

Implementación de prácticas DevOps para la integración continua y el despliegue continuo de software en sistemas IoT



Martín Miguel Acuña

Director de tesina: Lic. Ricardo Brea

Instituto de Tecnología e Ingeniería
Universidad Nacional de Hurlingham (UNAHUR)

Esta tesina se presenta para el título de
Licenciatura en Informática

A mi madre Carmen, mi hija Emma y mi hermana Jimena, por creer siempre en mí.

Resumen

En los últimos años, Internet de las Cosas (*Internet of Things, IoT*) se fue consolidando como una tecnología disruptiva. Su uso fue creciendo tanto en hogares como en la industria. Teniendo en cuenta el nivel actual de madurez de esta tecnología, es necesario realizar la integración con procesos de gestión y mejora continua que garanticen el correcto funcionamiento de estos dispositivos y a su vez permitan crear productos funcionales, de calidad, seguros y que su obsolescencia no esté definida por la disponibilidad del software necesario para su funcionamiento.

Este trabajo se enmarca en la aplicación de prácticas DevOps en sistemas IoT que permitan optimizar la integración continua y el despliegue continuo (*continuous integration and continuous delivery, CI/CD*) de software. Se propone una arquitectura *on-premise* que incorpore herramientas de desarrollo colaborativo, integración continua y actualizaciones Over-The-Air (OTA), automatizando la entrega de firmware de forma eficiente y segura.

La validación experimental demostró mejoras significativas en la eficiencia de actualizaciones remotas sin intervención humana, en escalabilidad, realizando despliegue simultáneo a múltiples dispositivos con tiempos aceptables y confiabilidad, con altas tasas de éxito y mecanismos de rollback ante fallos, confirmando que la adopción de DevOps mejora el ciclo de vida del software en sistemas IoT sin depender exclusivamente de servicios en la nube.

Palabras clave: Internet de las Cosas, IoT, DevOps, CI/CD, Over-the-air, OTA

Abstract

Over the past few years, the Internet of Things (IoT) has emerged as a disruptive technology, with growing adoption in both residential and industrial environments. Given the increasing maturity of this technology, it becomes essential to integrate it with management and continuous improvement processes that ensure the reliable operation of these devices and enable the development of functional, high-quality, and secure products—without their obsolescence being determined by the availability of the necessary software.

This work focuses on the application of DevOps practices in IoT systems to optimize the continuous integration and continuous delivery (CI/CD) of software. An on-premise architecture is proposed, incorporating collaborative development tools, CI workflows, and over-the-air (OTA) update mechanisms to automate firmware delivery in a secure and efficient manner.

The experimental validation demonstrated significant improvements in the reliability of unattended remote updates, scalability through simultaneous deployment to multiple devices, and overall robustness—achieving high success rates and rollback mechanisms in case of failure. These findings confirm that adopting DevOps practices improves the software lifecycle in IoT systems without relying exclusively on cloud-based services.

Keywords: Internet of Things, IoT, DevOps, CI/CD, Over-the-air, OTA

Índice general

1. Introducción	1
1.1. Descripción del problema	1
1.2. Importancia del problema	2
1.3. Objetivo del trabajo	3
1.4. Metodología utilizada para la investigación	4
2. Estado del arte	5
2.1. Investigaciones recientes sobre DevOps en IoT	5
2.2. DevOps en el desarrollo de software tradicional	6
2.3. Desafíos de implementar DevOps en IoT	8
2.3.1. Limitaciones del hardware en IoT	9
2.3.2. Conectividad limitada y entornos operativos adversos	9
2.3.3. Dependencia de OTA y su adopción limitada	9
2.3.4. Falta de estandarización e infraestructura unificada	10
2.3.5. Seguridad y confiabilidad en las actualizaciones	10
2.4. OTA como facilitador de DevOps en IoT	12
2.5. Beneficios de utilizar OTA para CI/CD en IoT	13
2.5.1. Despliegue automatizado sin intervención manual	13
2.5.2. Escalabilidad y agilidad en la entrega de software	14
2.5.3. Mejora en la seguridad y confiabilidad	14
2.5.4. Reducción del <i>downtime</i> o tiempo de inactividad	15
2.5.5. Mantenimiento más eficiente y menor costo a largo plazo	15
2.6. Limitaciones y desafíos de OTA en IoT	16
2.6.1. Problemas de conectividad y entornos inestables	16
2.6.2. Riesgo de fallos catastróficos durante la actualización	16
2.6.3. Seguridad y autenticación en el proceso OTA	17
2.7. Brecha de conocimiento y justificación de la investigación	17

3. Materiales y métodos	19
3.1. Configuración del entorno inicial de pruebas en macOS	20
3.2. Hardware	20
3.3. Software	20
3.4. Protocolos	22
3.5. Recursos adicionales	23
3.6. Configuración del entorno de pruebas	24
3.7. Implementación del flujo DevOps con OTA	27
3.7.1. Control de versiones y desarrollo colaborativo	27
3.7.2. Despliegue de infraestructura local mediante IaC	28
3.7.3. Automatización del proceso CI/CD de firmware IoT con GitHub Actions	28
3.7.4. Despliegue continuo de firmware mediante OTA: del repositorio al dispositivo	30
3.7.5. Seguridad y confiabilidad en el proceso continuo	32
4. Experimentos y resultados	35
4.1. Tiempo de actualización: OTA vs. manual	35
4.1.1. Actualización OTA	35
4.1.2. Actualización manual	36
4.1.3. Análisis de resultados	36
4.2. Tasa de éxito de las actualizaciones OTA	36
4.2.1. Actualizaciones exitosas	37
4.2.2. Actualizaciones fallidas	37
4.2.3. Análisis de resultados	37
4.3. Consumo de recursos durante el proceso OTA	37
4.3.1. Consumo de CPU	38
4.3.2. Consumo de memoria RAM	38
4.3.3. Análisis de resultados	39
4.4. Actualizaciones masivas con OTA	39
4.4.1. Evaluación experimental	39
4.4.2. Análisis de resultados	40
4.5. Prueba integral del proceso CI/CD con OTA	40
4.5.1. Pruebas unitarias	40
4.5.2. Compilación	40
4.5.3. Despliegue	41
4.5.4. Distribución	41

4.5.5. Análisis de resultados	41
4.6. Análisis comparativo y conclusiones de los resultados	41
4.6.1. Eficiencia de despliegue	42
4.6.2. Confiabilidad y reducción de riesgo	42
4.6.3. Impacto en recursos y operatividad	43
4.6.4. Escalabilidad y gestión centralizada	43
5. Conclusiones y futuras líneas de investigación	45
5.1. Beneficios	45
5.2. Consideraciones	46
5.3. Reflexión final	49
Referencias	51
Apéndice A. Gráficas de consumo de recursos durante actualizaciones OTA	55
Apéndice B. Glosario	67

Índice de figuras

2.1. Proceso continuo de implementación DevOps.	7
3.1. Proceso continuo de implementación DevOps propuesto para sistemas de software IoT. (Fuente: [11]).	19
3.2. Flujo de intercambio de datos MQTT. (Fuente: [16]).	22
3.3. Esquema de particiones OTA. (Fuente: [26])	24
3.4. Flujo de actualización de firmware. (Fuente: [26])	25
3.5. Ventana de perfil de dispositivos en ThingsBoard	25
3.6. Ventana de detalles de perfil de dispositivos en ThingsBoard	26
3.7. Ventana de dispositivos en ThingsBoard	26
3.8. Diagrama de la infraestructura	27
3.9. Diagrama de secuencia del proceso CI/CD de firmware IoT con GitHub Actions	30
3.10. Diagrama de secuencia de actualización mediante OTA.	31
A.1. Uso promedio de CPU - Caso 1	55
A.2. Uso promedio de CPU - Caso 2	56
A.3. Uso promedio de CPU - Caso 3	56
A.4. Uso promedio de CPU - Caso 4	57
A.5. Uso promedio de CPU - Caso 5	57
A.6. Uso promedio de CPU - Caso 6	58
A.7. Uso promedio de CPU - Caso 7	58
A.8. Uso promedio de CPU - Caso 8	59
A.9. Uso promedio de CPU - Caso 9	59
A.10. Uso promedio de CPU - Caso 10	60
A.11. Uso promedio de CPU - Internet connection lost - Caso 11	60
A.12. Uso de memoria RAM - Caso 1	61
A.13. Uso de memoria RAM - Caso 2	61
A.14. Uso de memoria RAM - Caso 3	62

A.15. Uso de memoria RAM - Caso 4	62
A.16. Uso de memoria RAM - Caso 5	63
A.17. Uso de memoria RAM - Caso 6	63
A.18. Uso de memoria RAM - Caso 7	64
A.19. Uso de memoria RAM - Caso 8	64
A.20. Uso de memoria RAM - Caso 9	65
A.21. Uso de memoria RAM - Caso 10	65
A.22. Uso de memoria RAM - Internet connection lost - Caso 11	66

Índice de tablas

2.1. Principios y prácticas DevOps aplicables al desarrollo en sistemas IoT . . .	8
2.2. Desafíos técnicos para la implementación de DevOps en IoT	11
4.1. Desempeño operativo: OTA vs. actualización manual	42
4.2. Consumo de recursos durante la actualización OTA	42

Capítulo 1

Introducción

En los últimos años, Internet de las Cosas (*Internet of Things, IoT*) se fue consolidando como una tecnología disruptiva. Su uso fue creciendo tanto en los hogares como en la industria. Teniendo en cuenta el nivel actual de madurez de esta tecnología, es necesario integrarla con procesos de gestión y mejora continua que garanticen el correcto funcionamiento de los dispositivos IoT y a su vez permitan crear productos funcionales, de calidad, seguros y que su obsolescencia no esté definida por la disponibilidad del software necesario para su funcionamiento.

Las prácticas DevOps nacieron a partir de las metodologías ágiles y tienen como eje central la mejora continua. Esto permite acortar los tiempos y mejorar la calidad del proceso de desarrollo al incorporar retroalimentación continua a lo largo del ciclo de vida del software.

Esta investigación parte de la hipótesis de que la adopción de prácticas DevOps puede optimizar la integración continua y el despliegue continuo de software en dispositivos IoT.

Como alternativa a las soluciones comerciales basadas en la nube, se propone un enfoque local que prioriza la autonomía, el control sobre la infraestructura y la posibilidad de personalizar el entorno según las necesidades de cada organización.

1.1. Descripción del problema

La evolución constante de los dispositivos IoT, sumada a la falta de un enfoque en concreto que estructure un marco de trabajo, genera que los desarrolladores adopten un enfoque reactivo, limitándose a resolver los problemas que surgen en lugar de anticiparse a ellos de manera proactiva.

Es importante tener en cuenta que las soluciones comerciales suelen depender de plataformas en la nube para la gestión de los dispositivos IoT. Estas plataformas presentan restricciones; muchas solo permiten gestionar dentro de una única nube. En muchos casos

esto dificulta la migración a otros entornos, debido a que el código de las aplicaciones está profundamente relacionado con las APIs específicas de cada nube, como señalan Bou Ghantous y Gill [1].

Actualmente, muchos despliegues de firmware no aprovechan las ventajas de la programación over-the-air (OTA), que se refiere a la capacidad de descargar aplicaciones, servicios y configuraciones a través de una red inalámbrica, como define Gartner [2]. Esta tecnología permite corregir errores, solucionar vulnerabilidades, añadir nuevas funcionalidades y reprogramar dispositivos ubicados en áreas remotas, sin necesidad de conexión física. La falta de adopción de OTA conlleva interrupciones prolongadas durante las actualizaciones manuales o, directamente, la imposibilidad de realizarlas.

En este contexto, las prácticas DevOps, específicamente la integración continua y el despliegue continuo (*continuous integration and continuous delivery, CI/CD*), convierten a la implementación de OTA en un componente indispensable dentro del proceso. OTA habilita la actualización remota de los dispositivos sin necesidad de acceso físico, lo que permite desplegar actualizaciones y parches en cuestión de minutos. No hacerlo puede volverlos vulnerables a ataques externos que podrían tener graves consecuencias para los usuarios.

Se parte de la hipótesis de que incorporar metodologías DevOps no solo optimiza el desarrollo de software en IoT, sino que también fortalece su estabilidad operativa y reduce la exposición a fallas o amenazas durante el ciclo de vida del dispositivo.

1.2. Importancia del problema

El uso del término DevOps fue utilizado por Patrick Debois en la Conferencia Agile 2008, destacando la necesidad de una infraestructura ágil y de una mayor interacción entre los equipos de desarrollo y operaciones [3]. "DevOps es el acrónimo de las palabras en inglés *development* (desarrollo) y *operations* (operaciones) que en TI, se refiere a la integración de los equipos de desarrollo, de administración y de operación de sistemas, empleando principios Lean y Agile con el objeto de acelerar las entregas de software"[4]. Esta práctica busca eliminar las barreras tradicionales entre los equipos de desarrollo y operaciones a través de la automatización y la colaboración continua.

Por otro lado, el término Internet of Things (IoT) fue acuñado por Kevin Ashton en 1999 en Procter & Gamble (P&G) y referenciado en su artículo 'That 'Internet of Things' Thing' publicado en RFID Journal en 2009 [5]. La idea de conectar objetos físicos a Internet para recopilar y compartir datos de forma automática ha evolucionado desde entonces hasta convertirse en una tecnología consolidada en múltiples sectores industriales y sociales.

En 2012, la Unión Internacional de Telecomunicaciones (ITU) definió a IoT como “una infraestructura global para la sociedad de la información, que permite servicios avanzados mediante la interconexión de cosas (físicas y virtuales) basadas en tecnologías de información y comunicación interoperables, existentes y en evolución” [6].

