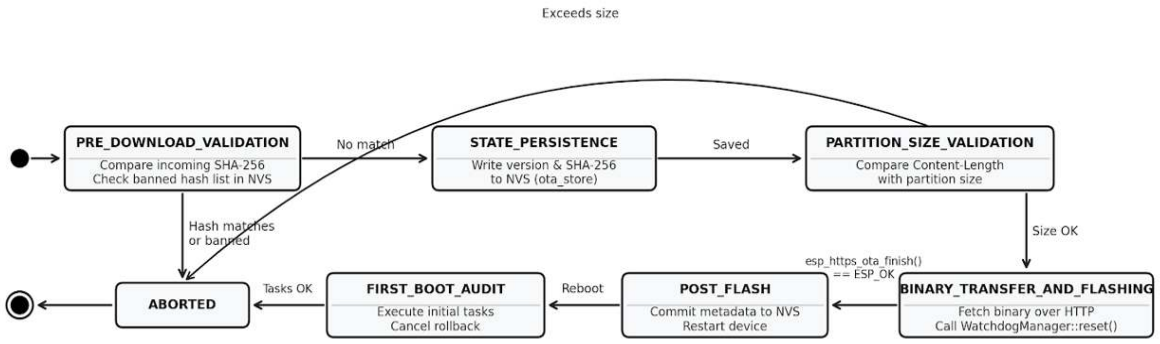
3 RESULTS AND DISCUSSION

This section presents the implementation of the proposed solution, followed by the experimental results and their discussion. For clarity, the section is organized into three subsections: Firmware Implementation, Rollback and Fault Tolerance Mechanisms, and Test Results and Discussion.

3.1 Firmware implementation

As shown in Figure 1 and 2, the OTA firmware update process involves a sequence of interactions between the IoT device and the components responsible for receiving, processing, and executing the update command. The subscribed IoT devices continuously monitor their designated command topics and are able to receive update commands published through the broker. Each device subscribes to its corresponding topic and remains idle until a valid update command is detected. Upon receipt of a valid update command, the CommandProcessor module deserializes the JSON payload and spawns a dedicated task. This task invokes OtaManager::verify_and_ update(), which executes sequential steps presented in Figure 3.

Figure 3: Finite State Machine (FSM) of the OTA firmware update process.



ISSN 2317-1316

As can be seen in Figure 3, the installed firmware executes a routine consisting of five steps:

Pre-download validation: the incoming firmware SHA-256 is compared to the currently running firmware hash to avoid redundant flashing and prevent against a persistent list of banned hashes stored in Non-Volatile Storage (NVS), aborting if either match is found.

State persistence: target firmware version identifier and its respective SHA-256 hash are written to NVS (ota_store) prior to download, allowing post-reboot rollback.

Partition size validation: after esp_https_ota_begin opens the HTTP connection, the image size reported by the server (Content-Length) is compared to the size of the inactive OTA partition. If the image does not fit, the transfer is aborted before any flash sector is erased, keeping the previous firmware intact for rollback.

Binary transfer and flashing: the firmware binary is fetched over HTTP from MinIO using the esp_https_ota API. The download loop periodically calls WatchdogManager::reset() to prevent watchdog resets during long transfers.

Post-flash: upon esp_https_ota_finish() returning ESP_OK, the new version metadata is committed to main_store in NVS and the device restarts into the freshly flashed partition.

First-boot audit: on the first boot following an OTA update, if the device successfully performs its initial tasks, esp_ota_mark_app_valid_cancel_rollback() is called to mark the image as valid and cancel any pending bootloader-initiaded rollback.

3.2 Rollback and Fault Tolerance Mechanisms

In the context of this work, resilience is defined as the ability of a device to automatically recover to an operational state without human intervention after communication, power, or software-related failures. In architectures composed of geographically distributed IoT devices, OTA updates may be affected by intermittent connectivity conditions, bandwidth constraints, and maintenance window limitations, requiring mechanisms capable of ensuring the reliability and consistency of the update process (Hu, 2025). In this context, SHA-256 hashes are used to verify the integrity of downloaded firmware before installation, contributing to secure firmware deployment and continuous verification of software artifacts (Rose, 2020). Furthermore, if a device experiences failures during or after the update process, firmware rollback mechanisms should be available to minimize operational disruptions and prevent gaps in data acquisition.