Eventos recientes, como la pandemia de COVID-19, aceleraron la adopción de tecnologías digitales, incluyendo IoT, para resolver necesidades emergentes en áreas como salud pública, logística y automatización industrial. Este crecimiento expuso también nuevas limitaciones relacionadas con la escalabilidad, la seguridad y la interoperabilidad de estas soluciones.

Diversos informes, como el IoT Signals - Manufacturing Spotlight de Microsoft (2022), señalan que IoT está modificando profundamente la operación de muchas industrias [7]. Además, proyecciones de Gartner estiman que para el año 2029 habrá más de 15.000 millones de dispositivos IoT integrados en infraestructuras empresariales [8].

En este contexto, la expansión de IoT hacia nuevos entornos como cloud, fog y edge computing, así como su convergencia con gemelos digitales y modelos SaaS, está aumentando la complejidad de los sistemas. Esta evolución tecnológica plantea nuevos desafíos en cuanto al diseño, mantenimiento y actualización del software distribuido en dispositivos con recursos limitados, resaltando la necesidad de enfoques específicos para asegurar su gestión eficiente y sostenible.

1.3. Objetivo del trabajo

El propósito de este trabajo es analizar y validar, mediante un enfoque teórico-práctico, el impacto de la adopción de prácticas DevOps en la eficiencia del ciclo CI/CD en entornos IoT, combinando una revisión estructurada de literatura con una implementación experimental basada en herramientas de código abierto.

En función de lo expuesto, se definen los siguientes objetivos específicos:

- Analizar las limitaciones existentes en los procesos actuales de integración y despliegue de software en dispositivos IoT.
- Identificar los componentes y herramientas necesarias para implementar un proceso de CI/CD con actualizaciones OTA en arquitecturas IoT on-premise, basado en herramientas de código abierto.
- Evaluar el impacto de una solución DevOps en la eficiencia, escalabilidad y confiabilidad del proceso de entrega de firmware en sistemas IoT.

Los resultados obtenidos buscan aportar una base que pueda servir como referencia para futuras investigaciones y desarrollos en el campo de prácticas DevOps aplicado a sistemas IoT.

1.4. Metodología utilizada para la investigación

El presente trabajo adopta un enfoque de investigación aplicada, orientado a resolver un problema técnico concreto mediante la implementación y validación de una solución DevOps para sistemas IoT. El objetivo es diseñar, construir y evaluar una arquitectura que permita la integración continua y el despliegue continuo de software en dispositivos IoT, priorizando el uso de herramientas de código abierto y entornos *on-premise*.

Como base teórica, se realizó una revisión de la literatura, siguiendo los lineamientos generales del método *Systematic Literature Review* (SLR) propuesto por Bárbara Kitchenham [9]. Si bien no se aplicó completamente, se respetaron sus principios fundamentales, ya que proporciona una visión general del estado del conocimiento en el área de estudio.

Como estrategia de búsqueda se utilizaron las siguientes palabras clave:
((“Internet de las cosas” OR “internet of things” OR iot) AND DevOps).

En una primera etapa se utilizó el buscador Google Académico y luego se amplió la búsqueda a IEEE Xplore y ACM Digital Library.

Para seleccionar los documentos relevantes se estableció que sólo serían analizados documentos escritos en español o inglés y publicados entre los años 2005 y 2023.

Además, se aplicó el método de *snowballing*, siguiendo las pautas propuestas por Wohlin [10]. Este método nos permite encontrar estudios referenciados en los documentos iniciales y que no aparecieron en la búsqueda inicial por alguna diferencia en las palabras clave utilizadas. Luego se agregaron estos a la lista.

Como resultado, se seleccionaron diez estudios primarios que sirvieron de base para el estado del arte presentado.

Capítulo 2

Estado del arte

2.1. Investigaciones recientes sobre DevOps en IoT

Múltiples investigaciones han explorado cómo adaptar los principios y herramientas de DevOps al contexto del Internet de las Cosas (IoT), dada su creciente complejidad y evolución constante.

De acuerdo con Pereira et al. [11], la naturaleza distribuida y multidisciplinaria del software IoT ha impulsado la evolución de la ingeniería de software tradicional hacia enfoques más colaborativos y automatizados como DevOps.

El trabajo de Leite et al. [12] destaca que DevOps surge como una extensión de los enfoques ágiles, introduciendo automatización en la entrega y operación del software. Aunque su origen se encuentra en entornos TI convencionales, su aplicación en IoT ha comenzado a ganar relevancia como respuesta a la necesidad de ciclos de vida más cortos, integración continua de hardware y software, y retroalimentación constante desde dispositivos en producción.

Guşeilă et al. [13] indican que "la integración de DevOps en IoT debe contemplar aspectos como la colaboración entre desarrollo y operaciones, la automatización de pruebas y despliegues, y la monitorización continua del sistema". Estos elementos permiten mejorar la eficiencia y confiabilidad de las aplicaciones IoT. Además, enfatizan que combinar estas prácticas con plataformas multi-nube facilita la entrega continua en entornos distribuidos.

Por su parte, López Peña [4] propone formalizar un enfoque DevOps adaptado a las necesidades de los sistemas IoT, en el que se integren "métricas de confiabilidad, retroalimentación rápida y monitorización como componentes centrales". Dentro de esta propuesta, define la actividad denominada "*Retroalimentación Rápida y Continua del Estado de un Sistema IoT*" como una práctica fundamental para asegurar la calidad y estabilidad de los dispositivos en producción.

Asimismo, el mapeo sistemático de Suescún-Monsalve et al. [14] identifica un creciente interés por parte de la industria en adaptar DevOps a los entornos de la Industria 4.0 e IoT, aunque reconoce que aún existe una escasa estandarización de metodologías y herramientas. Este trabajo resalta que los principios de automatización, colaboración interdisciplinaria y entrega continua son cada vez más considerados en desarrollos industriales, aunque su aplicación práctica enfrenta barreras tecnológicas y organizativas.

Finalmente, estudios como el de Bou Ghantous y Gill [1] plantean arquitecturas de referencia para integrar DevOps en aplicaciones IoT desplegadas en múltiples nubes. Su propuesta enfatiza la importancia de la "orquestación de herramientas en un pipeline automatizado" que abarque todas las etapas desde el desarrollo hasta la monitorización en producción.

A partir de lo expuesto, se lleva a cabo un análisis crítico y sistemático del estado del arte, fundamentado en las investigaciones relevadas, con el objetivo de identificar las buenas prácticas en la aplicación de DevOps en entornos IoT, así como también las limitaciones conceptuales, vacíos tecnológicos y desafíos operativos que condicionan su implementación efectiva.

2.2. DevOps en el desarrollo de software tradicional

En el desarrollo de software tradicional, DevOps se apoya en prácticas de Integración Continua (CI) y Entrega/Despliegue Continuo (CD) para automatizar el ciclo de vida de entrega de software (figura 2.1). La integración continua implica fusionar con frecuencia el código de todos los desarrolladores en una rama principal compartida y ejecutar construcciones y pruebas automáticas del sistema [4]. A su vez, la entrega o despliegue continuo extiende este proceso hasta publicar los cambios en los entornos de producción, una vez superadas las pruebas de calidad. Herramientas de orquestación de procesos continuos, por ejemplo, Jenkins, GitLab CI/CD, ArgoCD, GitHub Actions, etc., se utilizan para implementar estas prácticas, coordinando pasos como la compilación, ejecución de pruebas, empaquetado y despliegue de manera repetible y sin intervención manual [12]. CI/CD se ha convertido en un estándar de la industria para lograr despliegues ágiles y confiables de software.

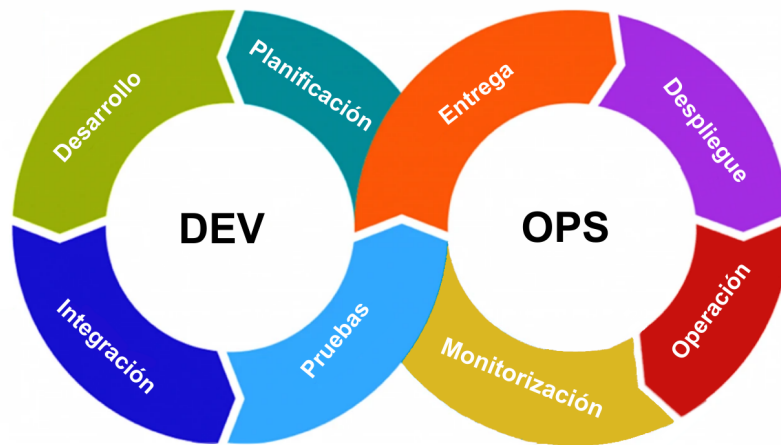


Figura 2.1 Proceso continuo de implementación DevOps.

Los beneficios de DevOps en el software tradicional han quedado demostrados. “El enfoque DevOps proporciona a los equipos un conjunto de prácticas y herramientas para automatizar el ciclo completo de desarrollo, pruebas, despliegue y monitorización, permitiendo entregas más rápidas y sin demoras” [1]. Al eliminar pasos manuales, se reducen errores humanos y se acelera significativamente la frecuencia de lanzamientos. En entornos con pruebas automatizadas, las herramientas DevOps incrementan la cadencia de liberación de versiones a la vez que mejoran la calidad en producción, gracias a la retroalimentación continua [1]. Asimismo, DevOps enfatiza la colaboración y comunicación entre equipos anteriormente aislados, lo que cierra la brecha entre Desarrollo y Operaciones y asegura que la automatización esté alineada con los objetivos de confiabilidad del servicio [1]. En suma, DevOps aplicado en la ingeniería de software convencional ha dado lugar a mayor velocidad de entrega, despliegues más consistentes y sistemas más estables, sentando las bases conceptuales que nos permiten extrapolar estos logros al dominio de IoT.

A partir del análisis realizado en el estado del arte, se identifican principios y prácticas comunes que pueden ser adoptados también en entornos IoT. Estas se presentan de manera resumida en la tabla 2.1.

Tabla 2.1 Principios y prácticas DevOps aplicables al desarrollo en sistemas IoT

Principio	Beneficio	Referencias
Cultura de colaboración y trabajo interdisciplinario	DevOps fomenta la colaboración entre desarrollo y operaciones, eliminando silos organizacionales, clave en IoT debido a su naturaleza multidisciplinaria.	[4, 11, 14, 13]
Prácticas de CI/CD (Integración Continua/Despliegue Continuo)	Permite automatizar la construcción, prueba y entrega de software, logrando entregas rápidas y confiables con identificación temprana de fallos.	[1, 4, 14, 13, 15]
Automatización y uso de infraestructura como código (IaC)	Facilita la gestión de infraestructura mediante código, permitiendo escalabilidad y automatización eficiente de entornos.	[1, 4, 14, 13, 16]
Monitorización continua y retroalimentación	Garantiza calidad al detectar problemas en tiempo real, mejorando rendimiento, seguridad y ciclo de desarrollo.	[4, 11, 13]
Pruebas continuas y automatizadas	Asegura calidad mediante pruebas automatizadas y continuas, adaptadas a sistemas IoT y entornos distribuidos.	[4, 13, 15]
Uso de contenedores	Facilita la ejecución de aplicaciones distribuidas con mayor velocidad e independencia del sistema operativo.	[17, 11, 15]
Uso de microservicios	Permite desarrollar aplicaciones con servicios independientes, mejorando escalabilidad y despliegue.	[4, 11, 15]
Proceso continuo de DevOps	Automatiza y orquesta todas las etapas del desarrollo, permitiendo flujos de trabajo eficientes y optimizados.	[4, 11]

2.3. Desafíos de implementar DevOps en IoT

La adopción de prácticas DevOps en el dominio del Internet de las Cosas presenta una serie de desafíos técnicos y operativos particulares. A continuación, se desarrollan los más relevantes según la literatura. Una visión más completa de los obstáculos identificados se

presenta de forma resumida en la tabla 2.2, junto con sus implicancias técnicas y referencias correspondientes.

2.3.1. Limitaciones del hardware en IoT

Uno de los principales desafíos al implementar DevOps en IoT es que el software no se ejecuta en servidores con recursos casi ilimitados como sería el caso del software tradicional, sino en “dispositivos físicos con capacidades limitadas de cómputo, memoria, almacenamiento y energía”, como señalan Arakadakis et al. [17]. Estos dispositivos suelen ser de bajo costo y bajo consumo, lo que impone restricciones en el diseño del firmware y en su actualización a lo largo de su vida útil. Un paquete de actualización debe ser lo suficientemente pequeño y eficiente como para transmitirse y aplicarse en estos entornos limitados.

2.3.2. Conectividad limitada y entornos operativos adversos

Además de las restricciones de hardware, los dispositivos IoT suelen operar en ubicaciones remotas con acceso intermitente a la red y condiciones ambientales desfavorables. Estas situaciones agravan la complejidad de implementar procesos CI/CD confiables, ya que la conectividad puede presentar latencia elevada, pérdida de paquetes o baja disponibilidad. En consecuencia, los mecanismos de despliegue deben ser resilientes ante fallos de red, cortes de energía y dispositivos que no permanecen siempre conectados. Según [17], estas características deben ser contempladas desde el diseño mismo de la infraestructura de actualización para garantizar su robustez en campo.

2.3.3. Dependencia de OTA y su adopción limitada

Dada la imposibilidad de acceder físicamente a muchos dispositivos IoT una vez desplegados, las actualizaciones OTA se convierten en un componente esencial para aplicar DevOps en este dominio. OTA permite distribuir e instalar software de forma remota a través de conectividad inalámbrica, sin requerir intervención física. “Típicamente, un sistema de actualización OTA se compone de una solución de software embebida en el dispositivo que soporta las secuencias específicas de descarga e instalación, y de una infraestructura en la nube que actúa como servidor central coordinando el proceso”, como describen Villegas et al. [18].

Por ejemplo, un dispositivo puede ejecutar un agente de actualización que consulta periódicamente por nuevas versiones y, si las hay, las descarga y aplica siguiendo un protocolo

definido. Este esquema, común a muchas implementaciones OTA, constituye el puente técnico que permite integrar a los dispositivos IoT dentro del flujo continuo de DevOps.

Sin embargo, la adopción de OTA aún es limitada. Un estudio reciente [18] muestra que, de 26 proyectos de sistemas operativos y software embebido IoT/CPS analizados, solo 7 detallaban mecanismos de actualización OTA, 4 mencionaban soluciones ad-hoc de terceros, mientras que 15 no reportaban ninguna capacidad de actualización. Esto revela que, en la práctica, muchos dispositivos se mantienen con el firmware original salvo intervención manual, lo cual es incompatible con los ciclos de DevOps que requieren actualizaciones frecuentes y automatizadas.

2.3.4. Falta de estandarización e infraestructura unificada

Otro desafío importante es la falta de estandarización en los procesos de despliegue de software para IoT. Históricamente, cada fabricante o proyecto ha desarrollado sus propios mecanismos de actualización, con poca convergencia hacia un estándar común. Aunque existen iniciativas recientes como las normas ISO/IEC 20924 e ISO/IEC 30141 (publicadas en 2018), todavía no hay arquitecturas consolidadas que respondan plenamente a estas recomendaciones [4].

Como consecuencia, los equipos de desarrollo deben lidiar con una multiplicidad de plataformas, protocolos y herramientas dispares, lo que dificulta la integración de un proceso continuo de CI/CD único. Esta heterogeneidad exige esfuerzos de adaptación constantes y conocimientos específicos sobre cada proveedor.

2.3.5. Seguridad y confiabilidad en las actualizaciones

La fiabilidad de las actualizaciones OTA es crítica. En entornos CI/CD tradicionales, una falla puede revertirse rápidamente; en cambio, en IoT, una actualización fallida puede dejar al dispositivo completamente inoperativo. Por esta razón, es indispensable contar con mecanismos de *fail-safe*, que permitan la recuperación del dispositivo ante errores durante el proceso de actualización. Esto incluye funcionalidades como reintentos automáticos, *rollbacks* y verificación de integridad del firmware [17].

La seguridad también debe estar contemplada desde el diseño: “Las actualizaciones OTA deben viajar cifradas y firmadas, y el dispositivo debe verificar la firma digital y la integridad del firmware antes de aplicarlo, garantizando que sólo código legítimo y no alterado se instale en el dispositivo” [17]. Esta necesidad introduce una capa adicional de complejidad al proceso continuo DevOps, ya que es necesario integrar procesos de firmado de firmware, gestión de claves y validaciones criptográficas como parte del flujo continuo de entrega.

Tabla 2.2 Desafíos técnicos para la implementación de DevOps en IoT

Desafío	Implicancia técnica	Referencias
Complejidad y heterogeneidad	La diversidad de dispositivos, protocolos y plataformas en IoT dificulta la integración y gestión de dependencias.	[1, 4, 16, 18]
Dispositivos con recursos limitados	Las limitaciones de hardware restringen la implementación de software y actualizaciones.	[4, 17, 11]
Infraestructura distribuida	La gran cantidad de dispositivos distribuidos introduce desafíos en la gestión, monitorización y orquestación.	[4, 15]
Conectividad limitada e intermitente	Las redes móviles, LoRa o satelitales pueden tener alta latencia o pérdida de paquetes, lo que dificulta la sincronización de actualizaciones.	[17, 18]
Falta de acceso físico a dispositivos	Las actualizaciones deben ser remotas debido a la imposibilidad de intervención física en muchos entornos.	[17]
Seguridad	La exposición a vulnerabilidades hace crucial la automatización de la seguridad.	[4, 17, 11, 16, 18]
Necesidad de adaptaciones específicas	Los procesos de DevOps deben adaptarse a la gestión de firmware y actualizaciones en dispositivos heterogéneos.	[4, 17, 11]
Falta de estandarización en OTA	La ausencia de marcos comunes dificulta la interoperabilidad entre dispositivos y plataformas.	[4, 18]
Ciclos de vida más largos	Los dispositivos IoT operan desatendidos por largos períodos y en entornos restringidos, lo que dificulta las actualizaciones frecuentes y extiende el ciclo de vida del software.	[4, 17]
Baja adopción de actualizaciones OTA	Muchos dispositivos aún no admiten actualizaciones remotas, imposibilitando un flujo CI/CD completo.	[18]

Continúa en la próxima página

Desafío	Implicancia técnica	Referencias
Mayor necesidad de automatización	Debido a la escala y distribución de IoT, la automatización es aún más crítica que en otros entornos.	[1, 4, 14, 13, 15]
Uso de contenedores	Aunque útiles, los contenedores pueden ser difíciles de implementar en dispositivos con recursos limitados.	[4, 15]
Uso de microservicios	La gestión de microservicios en IoT es más compleja debido a las limitaciones de hardware.	[11, 15]
Herramientas y técnicas específicas	El desarrollo de IoT requiere herramientas especializadas para configuración, monitorización y actualizaciones.	[4, 11, 15, 16]
Énfasis en la confiabilidad (trustworthiness)	La confiabilidad es esencial en IoT, abarcando seguridad, privacidad y disponibilidad. El proceso de actualización OTA debe contemplar: tolerancia a fallos, mecanismos de rollback y verificación de integridad.	[4, 17, 18]
Adopción incipiente	DevOps en IoT aún está en desarrollo, con poca adopción de algunas prácticas como IaC y retroalimentación continua.	[4, 11]

2.4. OTA como facilitador de DevOps en IoT

Como mencioné anteriormente, las actualizaciones mediante OTA permiten distribuir e instalar actualizaciones de software de forma inalámbrica en dispositivos IoT, sin acceso físico. Pero este tipo de actualización contrasta con los métodos tradicionales de mantenimiento de firmware. En el pasado y aún en muchos casos actuales sin OTA, actualizar un dispositivo IoT implicaba acceder a él manualmente, conectar un cable para actualizar el firmware o reemplazar el dispositivo in situ por uno nuevo con software actualizado. Esto es claramente inviable a gran escala. “La gran escala de las redes IoT y la ubicación remota de muchos nodos hacen obligatorio el uso de soluciones OTA para actualizar el firmware de forma eficiente” [17]. Sería logísticamente imposible actualizar cientos o miles de dispositivos mediante intervención manual. En contraste, OTA permite que las nuevas versiones de

software viajen por la red hacia los dispositivos, eliminando la necesidad de desplazamientos físicos y habilitando la automatización del despliegue. De esta manera, OTA extiende a IoT las mismas ventajas que en sistemas tradicionales ofrecen la entrega continua: la posibilidad de publicar correcciones y mejoras de forma rápida y centralizada.

Con OTA, los equipos de desarrollo pueden incluir a los dispositivos finales como un entorno más dentro del proceso continuo CI/CD. Por ejemplo, una vez que una nueva versión de software IoT es probada y validada, puede ser distribuida automáticamente vía OTA a todos los dispositivos, del mismo modo a cómo se despliega una aplicación en un servidor de producción. En vez de aplicar actualizaciones esporádicas y manuales, las organizaciones pueden aplicar pequeños incrementos de software con mayor frecuencia, recibiendo retroalimentación inmediata de su comportamiento en el dispositivo real. En la práctica, OTA actúa como el eslabón final del proceso continuo DevOps que conecta el entorno de integración continua con los entornos de operación IoT. Al manejar las actualizaciones de firmware de forma inalámbrica, las actualizaciones OTA hacen posible abordar las necesidades de actualización periódica de los sistemas embebidos de IoT de manera sistemática, similar a como se gestionan las actualizaciones en el software “convencional” [18]. Esto significa que conceptos como despliegues automatizados, *canary releases* o despliegues en fases, comunes en DevOps tradicional, pueden adaptarse al mundo IoT a través de OTA, por ejemplo, actualizando unos pocos dispositivos de prueba, monitoreando resultados y luego gradualmente actualizar el resto de la flota si todo funciona correctamente.

OTA se plantea como un punto clave de DevOps en IoT al permitir la automatización de las actualizaciones de dispositivos a distancia. Al compararlo con los métodos manuales, su ventaja es clara en términos de velocidad y escalabilidad. En la siguiente sección se detallan los beneficios concretos que aporta OTA cuando se integra en procesos CI/CD para IoT.

2.5. Beneficios de utilizar OTA para CI/CD en IoT

Si bien la integración de OTA en sistemas IoT responde a una necesidad técnica crítica para habilitar CI/CD, su adopción también conlleva importantes beneficios en términos de automatización, escalabilidad, seguridad y eficiencia operativa. A continuación, se detallan las principales ventajas que aporta su incorporación al flujo continuo de entrega de software.

2.5.1. Despliegue automatizado sin intervención manual

OTA permite que la distribución de nuevo software ocurra de forma desatendida. Ya no es necesario enviar técnicos para actualizar cada dispositivo; en su lugar, una plataforma central

puede orquestar las actualizaciones de forma remota, incluso sobre una base de miles de dispositivos simultáneamente [17]. Esto reduce drásticamente el esfuerzo y tiempo necesarios para mantener actualizado el parque de dispositivos. En un escenario DevOps, después de integrar y probar cambios en el código, el paso de despliegue puede invocar el proceso OTA para propagar el firmware nuevo automáticamente. Como resultado, las organizaciones pueden escalar sus despliegues sin aumentar linealmente los costos operativos. Al evitar visitas on-site para mantenimiento, las actualizaciones OTA conllevan también una reducción de costos operativos, especialmente apreciable cuando se administra una flota grande de dispositivos distribuidos geográficamente [17]. La automatización centralizada minimiza errores humanos y asegura que todos los dispositivos reciban el mismo paquete de software bajo las mismas condiciones controladas.

2.5.2. Escalabilidad y agilidad en la entrega de software

OTA está diseñado para escalar. En lugar de procesos secuenciales lentos, permite actualizaciones en paralelo a una multitud de dispositivos. Con un buen sistema OTA, una actualización crítica puede propagarse a toda la base instalada en cuestión de horas. Esta capacidad de respuesta ágil es intrínseca al objetivo de CI/CD de entregar valor de forma continua. En el pasado, actualizaciones de firmware podían tardar semanas o meses en completarse; hoy, con OTA, el *time-to-market* de una corrección o nueva funcionalidad en IoT se acorta notablemente, permitiendo sincronizar a los dispositivos con el ritmo de desarrollo de software que proporcionan las prácticas DevOps [13].

2.5.3. Mejora en la seguridad y confiabilidad

En IoT, como en cualquier sistema, aparecen vulnerabilidades de seguridad y errores de software con el tiempo. La diferencia es que, sin OTA, estos fallos podrían permanecer sin corregir indefinidamente en los dispositivos ya desplegados, exponiendo sistemas críticos a riesgos. OTA soluciona esto al habilitar parches de seguridad inmediatos. La capacidad de aplicar parches OTA de forma ágil contribuye a mitigar brechas de seguridad que, de otra manera, podrían comprometer la privacidad o la seguridad de los usuarios [17]. Desde la perspectiva DevOps, esta rapidez de respuesta ante incidentes es vital: se puede integrar la detección de vulnerabilidades mediante la implementación de prácticas DevSecOps con la entrega de parches OTA en el proceso continuo, logrando en última instancia sistemas IoT más seguros y resilientes. Además, mantener todos los dispositivos actualizados reduce la fragmentación de versiones de software en la flota, facilitando la gestión y asegurando que todos cuenten con las últimas mejoras disponibles.

2.5.4. Reducción del *downtime* o tiempo de inactividad

Un beneficio adicional de las técnicas OTA es que permiten aplicar actualizaciones con mínima interrupción del servicio. Las soluciones OTA bien diseñadas emplean estrategias como actualizaciones en segundo plano o intercambio de particiones (A/B) para lograr un despliegue con *downtime* muy reducido [17]. En la práctica, esto significa que un dispositivo IoT puede descargar la nueva versión mientras sigue funcionando con la versión actual e intercambiar a la nueva versión en cuestión de segundos, evitando afectar su disponibilidad. Para sistemas IoT críticos como sensores industriales o equipos médicos que no pueden simplemente apagarse por mucho tiempo, esta capacidad de actualización sin interrupciones prolongadas es fundamental. Integrar OTA en CI/CD garantiza que la velocidad de entrega de nuevas versiones no venga a costa de indisponibilidad en campo: se puede tener iteración continua de software y alta disponibilidad simultáneamente. En plataformas OTA robustas, características como instalaciones atómicas y *rollbacks* sencillos de hacer aseguran que, ante cualquier problema durante la actualización, el dispositivo pueda volver a la versión previa y seguir operando, evitando tiempos muertos extensos [17].

2.5.5. Mantenimiento más eficiente y menor costo a largo plazo

Es evidente que OTA transforma radicalmente la manera de mantener dispositivos IoT. Las organizaciones pueden administrar remotamente su base instalada de manera proactiva en lugar de reactiva, aplicando mantenimiento preventivo de software. Esto se traduce en un menor costo total de la solución IoT, ya que disminuyen los gastos en soporte y se prolonga la vida útil de los dispositivos, al poder solventar problemas de software que, de no arreglarse, podrían inutilizarlos prematuramente. En el contexto de CI/CD, el hecho de incluir los dispositivos finales en el ciclo continuo cierra el bucle de retroalimentación: cualquier fallo detectado en producción puede ser corregido y enviado mediante OTA rápidamente, disminuyendo también las incidencias de soporte y aumentando la satisfacción de los usuarios finales.