ISSN 2317-1316

To mitigate operational interruptions and preserve system availability, three distinct rollback modalities are implemented:

Passive rollback: relies on the ESP-IDF bootloader's native image verification mechanism. Upon the first boot following an OTA update, the newly flashed partition is assigned to the state ESP_OTA_IMG_PENDING_VERIFY. If the firmware fails to execute esp_ota_mark_app_valid_cancel_rollback() before a system reset occurs, the bootloader autonomously reverts to the previously validated partition on the subsequent restart.

Active rollback: implemented at the application level. A variable named crash_count is used to track the number of consecutive firmware crashes caused by watchdog resets or panics. The crash count limit defined for this project was 3, which triggers the OtaManager::set_invalid_version(). The SHA-256 hash of the currently running firmware is registered in NVS under the key invalid_sha, permanently blacklisting the image from future installations. The function then calls esp_ota_mark_app_invalid_rollback_and_reboot(), marking the running partition as invalid and immediately restarting the device, upon which the bootloader selects the previous partition as the boot target.

Command-triggered: available as an externally initiated mechanism via a dedicated firmware_rollback command. Upon receipt, CommandProcessor retrieves the previous firmware version and SHA-256 hash from NVS (ota_store), calls esp_ota_set_boot_partition() to designate the alternate OTA partition as the next boot target and issues a controlled restart. This modality enables operators to revert a device to a prior firmware version independently of any detected fault condition.

In all three cases, the post-reboot routine OtaManager::verify_rollback() inspects the partition state via esp_ota_get_last_invalid_partition(). If a rollback is detected, NVS metadata is reconciled and the event is flagged for upstream notification via the MQTT broker.

3.3. Experimental Results and Discussion

This subsection presents the experimental evaluation of the proposed OTA architecture. The tests were conducted using three ESP32-S3 devices distributed across the university

ISSN 2317-1316

campus, aiming to assess the reliability of firmware updates and the effectiveness of the recovery mechanisms under fault conditions. The evaluated scenarios are summarized in Table 1.

Before presenting the results, it is important to characterize the device startup sequence, as it represents a baseline latency component shared by all scenarios involving a device reboot. Upon each cold boot, the device executes the following routine: Wi-Fi association, Dynamic Host Configuration Protocol (DHCP) negotiation, Transmission Control Protocol (TCP) connection to the authentication server for JSON Web Token (JWT) acquisition, MQTT broker connection, and topic subscription. Only after completing this sequence does the device publish its first status message. This startup overhead is inherently present in all latency measurements that follow a reboot event.

Scenario 1 (Nominal Update), as illustrated in the sequence diagram presented in Figure 2, evaluates the interaction between the MQTT broker and the ESP32 during a successful firmware update. A total of 30 OTA update cycles were performed across the three ESP32-S3 devices, covering the complete update workflow. The experiments achieved a success rate of 96.7%, with a single failure due to a network interruption during the download phase (Table 3). The reported latency of 41.2 ± 5.9 s encompasses the entire OTA cycle, corresponding to approximately 23.37 s for binary transfer plus the startup routine overhead following the device reboot.