La utilización de OTA como parte de una estrategia DevOps en IoT aporta automatización, escalabilidad, rapidez de respuesta y seguridad al proceso de entrega de software embebido. Esto habilita a las organizaciones a manejar despliegues masivos con la misma agilidad y control que se tiene en entornos de software tradicionales, cumpliendo así con el ideal de integración continua y entrega continua, aun cuando el “entorno de producción” consista en dispositivos distribuidos en el mundo físico.

2.6. Limitaciones y desafíos de OTA en IoT

Si bien OTA es un componente esencial para lograr CI/CD en IoT, no está exento de limitaciones y desafíos técnicos que deben ser considerados.

2.6.1. Problemas de conectividad y entornos inestables

La efectividad de OTA depende de la disponibilidad de la red de comunicación con los dispositivos. En entornos reales, puede haber dispositivos IoT conectados a redes celulares con cobertura intermitente, sensores en ubicaciones con señal débil o nodos que se conectan solo esporádicamente para ahorrar energía. Estas condiciones pueden causar fallos en la transferencia OTA, como cortes de energía o pérdida de conectividad a mitad de la descarga, por lo que se requieren mecanismos robustos de gestión de fallos durante la actualización [17]. Sin tales mecanismos, un dispositivo podría quedarse con un firmware incompleto y volverse inoperativo. Las buenas prácticas indican implementar *watchdogs*, reintentos y verificar la integridad de cada paquete descargado. Sin embargo, incluso con precauciones, la realidad es que OTA no puede garantizar al 100 % la entrega en entornos con conectividad muy pobre, lo que obliga a veces a planificar ventanas de actualización cuando las condiciones de red sean óptimas o a utilizar redes redundantes.

2.6.2. Riesgo de fallos catastróficos durante la actualización

A diferencia de actualizar una aplicación en un servidor, donde un fallo podría revertirse rápidamente, en un dispositivo IoT, un error en el proceso de actualización de firmware puede dejar al equipo inutilizable. Esto es más delicado si el componente afectado es el propio sistema operativo embebido o el *bootloader*. Por ello, los sistemas OTA deben integrar características de *fail-safe*. Una de las técnicas utilizadas es la instalación atómica con doble partición A/B, que permite tener un firmware activo y otro de reserva: el nuevo firmware se instala en la partición inactiva y, sólo tras comprobar su integridad, el dispositivo conmuta a esa nueva partición; si algo falla, puede volver a la anterior [17]. Implementar este tipo de esquemas aumenta la complejidad y puede requerir más recursos en el dispositivo, como espacio de almacenamiento extra. Asimismo, es necesario contar con gestión de *rollback* automático. Según [17], si la actualización no se completa exitosamente, por ejemplo, por múltiples intentos fallidos, el dispositivo debe revertir a la versión previa estable. Muchos *frameworks* OTA, como ESP-IDF, Mender o SWUpdate en Linux embebido, ya ofrecen estas funcionalidades; no obstante, integrarlas correctamente en todos los dispositivos sigue siendo un reto que los DevOps en IoT deben abordar en detalle.

2.6.3. Seguridad y autenticación en el proceso OTA

La posibilidad que brinda OTA de realizar actualizaciones remotas desde Internet es un arma de doble filo si no se maneja con estrictas medidas de seguridad. Un sistema OTA podría ser explotado por atacantes para distribuir firmware malicioso, tomar control del dispositivo o corromperlo. Por ello, un desafío crítico es asegurar el canal OTA y el firmware en sí mismo. Las plataformas OTA modernas garantizan la seguridad mediante comunicaciones cifradas durante la descarga, autenticación mutua entre el dispositivo y el servidor, y verificación criptográfica de la integridad y autenticidad del firmware antes de la instalación [17]. Esto usualmente involucra el uso de certificados digitales, firmas con algoritmos (ej.: ECDSA, Ed25519) y protocolos seguros (como TLS, DTLS, etc.) [17]. El desafío para los equipos DevOps es integrar estos aspectos de seguridad en los procesos continuos de CI/CD, como incluir el firmado digital de cada compilación de firmware como parte del proceso automatizado y gestionar de forma segura las claves privadas utilizadas para ello. Adicionalmente, se debe prever la protección contra ataques como *rollback attacks*, impidiendo que alguien reinstale una versión anterior vulnerable o contra intentos de acceso no autorizado al servidor de actualizaciones. Como se puede apreciar, OTA añade un nuevo escenario de ataque que debe ser controlado, y esto complejiza la implementación en comparación con un proceso continuo CI/CD tradicional.

Las actualizaciones OTA enfrentan retos operativos, riesgos técnicos y consideraciones críticas de seguridad. Superar estos desafíos requiere diseñar la solución OTA con características adicionales. Esto teniendo en cuenta las limitaciones propias de los dispositivos IoT. Si bien supone un esfuerzo extra, atender estas limitaciones es indispensable para que la aplicación de DevOps en IoT se materialice de forma segura y eficiente.

2.7. Brecha de conocimiento y justificación de la investigación

A pesar del avance de las prácticas DevOps y de la necesidad de aplicarlas en IoT, todavía existe una brecha de conocimiento en este campo. Por un lado, faltan estudios y referencias detalladas sobre cómo integrar efectivamente CI/CD en sistemas IoT empleando OTA. La literatura actual reconoce la importancia del tema, pero a menudo de forma general. Por ejemplo, “se ha observado un escaso nivel de documentación sobre cómo los fabricantes deberían diseñar sistemas de actualización OTA de manera estructurada o estandarizada” [18]. Esto significa que los ingenieros de software carecen de guías claras o casos de estudio comunes que muestren paso a paso la implementación de un proceso continuo CI/CD. La

mayor parte de las publicaciones se han enfocado en componentes aislados como seguridad OTA, herramientas específicas, etc., más que en una visión holística del problema.

Se aprecia una ausencia de prácticas estandarizadas para la implementación de DevOps en IoT. Mientras que en el desarrollo web o de aplicaciones empresariales han emergido *frameworks* y prácticas DevOps bien establecidas, en IoT cada organización está reinventando la rueda en cuanto a cómo realizar despliegues continuos. “El enfoque DevOps, [...] al menos en IoT, no ha sido formalizado mediante técnicas, lenguajes y herramientas de especificación y diseño de procesos propios de la ingeniería del software” [4]. No hay aún un consenso de la industria en preguntas básicas como: ¿Cuál es la mejor estrategia de integración continua cuando se involucran dispositivos físicos? ¿Cómo versionar y probar componentes de firmware embebido de forma continua? ¿Qué arquitectura de referencia seguir para desplegar en IoT? Estas lagunas dificultan la adopción consistente de DevOps en proyectos IoT, ya que cada equipo debe improvisar su aproximación y aprender por ensayo y error qué funciona y qué no en cada caso particular.

Como respuesta a este contexto, se apunta a establecer procesos reproducibles que combinen herramientas DevOps con plataformas de gestión de dispositivos IoT, logrando así un flujo de entrega de software automatizado de extremo a extremo. Estudios recientes ya subrayan la necesidad de avanzar en esta dirección: “El enfoque DevOps ofrece herramientas útiles que podrían automatizar el despliegue de aplicaciones IoT, lo cual marca la necesidad de más investigación al respecto” [1]. Este trabajo busca contribuir a cerrar esa brecha, proponiendo formas concretas de integrar prácticas CI/CD en sistemas IoT mediante el uso de OTA como mecanismo de entrega continua, garantizando al mismo tiempo la seguridad y confiabilidad necesarias en entornos IoT.

Capítulo 3

Materiales y métodos

En esta sección se detalla el entorno de hardware y software utilizado, así como la metodología empleada para implementar un flujo DevOps con actualizaciones OTA en sistemas IoT. Se describe la configuración del entorno de pruebas, la arquitectura del sistema y los pasos a seguir para lograr la integración continua y el despliegue continuo del firmware en los dispositivos ESP32, aprovechando las capacidades de OTA para minimizar la intervención manual en los despliegues.

Para guiar el diseño de la solución, se tomó como referencia el proceso continuo de implementación DevOps para sistemas de software IoT propuesto por [11]. La figura 3.1 muestra este ciclo como marco conceptual adoptado en este trabajo.

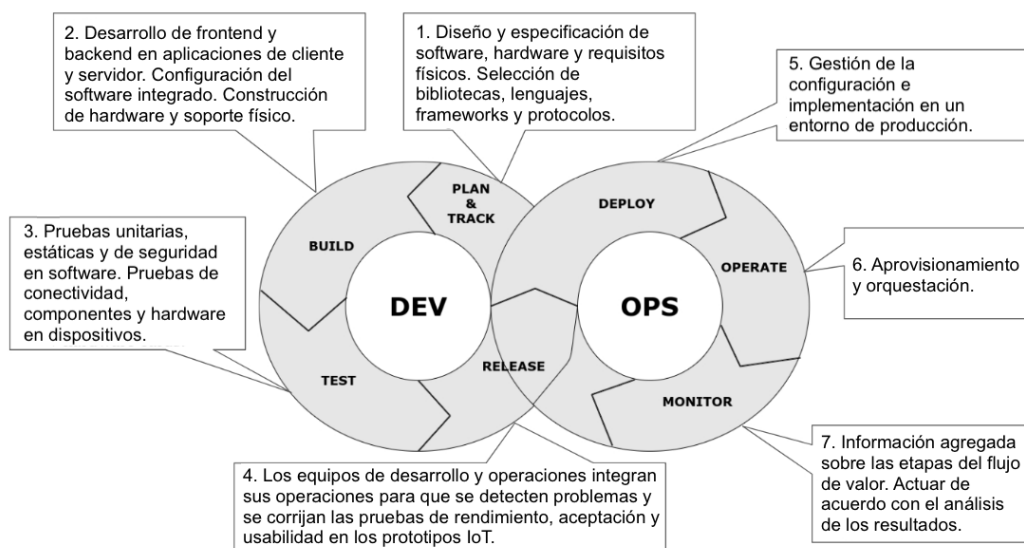


Figura 3.1 Proceso continuo de implementación DevOps propuesto para sistemas de software IoT. (Fuente: [11]).

3.1. Configuración del entorno inicial de pruebas en macOS

Para realizar las pruebas iniciales del entorno de CI/CD se utilizó una computadora con sistema operativo macOS y un procesador Apple M3. Adicionalmente, se instaló el IDE PlatformIO sobre el editor de código Visual Studio Code. El framework ESP-IDF también fue instalado para pruebas locales. La mayor parte del CD se ejecutó en contenedores utilizando Kubernetes locales corriendo en Docker Desktop y runners en Ubuntu para las compilaciones de firmware con GitHub Actions.

3.2. Hardware

El hardware utilizado en el proyecto consiste en los siguientes componentes:

- Raspberry Pi 3 Modelo B+ (1 unidad): Microcomputadora de bajo costo utilizada como gateway de borde. Cuenta con un procesador ARM Cortex-A53 de 4 núcleos y 1 GB de RAM. En este proyecto, la Raspberry Pi se empleó para ejecutar un GitHub Runner, que se encarga de compilar y desplegar el firmware correspondiente a las pruebas unitarias en los ESP32 conectados por USB.
- ESP32 (2 unidades): Microcontroladores Espressif ESP32, cada uno en su respectiva placa de desarrollo. El ESP32 es un SoC de bajo costo y bajo consumo, con WiFi y Bluetooth integrados, con un procesador *dual-core* a 240 MHz, 520 KB de SRAM y 4 MB de memoria flash integrada. Estos dispositivos representan los nodos IoT objetivo donde se despliega el firmware. Se seleccionaron dos ESP32 para realizar pruebas tanto individuales como simultáneas, evaluando la escalabilidad del enfoque.
- Pantallas OLED SSD1306 (2 unidades): Pantallas OLED de 0,96 pulgadas conectadas al ESP32 mediante la interfaz I²C. Aunque no son estrictamente necesarias para el mecanismo OTA, se utilizaron para visualizar en tiempo real los resultados de las actualizaciones y facilitar la depuración visual durante los experimentos.

3.3. Software

Las herramientas de software y plataformas empleadas incluyen:

- ThingsBoard Community Edition v3.8: Plataforma open-source para IoT. Permite monitorear y controlar dispositivos IoT, gestionando telemetría, atributos y actualizaciones OTA. Se instaló en un contenedor corriendo en Docker Desktop y orquestado con

Kubernetes. Esta plataforma se eligió por su soporte nativo a actualizaciones OTA a partir de la versión 3.3, lo que posibilita cargar imágenes de firmware en un repositorio y distribuirlas a dispositivos de forma remota.

- Espressif IoT Development Framework (ESP-IDF) v5.5: Framework oficial de desarrollo para programar el ESP32 en lenguaje C/C++. ESP-IDF proporciona herramientas de compilación y un conjunto de librerías, incluyendo soporte nativo para actualizaciones OTA de firmware.
- PlatformIO v3.3.4: Plataforma open-source orientada al desarrollo de aplicaciones embebidas y sistemas IoT. Integrada con Visual Studio Code, permite la interacción directa con diferentes *frameworks*, incluyendo ESP-IDF, simplificando la gestión de dependencias, la automatización del proceso de compilación y el despliegue del firmware. Asimismo, permite configurar y gestionar fácilmente entornos de desarrollo y producción mediante archivos de configuración específicos (`platformio.ini`). La elección de PlatformIO en este proyecto se fundamentó en su capacidad para automatizar flujos de trabajo complejos y en la posibilidad de ejecutar scripts personalizados durante la compilación y el despliegue, lo que permitió optimizar el proceso de entrega continua del firmware y pruebas unitarias.
- Git: Para gestionar el código fuente se utilizó Git en un repositorio local, integrando así control de versiones en el flujo de trabajo.
- Docker Desktop con k8s v4.38.0: Software de virtualización basado en contenedores que integra soporte nativo para Kubernetes. En este proyecto, Docker Desktop permitió ejecutar localmente un entorno de Kubernetes. Para la configuración del entorno se utilizó una estrategia basada en automatización mediante scripts con Helm y Kubectl.
- GitHub: Plataforma web que ofrece herramientas de control de versiones, automatización y CI/CD. En este proyecto, GitHub fue utilizado para almacenar y gestionar el código fuente del firmware destinado a dispositivos ESP32. También se hizo uso de la funcionalidad GitHub Actions que nos permite, mediante la creación de archivos de configuración YAML, automatizar los procesos de compilación, prueba y despliegue del firmware. Para ejecutar estas tareas, se configuraron runners específicos en la infraestructura local basada en contenedores.
- Vscode: Editor de código integrado con la extensión PlatformIO.

3.4. Protocolos

Los principales protocolos utilizados en este proyecto incluyen:

- MQTT (Message Queue Telemetry Transport): es un protocolo ligero de mensajería basado en el modelo cliente-servidor, diseñado para funcionar sobre TCP y utilizado en entornos IoT. Como señala Brea [16], su arquitectura sigue una topología de estrella, donde el bróker central actúa como intermediario entre los dispositivos que publican datos (productores) y los que los consumen (suscriptores), utilizando una estructura jerárquica de tópicos para organizar la información transmitida. Este modelo desacopla a los emisores de los receptores, permitiendo escalabilidad y flexibilidad. Además, soporta múltiples niveles de calidad de servicio (QoS), mecanismos de autenticación y cifrado SSL/TLS.

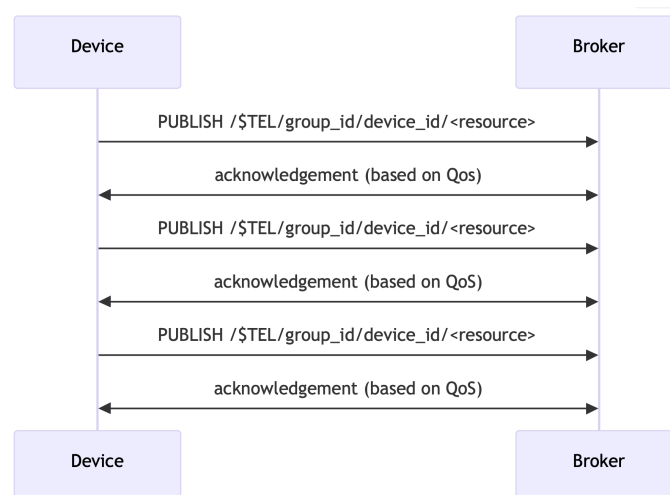


Figura 3.2 Flujo de intercambio de datos MQTT. (Fuente: [16]).

- HTTP (Protocolo de Transferencia de Hipertexto): es un protocolo de comunicación sin estado (stateless), es decir, que no guarda información sobre interacciones entre un cliente y un servidor. Ubicado en la capa de aplicación del modelo OSI. Está diseñado para sistemas de información de hipertexto distribuidos y colaborativos. Es la base de la comunicación de datos en la World Wide Web, permitiendo la transferencia de documentos de hipertexto, como páginas HTML, entre clientes y servidores [16] [19]. Según la RFC 9112 [20], una transacción HTTP está formada por:
 - Línea de inicio (Start-Line): Indica si el mensaje es una solicitud o una respuesta y proporciona información como el método de la solicitud o el código de estado de la respuesta.

- Campos de encabezado (Header Fields): Proporcionan metadatos sobre la solicitud o respuesta, como información sobre el tipo de contenido o detalles de control de caché.
- Línea en blanco (CRLF): Separa los campos de encabezado del cuerpo del mensaje.
- Cuerpo del mensaje (Message Body): Contiene los datos asociados con la solicitud o respuesta, si existen.

Según [16], el uso de campos de encabezado en las transacciones HTTP permite incluir información adicional como autenticación, cifrado e identificación de usuario, lo que otorga gran flexibilidad al protocolo.

3.5. Recursos adicionales

Durante el desarrollo se consultaron y emplearon varios recursos técnicos:

- Documentación oficial de Thingsboard [21]: Se utilizó como referencia durante la implementación.
- Guía OTA de ThingsBoard para ESP32 [22]: Un tutorial oficial de ThingsBoard que explica paso a paso cómo habilitar OTA en un ESP32 usando la plataforma. Esta guía sirvió para orientar la configuración inicial, tanto del lado del servidor como del cliente. Luego se analizaron las capacidades nativas de ThingsBoard v3.8 para OTA.
- Repositorio de ejemplos de ThingsBoard Client SDK en GitHub [23]: Este repositorio contiene ejemplos que muestran cómo utilizar el SDK de cliente de ThingsBoard para conectar dispositivos IoT a la plataforma mediante protocolos MQTT y HTTP.
- Documentación oficial de ESP-IDF [24]: Incluye la guía de *Over The Air Updates (OTA)* de Espressif, que describe el mecanismo de particiones duales y la API para realizar la actualización. Este documento permitió configurar correctamente la tabla de particiones del ESP32 y manejar casos de *rollback*.
- Configuración del entorno en macOS [25]: Se siguieron las instrucciones oficiales de Espressif para instalar ESP-IDF en un entorno macOS.

3.6. Configuración del entorno de pruebas

El entorno de pruebas se diseñó para simular un escenario de actualización remota de firmware en IoT, manteniendo los componentes principales en la red local.

Se realizó la instalación del sistema operativo Ubuntu 22.04 LTS en la Raspberry Pi. Para poder utilizarla como nodo *edge* y correr el runner que nos permite ejecutar las pruebas unitarias en dispositivos físicos, se instaló la aplicación de GitHub Actions. Se asignó la etiqueta específica para identificar el runner (arc-rbpi), se ajustaron variables del entorno y permisos de usuario. Para asegurar la disponibilidad del runner, se configuró el proceso como servicio y se parametrizó para iniciar automáticamente en cada arranque del sistema operativo.

Para que el dispositivo ESP32 pueda recibir actualización vía OTA, es necesario cargar un firmware base vía USB. Este firmware, cuya versión fue la 1.0, incluía ya la lógica de conexión al WiFi, a ThingsBoard y soporte de OTA, además del manejo de la pantalla. La tabla de particiones fue configurada con el esquema adecuado para OTA dual: una partición de almacenamiento no volátil (NVS), dos particiones OTA (ota_0 y ota_1) y una partición de datos OTA (figura 3.3). Esta configuración permite que, en caso de fallos, el dispositivo tenga la posibilidad de volver a una versión estable, como se muestra en la figura 3.4.

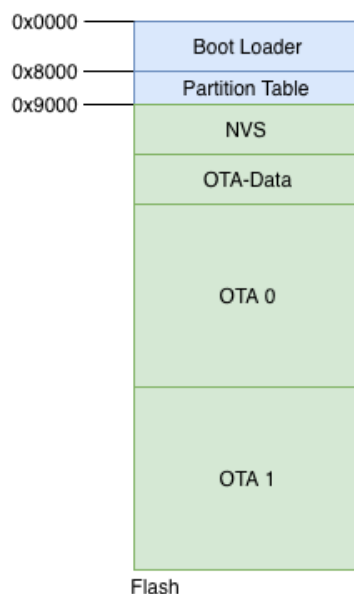


Figura 3.3 Esquema de particiones OTA. (Fuente: [26])

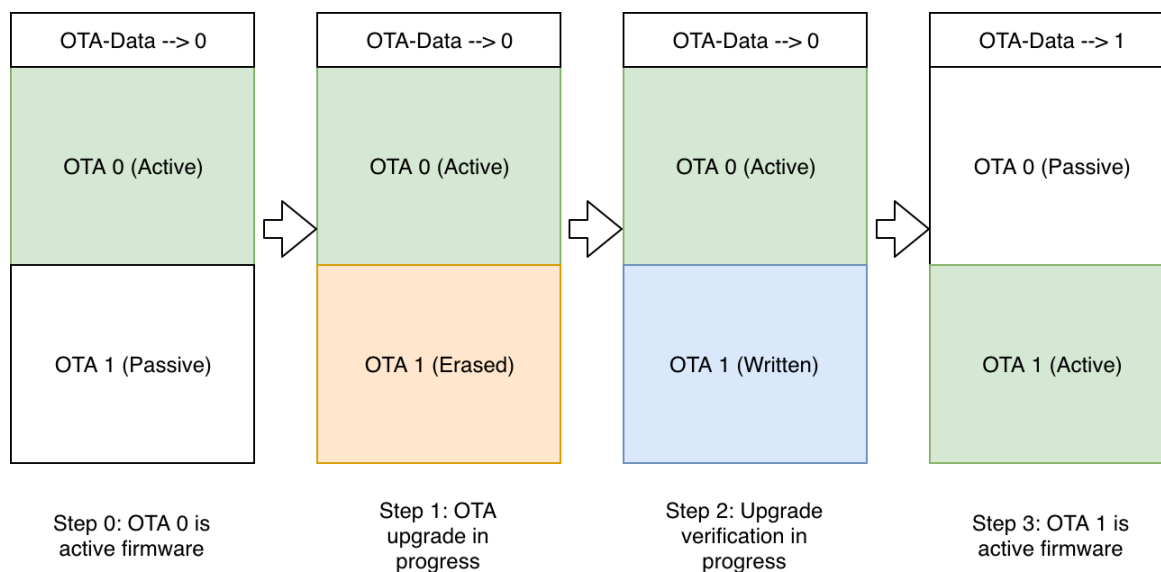


Figura 3.4 Flujo de actualización de firmware. (Fuente: [26])

Se creó en Thingsboard el perfil de dispositivo *ESP32_OTA*, que permite asociar y distribuir paquetes de firmware compatibles.

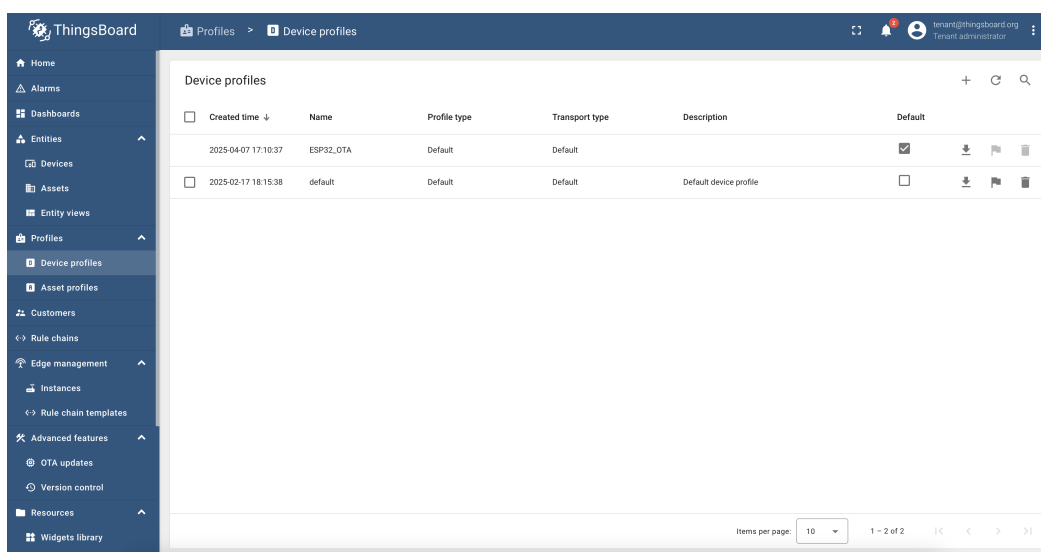


Figura 3.5 Ventana de perfil de dispositivos en ThingsBoard

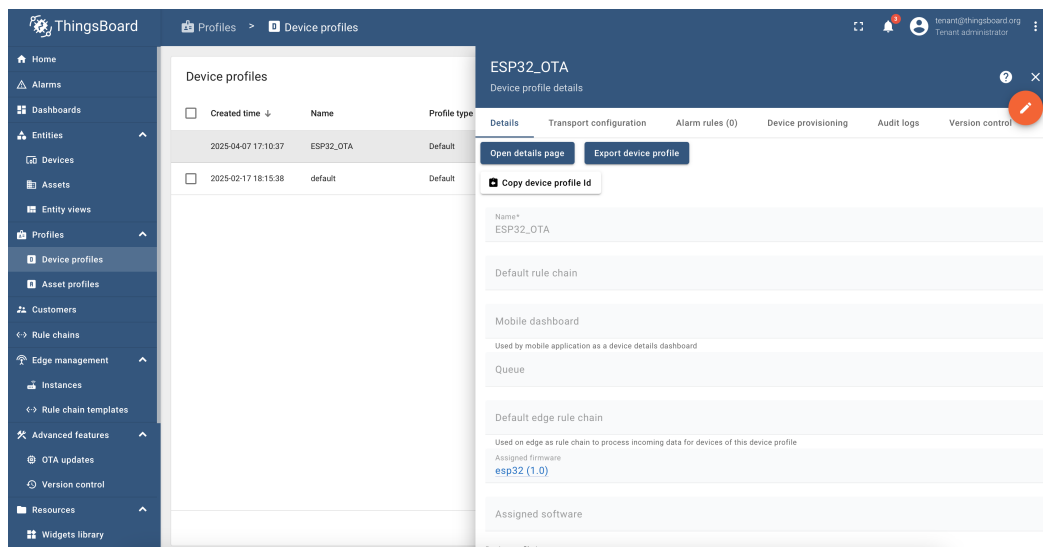


Figura 3.6 Ventana de detalles de perfil de dispositivos en ThingsBoard.