Table 3. Quantitative results for nominal update

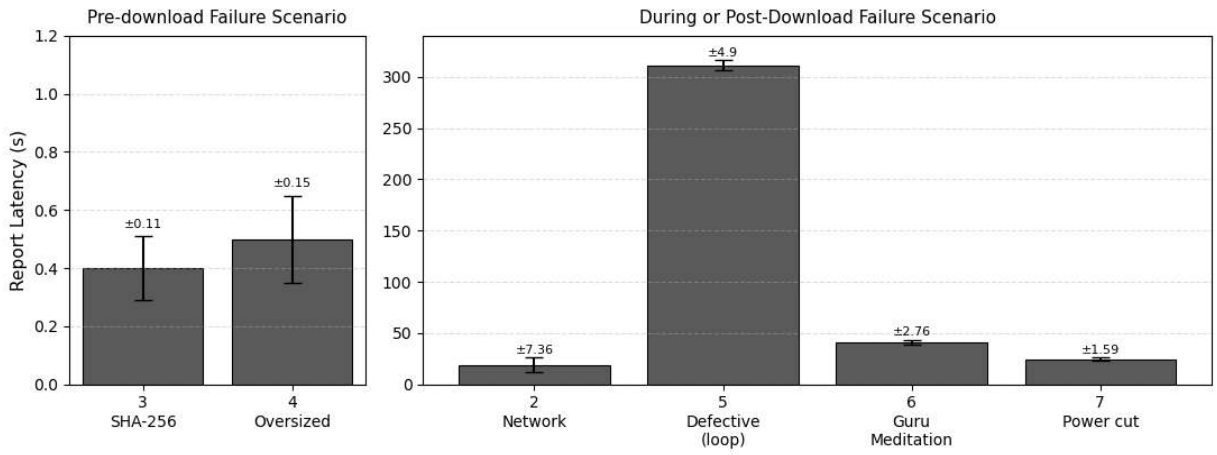
ID	Sample size	Success Rate (%)	Update latency (s)	Download Throughput (KB/s)	Report latency (s)
1	30	96.7%	*23.37 ± 1,05*	*58.9 ± 15,4*	*41.2 ± 5,9*

The remaining scenarios presented in Table 2 (Scenarios 2 to 7) involved the simulation of operational failures during the firmware update process. Due to the nature of these tests, it was not possible to measure the update latency and download throughput metrics for Scenarios 2 to 7. Therefore, the primary metric analyzed was the reported latency, defined as the time elapsed between the update command being sent by the OTA service and the status report

ISSN 2317-1316

published by the device once it becomes operational. Figure 4 presents the reported latency for each of the evaluated scenarios.

Figure 4: Measured Report Latency for each test scenario (mean ± 1 standard deviation).



As presented in Figure 4, all fault scenarios were evaluated through 10 intentionally induced failure events. Based on the reported latency values, the scenarios can be divided into two groups: pre-download failure scenarios and during- or post-download failure scenarios. In the pre-download failure scenarios, the fault is introduced in such a way that the firmware download process is aborted before any data transfer occurs. In contrast, during or post-download failure scenarios involve cases in which the firmware transfer is initiated and the fault occurs either during the download process or after its completion.

The Scenarios 3 and 4 had the lowest report latency among all scenarios. This is expected since both are pre-download validations and reject the update before the download starts, avoiding the full reboot routine. Scenario 3 (SHA-256 blacklisted/banned) presented a reported latency of 0.40 ± 0.11 s, the lowest reported latency among all scenarios since the device rejects the command immediately after an NVS lookup, with no download, no reboot, and no startup overhead. Scenario 4 (Oversized Binary) presented a report latency of 0.49 ± 0.15 s which is also expected, since it is just a simple comparison between the OTA partition size and firmware size, without initiating a full transfer or requiring a device reboot.

Scenarios 2 (Network Interruption) and 7 (Power Cut) involve an interruption during the download process. Scenario 7 presented a reported latency of 24.42 ± 1.59 s, which corresponds almost entirely to the device startup sequence. Scenario 2 presented a reported latency of 18.9

ISSN 2317-1316

± 7.4 s, with the highest variance among all scenarios. This variability is expected given the nature of the fault, as it depends on how long the download had been running before the interruption occurred, as well as on Wi-Fi reconnection, JWT authentication overhead, and broker reconnection. Unlike Scenario 7, the device remains powered and resumes operation from its running state, which explains the lower mean latency compared to the Power Cut scenario.

The only scenarios involving rollback are Scenarios 5 and 6. Their reported latency values were consistent with expectations: Scenario 6 (Null-Pointer Dereference) produced 40.98 ± 2.76 s, comparable to the nominal scenario, since each crash is resolved almost instantaneously. Scenario 5 (Infinite Loop) produced 311.1 ± 4.9 s, also as expected, as each of the three crash cycles requires a full watchdog timeout before the device resets. Regarding rollback latency, Scenario 6 presented 0.57 ± 0.00 s, while Scenario 5 presented 266.0 ± 2.0 s, a difference that directly reflects the watchdog timeout accumulated over three crash cycles, in contrast to a null-pointer crash, which is resolved almost instantaneously by the bootloader without waiting for any timer expiration.