Se crearon los dispositivos *esp32_ota1* y *esp32_ota2* en ThingsBoard y se les asignó el *Device Profile: ESP32_OTA*. Cada dispositivo en ThingsBoard tiene un token de autenticación. Este token se solicita por consola la primera vez que se inicia el dispositivo luego de cargar el firmware base y se almacena en la memoria NVS. De esta forma, el firmware se vuelve genérico y reutilizable en múltiples dispositivos, ya que cada uno de ellos puede conservar su propio token aun después de reinicios o actualizaciones OTA. Aunque en este caso solo se almacena el token, esta técnica se puede aplicar para persistir el resto de las variables.

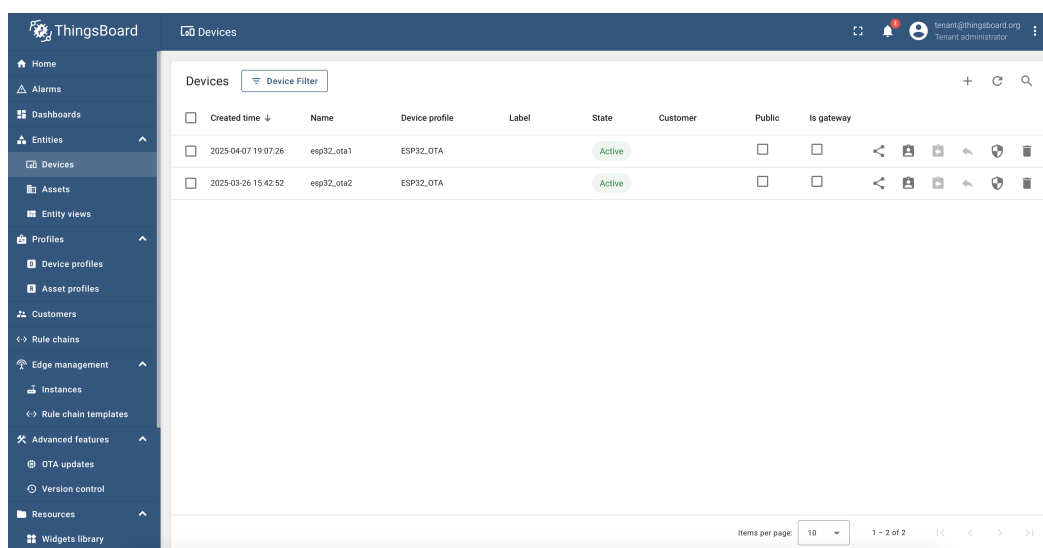


Figura 3.7 Ventana de dispositivos en ThingsBoard.

3.7. Implementación del flujo DevOps con OTA

Crear un flujo DevOps nos permite lograr un ciclo donde los cambios de software en los dispositivos puedan ser integrados, probados y desplegados de forma continua y autónoma, análogo a los entornos de desarrollo tradicionales, pero considerando las particularidades de IoT.

La figura 3.8 muestra la arquitectura general de la infraestructura desplegada para soportar este flujo DevOps.

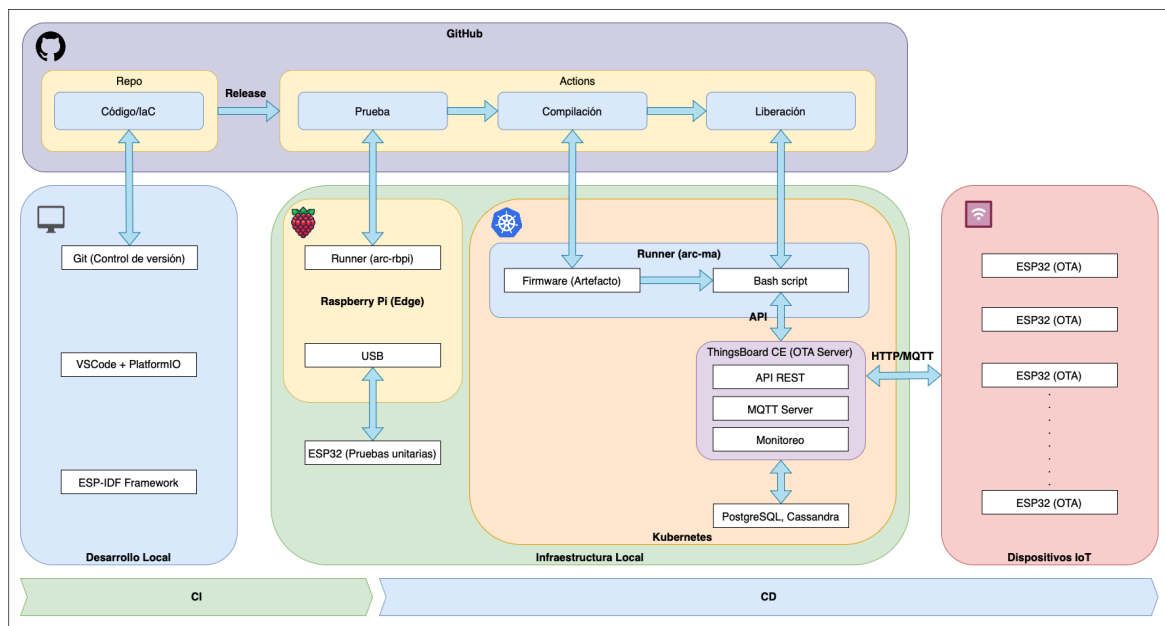


Figura 3.8 Diagrama de la infraestructura.

3.7.1. Control de versiones y desarrollo colaborativo

El código fuente del firmware del ESP32 se mantuvo en un repositorio Git local, y cada cambio que representaba una posible nueva versión de firmware se subía a una cuenta de GitHub. La elección de GitHub como plataforma para control de versiones y ejecución del proceso CI/CD respondió a su madurez, amplia adopción y capacidad para facilitar la colaboración entre los equipos de desarrollo y operaciones, en línea con los principios de DevOps.

Aunque este trabajo prioriza una arquitectura on-premise para mantener el control local y evitar la dependencia de servicios en la nube, el uso de GitHub dentro del proceso fue acotado, limitado exclusivamente a la gestión de versiones y a la orquestación del pipeline. Los procesos sensibles como las pruebas, la compilación y el despliegue se ejecutaron

íntegramente en infraestructura local. Cabe destacar que GitHub se basa en Git y, al ser un estándar abierto, garantiza la portabilidad hacia otras plataformas como GitLab o Bitbucket.

3.7.2. Despliegue de infraestructura local mediante IaC

Se implementó la infraestructura local mediante la técnica de infraestructura como código (*Infrastructure as Code, IaC*), que permite definirla y gestionarla utilizando *scripting*, facilitando la replicabilidad, escalabilidad y alineada con el objetivo de este trabajo de priorizar una solución *on-premise* en lugar de servicios en la nube comerciales.

Mediante esta automatización, se crearon los clústeres y se desplegaron los contenedores necesarios: ThingsBoard Community Edition, PostgreSQL y Cassandra. Componentes de infraestructura como Ingress Controller (Nginx) y Cert-Manager para gestión de certificados. Además, se utilizó Actions Runner Controller (ARC) para desplegar runners locales desde GitHub Actions.

3.7.3. Automatización del proceso CI/CD de firmware IoT con GitHub Actions

El proceso de CI/CD se realizó utilizando un flujo de trabajo de GitHub Actions con runners locales que permitieron automatizar las tareas de prueba, compilación y despliegue del firmware IoT.

1. Cada vez que se crea un nuevo *release* de software, se ejecuta un flujo de trabajo en GitHub Actions llamado “*Build, Test, and Release OTA Firmware*”.
2. Se ejecutan las pruebas unitarias del código en un runner local que corre en una Raspberry Pi (arc-rbpi). Esta tarea incluye la instalación de las dependencias necesarias como Python y PlatformIO.
3. Se configuran los permisos para acceder al dispositivo ESP32 conectado al puerto */dev/ttyACM0* de la Raspberry Pi. A continuación, se cargan y ejecutan las pruebas en el hardware real. Si todas las pruebas son superadas, el flujo continúa; en caso contrario, se detiene en este punto.
4. Mediante la acción *checkout*, se obtiene el código fuente y se transfiere a una máquina virtual que corre sobre el runner local (arc-ma). Se instala Python 3 y PlatformIO.
5. Con el entorno listo, se ejecuta PlatformIO para compilar el firmware. En este paso se utiliza la variable *ref_name* para establecer la versión del firmware. PlatformIO ejecuta

- internamente ESP-IDF con la configuración del proyecto (*boards, frameworks, etc.*), generando el archivo binario correspondiente.
6. Si la compilación es exitosa, el binario generado se almacena como artefacto del flujo de trabajo mediante la acción *upload-artifact*.
 7. El artefacto es recuperado usando la acción *download-artifact*.
 8. Mediante un script en Bash, se realiza una llamada HTTP al endpoint de autenticación de ThingsBoard para obtener un token de acceso a la API REST.
 9. Se calcula el *checksum* SHA-256 del binario y, junto con el nombre, versión, descripción y perfil de dispositivo, se crea y registra un nuevo paquete OTA en ThingsBoard por medio de una solicitud a la API.
 10. Se realiza una segunda llamada a la API para cargar el archivo binario del firmware en el servidor de ThingsBoard.
 11. El paquete OTA creado se asigna al *Device Profile* correspondiente, completando el ciclo de integración continua y entrega continua del firmware.

La figura 3.9 resume el proceso CI/CD.

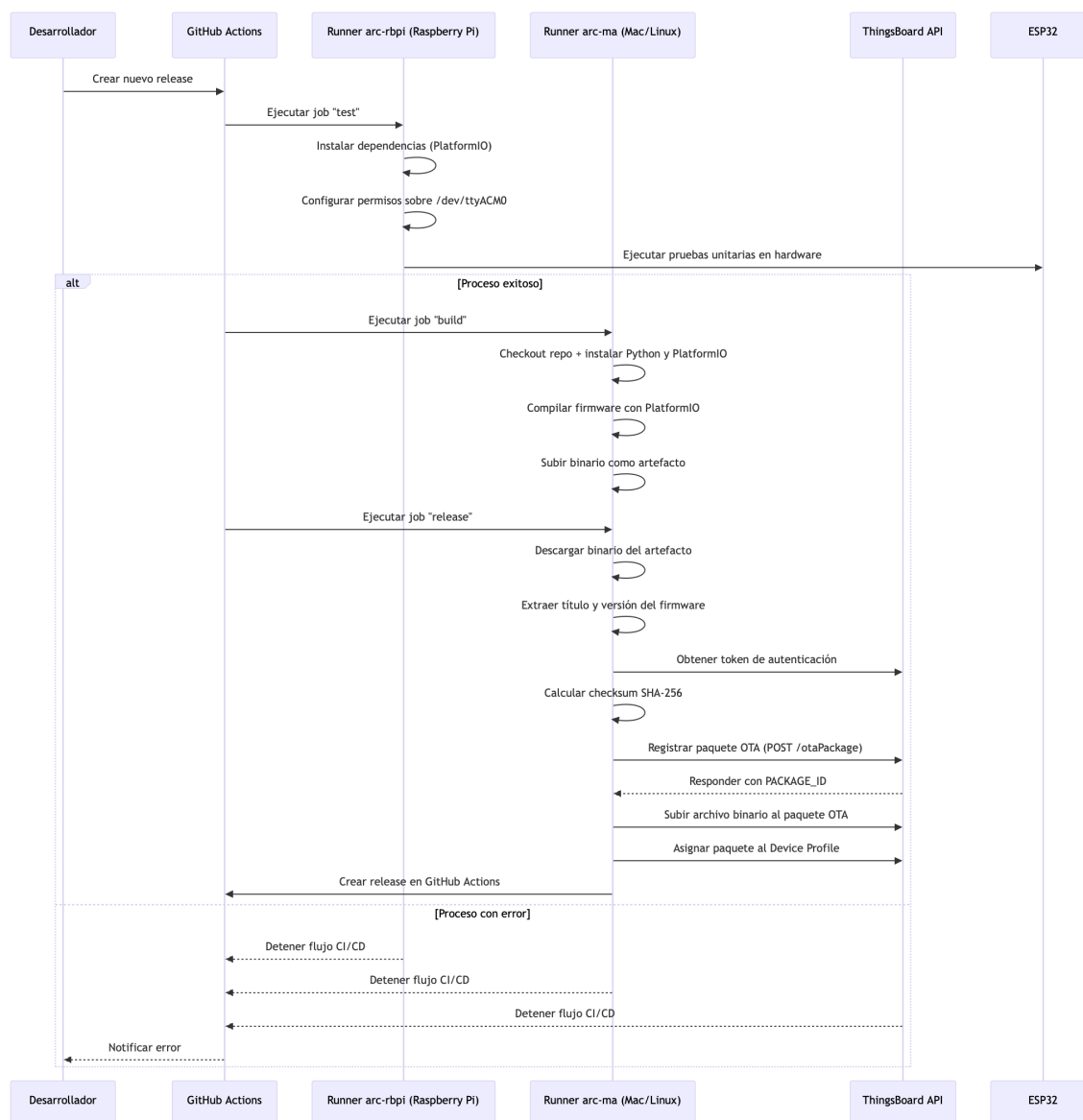


Figura 3.9 Diagrama de secuencia del proceso CI/CD de firmware IoT con GitHub Actions.

3.7.4. Despliegue continuo de firmware mediante OTA: del repositorio al dispositivo

El firmware publicado en ThingsBoard cierra el ciclo de CI/CD llevando la actualización hasta el dispositivo final mediante OTA.

Con la nueva versión de firmware disponible, se envía un mensaje MQTT al tópico de atributos del dispositivo. El ESP32 suscrito a ese tópico recibe la notificación de una actualización disponible.

El dispositivo compara la versión recibida con la actualmente instalada. Si son distintas, inicia el proceso de descarga OTA.

El dispositivo reporta el estado de cada etapa del proceso de actualización (*DOWNLOADING*, *DOWNLOADED*, *UPDATE*, *UPDATED*, *FAILED*) a ThingsBoard, que los registra como telemetría.

Una vez finalizada la descarga, el firmware del ESP32 verifica la integridad del archivo. Si la verificación es exitosa, procede a instalar el nuevo firmware en la partición OTA designada.

El dispositivo reinicia para aplicar la actualización. Al iniciar, envía un mensaje de telemetría indicando la versión actual y el estado *UPDATED*, confirmando la correcta aplicación del firmware.

Si ocurre un error en cualquier etapa, el dispositivo reporta el estado *FAILED* junto con un mensaje descriptivo; esto permite identificar dispositivos que no lograron actualizarse correctamente.

El proceso descrito se sintetiza en la figura 3.10.

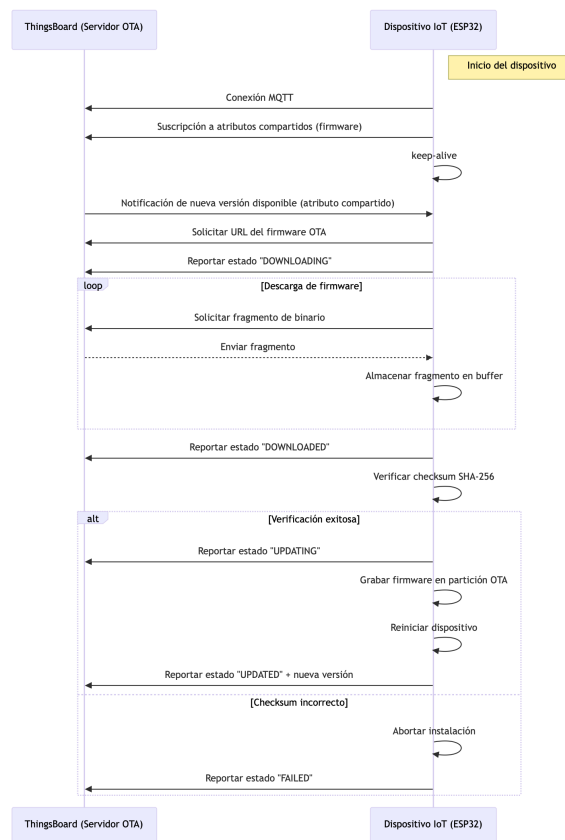


Figura 3.10 Diagrama de secuencia de actualización mediante OTA.

3.7.5. Seguridad y confiabilidad en el proceso continuo

Aunque la implementación de prácticas DevSecOps y sistemas de monitoreo queda fuera del alcance de este estudio, se integraron mecanismos básicos de seguridad y supervisión para garantizar la integridad, autenticidad y confiabilidad del proceso.

Se incorporó el uso de canales cifrados y autenticados en todas las interacciones entre componentes. Por ejemplo, el agente en Kubernetes interactúa con GitHub y ThingsBoard utilizando HTTPS con autenticación mediante tokens o credenciales, asegurando que solo entidades autorizadas puedan publicar nuevas versiones de firmware. ThingsBoard requiere credenciales administrativas o un token JWT válido para aceptar la carga de nuevos paquetes OTA vía API, lo que previene accesos no autorizados.

Durante el transporte del firmware al dispositivo, es posible habilitar MQTT seguro, lo que garantiza cifrado de extremo a extremo. El dispositivo valida el certificado del servidor antes de aceptar la descarga. Además, cada paquete de firmware incluye un *checksum* SHA-256 que el dispositivo puede comparar tras la descarga; si la verificación de integridad falla, el proceso de actualización se aborta. El propio dispositivo se autentica ante ThingsBoard utilizando un token que lo identifica de forma inequívoca.

La utilización de runners locales aporta también beneficios en términos de seguridad. Evita exponer servicios mediante NAT y abrir puertos o publicar direcciones IP en servicios DNS dinámicos. El flujo funciona de manera inversa: cuando se activa una acción en GitHub Actions, el runner local toma la tarea de la cola y la ejecuta internamente, sin requerir conexiones entrantes desde Internet. Como plus, esto mantiene los artefactos y operaciones críticas dentro del perímetro seguro de la red privada.

En cuanto a la confiabilidad, el proceso incorpora mecanismos de monitoreo y recuperación ante fallos. ThingsBoard registra el estado de cada actualización OTA como atributos del dispositivo, lo que permite supervisar en tiempo real el progreso de cada despliegue. Esta visibilidad facilita la detección proactiva de fallos, como dispositivos que no completaron la actualización.

Además, la arquitectura del proceso continuo desacoplado mejora la resiliencia: si una etapa falla, ya sea una prueba, una compilación o un error al subir el firmware, las etapas siguientes no se ejecutan hasta que se resuelva el problema, evitando la propagación de versiones defectuosas.

En las pruebas realizadas, el sistema demostró ser capaz de entregar nuevas versiones de firmware de forma consistente y segura, reduciendo significativamente los tiempos de entrega en comparación con procesos manuales. Este enfoque confirma que la implementación de prácticas DevOps, específicamente CI/CD complementado con OTA, contribuye tanto a la seguridad como a la confiabilidad del proceso de actualización en entornos IoT.

Finalmente, la información obtenida a través del monitoreo continuo no solo permite detectar fallos en tiempo real, sino que también constituye una fuente de retroalimentación útil para los equipos de desarrollo. Esta retroalimentación posibilita analizar el comportamiento del firmware desplegado, identificar patrones de error recurrentes y mejorar versiones futuras, cerrando así el ciclo DevOps incluso en entornos IoT con limitaciones de acceso y conectividad.

Capítulo 4

Experimentos y resultados

Luego del análisis realizado, se plantea un experimento para evaluar el desempeño y la fiabilidad de la implementación propuesta. El objetivo es comparar un proceso que implementa prácticas DevOps contra métodos tradicionales de despliegue de firmware en IoT.

Se buscó medir cinco aspectos que, a mi entender, son fundamentales durante el proceso de actualización:

- Tiempo de actualización: OTA vs. manual.
- Tasa de éxito de las actualizaciones OTA.
- Consumo de recursos durante el proceso OTA.
- Actualizaciones masivas con OTA.
- Prueba integral del proceso CI/CD con OTA.

4.1. Tiempo de actualización: OTA vs. manual

Se midió el tiempo requerido para actualizar el firmware del ESP32 utilizando OTA a través de ThingsBoard, comparándolo con el método tradicional de actualización manual mediante conexión física. Para ello, se definió un caso de prueba que consiste en pasar de la versión 1.0 a la 1.1 del firmware en un dispositivo:

4.1.1. Actualización OTA

El proceso comienza cuando se asigna la nueva versión en ThingsBoard y termina cuando el ESP32 completa el reinicio y se registra con la nueva versión. En las pruebas, el tamaño

del firmware era de 802,9 KB. El dispositivo tardó en promedio 58,10 segundos en completar la actualización OTA. Donde tardó 7,2 segundos para iniciar el proceso, 46,45 segundos correspondientes a la descarga del binario, 2,3 segundos a la verificación e instalación y 2,15 segundos al reinicio y reconexión a la plataforma.

Durante la descarga OTA, el dispositivo continuó ejecutando sus tareas de aplicación, aunque con ligeras demoras, lo cual demuestra que la actualización no paraliza completamente la funcionalidad del dispositivo hasta el momento del reinicio.

4.1.2. Actualización manual

Consiste en desconectar el dispositivo de su ubicación de operación, conectarlo físicamente vía USB y utilizar la herramienta de *flashing* (PlatformIO/Upload) para grabar el nuevo firmware. En un entorno controlado, este método tomó en promedio 62,97 segundos por dispositivo, donde 15,02 segundos fueron de preparativos, 17,71 segundos de borrado y carga, 30,24 segundos de reinicio y verificación manual. Si bien el tiempo de transmisión de datos es 2,6 veces más rápido comparado contra OTA, la sobrecarga manual y el hecho de requerir atención individual en cada dispositivo añaden demoras significativas. Además, en casos con muchos dispositivos, este proceso debe repetirse secuencialmente para cada unidad, lo que escala linealmente el tiempo total de despliegue.

4.1.3. Análisis de resultados

Los resultados evidencian que la actualización OTA es mucho más rápida y escalable en términos prácticos. En escenarios donde los dispositivos IoT estén remotos o difíciles de alcanzar, el método tradicional no es viable en tiempo real, mientras que OTA permite que las actualizaciones ocurran sin importar la ubicación física del dispositivo. Este beneficio coincide con lo reportado en la literatura, donde se señala que la programación OTA es indispensable para actualizar dispositivos remotos sin conexión física, evitando largas interrupciones en su servicio.

4.2. Tasa de éxito de las actualizaciones OTA

La fiabilidad es un aspecto crítico al actualizar dispositivos de forma remota. Para evaluar la tasa de éxito de las actualizaciones OTA, se llevó a cabo una serie de iteraciones de actualización y se registró cuántas veces la actualización concluía exitosamente. En total, se realizaron 12 actualizaciones OTA. Los resultados fueron los siguientes:

4.2.1. Actualizaciones exitosas

En diez casos, los dispositivos recibieron el nuevo firmware, lo instalaron y reportaron el estado UPDATED a ThingsBoard, indicando éxito. La plataforma confirmó cada caso, marcando la versión nueva como activa para el dispositivo y almacenando el historial de actualización correctamente.

4.2.2. Actualizaciones fallidas

Se generaron fallos de forma simulada, modificando el parámetro *FIRMWARE_PACKET_SIZE* a un valor no soportado por el dispositivo. En estos casos, el dispositivo detectó el fallo y revirtió al firmware anterior, reportando estado FAILED junto con un mensaje de error enviado como telemetría a ThingsBoard.

4.2.3. Análisis de resultados

Podemos calcular la tasa de éxito observada, excluyendo la falla intencional, como del 100% en las pruebas realizadas. Esto sugiere que, en condiciones de red estables y con dispositivos dentro del alcance de WiFi, el mecanismo OTA implementado es confiable.

Comparativamente, en métodos manuales, la tasa de éxito suele depender de factores humanos, por ejemplo, errores al actualizar el firmware equivocado, desconexiones accidentales, etc., pero asumiendo un procedimiento cuidadoso, también puede ser cercana al 100%. Sin embargo, un punto importante es que OTA reduce la probabilidad de error, ya que automatiza el proceso. Las comprobaciones automáticas añaden una capa de seguridad que en actualizaciones manuales podría omitirse. Además, al estar integrado en una plataforma, es posible monitorear en tiempo real el estado de cientos de dispositivos durante la actualización, detectando inmediatamente si alguno falla. Esto contrasta con el proceso manual donde podría no ser evidente un fallo hasta intentar usar el dispositivo.

Estos hallazgos están alineados con estudios previos que destacan la necesidad de robustez en las actualizaciones OTA para IoT, incluyendo verificaciones post-actualización y capacidad de revertirlas para garantizar que los nodos sigan operativos aún si ocurre un error.

4.3. Consumo de recursos durante el proceso OTA

Una preocupación común al ejecutar OTA en dispositivos IoT es el impacto en los recursos limitados que estos tienen, como la CPU y la memoria. Los ESP32 entran en esta lista, por lo que es importante que el proceso de actualización no comprometa la funcionalidad principal

del dispositivo ni exceda sus capacidades. Para evaluar esto, se generó un firmware que nos permite medir de forma aproximada el consumo de CPU y la memoria RAM utilizada.

4.3.1. Consumo de CPU

Dado que el ESP32 es *dual-core*, se observó la carga en ambos núcleos antes, durante y después de una actualización OTA. En estado inactivo, solo enviando telemetría cada cierto tiempo, la CPU0 llegó al 1,00 % y la CPU1 no superó el 6,00 % de uso. Durante la descarga e instalación OTA, la CPU0 tuvo un pico de 46,63 % al inicio de la descarga para luego pasar al 3,50 % en promedio, mientras que en la CPU1 la carga subió momentáneamente hasta el 70 %, manteniéndose aproximadamente a un 40 % en promedio hasta finalizar el proceso de actualización. Este incremento era esperable, ya que el dispositivo estaba realizando simultáneamente operaciones de red y de escritura en memoria *flash*, además de las tareas base. Sin embargo, FreeRTOS ajustó las prioridades de tareas de modo que la tarea de OTA obtuviera la mayoría del tiempo de CPU sin congelar por completo otras actividades. Podemos observar que, aunque con pequeñas demoras, el dispositivo siguió respondiendo a otras tareas durante la actualización. Tras completar la actualización OTA y reiniciar, la carga volvió a los niveles normales (para más detalles de cada caso en particular, ver las figuras A.1 a A.10 del Apéndice A).

Se realizó además una prueba específica simulando una falla durante la actualización OTA, generando *Internet connection lost*. En este caso particular, como se observa en la figura A.11, el dispositivo comenzó la actualización y rápidamente presentó una carga en el uso del CPU, observándose un incremento significativo, especialmente en la CPU1, que alcanzó picos del 100 %, mientras que la CPU0 alcanzó picos del 50 %. Cabe destacar que, a pesar de los picos, el dispositivo nunca dejó de enviar telemetría a Thingsboard.

4.3.2. Consumo de memoria RAM

El ESP32 tiene 520 KB de SRAM; parte de esta memoria se asigna para manejar buffers para la descarga OTA, además de la memoria de la aplicación. Mediante las funciones de ESP-IDF `esp_get_free_heap_size()`, se comprobó la memoria ocupada antes y durante la actualización OTA. Antes de la actualización, el dispositivo tenía aproximadamente 200 KB ocupados. Al iniciar la actualización OTA, se observó un incremento de aproximadamente 45 KB, correspondiente a los buffers de la tarea de actualización OTA. Este uso adicional de RAM es relativamente bajo y se mantuvo constante durante la transferencia. Tras la finalización, esa memoria fue liberada. En ningún momento se produjo un *overflow* ni un

out-of-memory, lo que indica que la configuración de buffers de OTA fue correcta (para más detalles de cada caso en particular, ver las figuras A.12 a A.21 del Apéndice A).