It is important to emphasize that this study was conducted in a controlled local environment. Therefore, some results are specific to the evaluated conditions and may not directly generalize to large-scale deployments. Environments involving dozens or hundreds of devices may introduce additional scalability challenges, including increased broker load, concurrent firmware downloads, and higher infrastructure resource consumption. Furthermore, the experiments were conducted under relatively stable network conditions. Future work should evaluate the proposed architecture in mobile networks, public Internet environments, NAT-restricted connections, and scenarios with higher latency, packet loss, and bandwidth variability to further assess its robustness and scalability.

4 CONCLUSION

This work presented the design, implementation, and experimental evaluation of a firmware OTA update system for IoT devices, integrating an MQTT broker, a MinIO object storage instance, and an ESP32-S3 embedded firmware developed in C++ atop the ESP-IDF framework. The proposed architecture demonstrated reliable update delivery and effective fault tolerance across seven distinct test scenarios, while incorporating complementary recovery mechanisms, including passive rollback, active rollback, and command-triggered rollback, to

ISSN 2317-1316

improve device resilience and operational continuity. Under nominal conditions, 30 trials achieved a 96.7% success rate, with a mean report latency of 41.2 ± 5.9 s encompassing the full OTA cycle including device reboot and startup sequence. Pre-download rejection scenarios (SHA-256 blacklisting and oversized binary) responded in under 0.5 s without requiring a device reboot, confirming the effectiveness of early validation. Fault scenarios involving network interruption and power cut achieved 100% correct abort behavior, with report latencies dominated by reconnection and startup overhead. Both rollback mechanisms were successfully validated, the passive rollback (null-pointer dereference) restored device operation in 0.57 ± 0.00 s, while the active rollback (infinite loop) required 266.0 ± 2.0 s, directly reflecting three accumulated watchdog timeout cycles. All experiments were conducted in a controlled local environment, and future work should evaluate the system over public networks and under higher device concurrency, as well as integrate the solution into the full MackLEAPS IoT governance platform.

5 AI USAGE STATEMENT

AI Use Declaration: During the preparation of this work, the tool Claude (Anthropic) was used to assist in drafting and translating sections of the text. The generated content was reviewed, edited, and validated by the authors, who assume full responsibility for the final text.

6 REFERENCES

De Simone A, Turvani G, Riente F. Incremental firmware update over-the-air for low-power IoT devices over LoRaWAN. Internet of Things. 2025;101772. DOI: 10.1016/j.iot.2025.101772.

Ganesan S, et al. IoT-enabled Smart Weather Stations: Innovations, Challenges, and Future Directions. Malaysian Journal of Science and Advanced Technology. 2024;4(2):180-190.

Hu W. Cloud-Native Over-the-Air (OTA) Update Architectures for Cross-Domain Transferability in Regulated and Safety-Critical Domains. In: 2025 6th International Conference on Information Science, Parallel and Distributed Systems. 2025.

Maurya V, et al. Blockchain-driven security for IoT networks: State-of-the-art, challenges and future directions. Peer-to-Peer Networking and Applications. 2025;18.

ISSN 2317-1316

Mehalaine R, et al. Watchdog timer for fault tolerance in embedded systems. Journal Européen des Systèmes Automatisés. 2024;57(6):1713.

Miettinen M, Van Oorschot PC, Sadeghi AR. Baseline functionality for security and control of commodity IoT devices and domain-controlled device lifecycle management. arXiv preprint arXiv:1808.03071. 2018.

Rose S, Borchert O, Mitchell S, Connelly S. Zero Trust Architecture. Gaithersburg, MD: National Institute of Standards and Technology. 2020. (NIST Special Publication 800-207). DOI: 10.6028/NIST.SP.800-207.

Sedrati A, Mezrioui A, Ouaddah A. IoT-Gov: a structured framework for internet of things governance. Computer Networks. 2023;233:109902.

Sudharsan B, et al. OTA-TinyML: Over the Air Deployment of TinyML Models and Execution on IoT Devices. IEEE Internet Computing. 2022;26(3):69-78.

Xing L. Reliability in Internet of Things: Current Status and Future Perspectives. IEEE Internet of Things Journal. 2020;7(8):6704-6721. DOI: 10.1109/JIOT.2020.2993216.