Se realizó además una prueba específica simulando una falla durante la actualización OTA, generando un escenario de *Internet connection lost*. En este caso particular, como se evidencia en la figura A.22, el consumo de memoria permaneció prácticamente constante en torno a los 245 KB, similar a un caso normal sin interrupciones.

4.3.3. Análisis de resultados

En general, el consumo de recursos durante el proceso OTA fue adecuado. El ESP32 logró ejecutar la operación sin alcanzar el 100 % de utilización, tanto en CPU como en memoria. Los resultados obtenidos son satisfactorios, considerando que uno de los principales riesgos al implementar prácticas DevOps en IoT es que las herramientas de automatización representen una carga excesiva para dispositivos con recursos limitados. En este caso, el proceso se mantuvo dentro de márgenes operativos seguros. Para contextos industriales con hardware aún más restringido, sería necesario evaluar su viabilidad; sin embargo, muchos dispositivos IoT modernos cuentan con los recursos suficientes para beneficiarse de estas prácticas.

4.4. Actualizaciones masivas con OTA

Se evaluó la capacidad del sistema para manejar actualizaciones OTA en varios dispositivos de forma concurrente. Enviando una actualización al perfil de dispositivo de ThingsBoard con dos dispositivos asociados.

4.4.1. Evaluación experimental

En las pruebas realizadas, ambos dispositivos comenzaron la descarga de firmware de forma simultánea; se registró una diferencia de 0,5 segundos, atribuible al orden interno de la cola de notificaciones. En el dashboard de ThingsBoard, se pudo observar que los dos dispositivos pasaron por los estados `DOWNLOADING` → `DOWNLOADED` → `UPDATE` → `UPDATED` en paralelo.

El tiempo de actualización en los dos dispositivos fue similar al caso individual: el primer dispositivo tardó 58,15 segundos y el segundo dispositivo tardó 59,22 segundos.

Se repitió la prueba nueve veces más, obteniendo tiempos similares en todos los casos.

La tasa de éxito fue del 100 %. En ningún caso se experimentó pérdida de paquetes ni necesidad de reintentos.

4.4.2. Análisis de resultados

Los experimentos realizados demuestran que la arquitectura propuesta es escalable; por su diseño, es capaz de soportar muchos más dispositivos. La principal limitación se encuentra en el hardware del servidor y el ancho de banda de red. Para mitigar esta restricción, la plataforma ThingsBoard ofrece mecanismos que permiten distribuir las actualizaciones a lo largo del tiempo, evitando así picos de carga. En entornos empresariales, este sistema puede integrarse con soluciones más robustas, como un servidor en la nube o un servidor de borde (*edge server*), el cual opera en la red local de los dispositivos y actúa como intermediario o *gateway* hacia la infraestructura en la nube. Este enfoque permite coordinar actualizaciones OTA para miles de dispositivos utilizando el mismo software. Por lo tanto, desde la perspectiva de DevOps aplicado a IoT con soporte de OTA, resulta factible orquestar despliegues masivos con mínima intervención, manteniendo la consistencia y el control centralizado.

4.5. Prueba integral del proceso CI/CD con OTA

Se llevó a cabo una prueba integral del proceso CI/CD que comenzó con la creación de un *release* en GitHub (CI). Una vez publicado este nuevo *release*, se ejecutó automáticamente el proceso continuo definido en GitHub Actions (CD). Para más detalles sobre el flujo de actualización OTA, véase la figura 3.9 y la figura 3.10.

4.5.1. Pruebas unitarias

Las pruebas se ejecutaron en una Raspberry Pi conectada físicamente a un ESP32 y concluyeron exitosamente en todos los casos, validando la funcionalidad básica del firmware antes de realizar la compilación definitiva, con un tiempo promedio de ejecución de aproximadamente 13 minutos y 30 segundos.

Este tiempo, si bien es aceptable para un entorno de prueba, podría reducirse significativamente utilizando hardware más potente, como una Raspberry Pi 5 o un servidor dedicado.

4.5.2. Compilación

El proceso de compilación se realizó utilizando PlatformIO ejecutándose en un runner local. Este paso generó satisfactoriamente el artefacto del firmware y lo almacenó para su

posterior uso en el proceso de despliegue. El tiempo promedio de compilación fue de 3 minutos y 24 segundos.

4.5.3. Despliegue

El proceso de despliegue recuperó el artefacto generado anteriormente y lo subió al repositorio OTA de ThingsBoard mediante el uso de la API REST. El tiempo total en concluir esta tarea fue de aproximadamente 41 segundos.

4.5.4. Distribución

ThingsBoard notificó al dispositivo ESP32 a través de MQTT sobre la disponibilidad del nuevo firmware. Tras recibir la notificación, el dispositivo descargó, verificó, instaló y reinició con el nuevo firmware. Se confirmó que el dispositivo reportó correctamente el estado UPDATED a la plataforma, completando este paso en un promedio de 58 segundos.

4.5.5. Análisis de resultados

El proceso completo, desde la publicación del *release* en GitHub hasta la confirmación exitosa del nuevo firmware en el dispositivo, tomó en promedio 18 minutos y 33 segundos, a lo largo de las cinco pruebas realizadas. Cabe aclarar que, si es necesario desplegar el nuevo firmware en cientos de dispositivos, el tiempo de actualización no crece necesariamente de forma lineal. Por el contrario, la distribución del firmware puede realizarse prácticamente en simultáneo, incrementando el tiempo total en unos pocos segundos.

4.6. Análisis comparativo y conclusiones de los resultados

Los experimentos realizados en este capítulo nos permiten comparar de forma objetiva el enfoque tradicional vs. el enfoque DevOps. En la tabla 4.1 se presenta un resumen de las diferencias halladas entre ambos enfoques.

Tabla 4.1 Desempeño operativo: OTA vs. actualización manual

Métrica	OTA	Manual
Tiempo (1 disp.)	~58 s	~63 s + manipulación
Downtime	~3 s	Todo el proceso
Tasa de éxito	~100 % (rollback automático)	~100 % (depende del técnico)
Datos transferidos	~800 KB (Wi-Fi)	~800 KB (USB)
Concurrencia	Cientos de disp.	1 por técnico
Soporte de rollback	Sí (automático)	No (manual)
Presencia física	No	Sí

En la tabla 4.2 se observa el consumo adicional de CPU y memoria RAM asociado al proceso de actualización OTA. Estos valores corresponden al pico de utilización registrado durante la descarga y escritura del firmware en el dispositivo. Cabe aclarar que no es posible realizar una medición equivalente en el proceso de actualización manual, ya que el dispositivo permanece desconectado del sistema durante dicho procedimiento.

Tabla 4.2 Consumo de recursos durante la actualización OTA

Métrica	Valor (OTA)
Pico de CPU	70 % en CPU1, 45 % en CPU0 (5–6 s)
RAM adicional	~45 KB (buffers OTA)

4.6.1. Eficiencia de despliegue

El método DevOps con OTA superó al enfoque tradicional en términos de rapidez. La capacidad de actualizar dispositivos remotamente en segundos habilita ciclos de entrega continua y despliegue continuo en IoT que antes eran inviables. Esto acorta significativamente el tiempo de puesta en producción de mejoras o correcciones, alineándose con la práctica DevOps de entregas frecuentes y rápidas. Además, la posibilidad de realizar actualizaciones concurrentes elimina cuellos de botella que no serían posibles con un enfoque manual.

4.6.2. Confiabilidad y reducción de riesgo

A pesar de automatizarse, el proceso resultó ser confiable, con mecanismos integrados para asegurar que una actualización defectuosa no deje inutilizable al dispositivo, gracias a las pruebas unitarias implementadas y a *rollbacks* automáticos. Los resultados muestran que con OTA se puede mantener la estabilidad incluso aumentando la frecuencia de despliegues,

ya que el riesgo de fallo es bajo. La tasa de éxito del 100 % en las pruebas es un indicador de que las actualizaciones automáticas pueden ser tan o más confiables que las manuales, eliminando el factor humano y permitiendo un enfoque proactivo en lugar de reactivo.

4.6.3. Impacto en recursos y operatividad

Contra la posible creencia de que incorporar DevOps/CI-CD pueda representar una carga significativa para dispositivos IoT, las mediciones de CPU y memoria muestran un impacto moderado. Los dispositivos pueden seguir realizando su función principal durante las actualizaciones y no requieren hardware significativamente más potente para soportar el mecanismo de actualización OTA. Esto significa que la sobrecarga de introducir DevOps en IoT es baja en comparación con los beneficios obtenidos.

4.6.4. Escalabilidad y gestión centralizada

El experimento de actualizaciones masivas demuestra que el enfoque es escalable. Para una organización que maneje una flota grande de dispositivos IoT, un proceso continuo DevOps centralizado permitiría actualizar todos sus dispositivos de forma coordinada y consistente, manteniendo versiones homogéneas y facilitando el seguimiento mediante *dashboards* unificados. Esto contrasta con escenarios tradicionales donde cada dispositivo podría terminar con versiones distintas por actualizaciones manuales inconsistentes. La plataforma IoT ThingsBoard, actuando como intermediario, es clave, ya que provee herramientas de gestión de dispositivos a gran escala, algo que se relaciona con las prácticas DevOps de tener visibilidad y control de todo el ciclo de vida del software.

Capítulo 5

Conclusiones y futuras líneas de investigación

5.1. Beneficios

Según los hallazgos de esta investigación, la aplicación de prácticas DevOps en sistemas IoT ofrece numerosos beneficios, varios de ellos análogos a los obtenidos en el desarrollo de software tradicional y otros particulares al dominio de IoT. En primer lugar, la automatización de la integración continua y el despliegue continuo permite agilizar la entrega de nuevas funcionalidades y correcciones en dispositivos distribuidos. Al eliminar tareas manuales en el ciclo de vida como la compilación, pruebas e implementación, se reducen significativamente los tiempos para llevar un cambio desde el código fuente hasta el dispositivo en producción. Esto es crucial para mejorar el *time-to-market* y la entrega de parches de seguridad, traducándose en dispositivos que permanecen actualizados, seguros y funcionales por más tiempo. Un ejemplo ilustrativo es la capacidad de reaccionar rápidamente ante fallos: donde antes un error crítico en el firmware de un dispositivo podía requerir días de intervención manual, con un proceso continuo DevOps es posible desarrollar, probar y distribuir un parche en cuestión de horas, minimizando el tiempo de inactividad del sistema. Esta mayor agilidad en la entrega de software otorga una ventaja competitiva y una capacidad de respuesta mucho más rápida frente a incidentes o cambios en los requisitos.

Al adoptar una estrategia de implementación continua, cada modificación de firmware puede someterse a un conjunto de pruebas unitarias, de integración y regresivas antes de ser desplegada. Esto ayuda a detectar defectos de forma temprana, evitando propagar errores a dispositivos productivos. Asimismo, las prácticas DevOps fomentan la implementación de telemetría y monitoreo en producción: una vez desplegado el firmware, los dispositivos

IoT reportan métricas de desempeño, errores y eventos que son recopilados y analizados por el equipo. Este bucle de retroalimentación continua proporciona información valiosa para entender el comportamiento en producción y realizar mejoras. Con ello, se logra incrementar la estabilidad de los dispositivos a largo plazo, reduciendo la frecuencia de fallos y optimizando el rendimiento mediante ajustes sucesivos basados en datos reales.

DevOps fomenta la colaboración entre los equipos de desarrollo y operaciones, estableciendo una cultura organizacional centrada en la comunicación eficaz y la retroalimentación continua. Esto incrementa la calidad del software IoT, ya que los problemas pueden detectarse y resolverse más rápidamente, fomentando que el software evolucione de forma proactiva en lugar de reactiva. La coordinación fluida entre áreas permite responder con agilidad a cambios inesperados, como atender vulnerabilidades de seguridad críticas o adaptar el firmware a nuevas tecnologías de hardware. Asimismo, la adopción de prácticas como pruebas automatizadas y monitoreo continuo en los dispositivos proporciona un bucle de mejora constante: cada iteración de firmware puede validarse exhaustivamente antes del despliegue y supervisarse en tiempo real una vez en producción, asegurando dispositivos más estables y con menos fallos a largo plazo.

Por último, los resultados de las pruebas en este estudio resaltan la optimización de la distribución de firmware y la independencia de la infraestructura como beneficios adicionales. La arquitectura propuesta, basada en una plataforma local de IoT integrada al proceso continuo CI/CD, ofrece una alternativa económica y flexible frente a soluciones comerciales en la nube. Esta aproximación local con OTA otorga a la organización mayor control sobre sus dispositivos y datos, minimizando la dependencia de proveedores externos y permitiendo personalizar las herramientas según necesidades específicas. En conjunto, la implementación de DevOps en IoT mejora la robustez, seguridad y sostenibilidad de la infraestructura de software, asegurando que la obsolescencia de los dispositivos no esté determinada por la falta de actualizaciones, sino que puedan evolucionar continuamente junto con la demanda del entorno.

5.2. Consideraciones

A pesar de las ventajas descritas anteriormente, la implementación de DevOps en IoT supone diversos desafíos y limitaciones que emergen tanto en el plano técnico como en el organizativo. Según se identificó durante la investigación, en el plano técnico, uno de los obstáculos más evidentes es la heterogeneidad de dispositivos y sus restricciones de recursos. A diferencia de un entorno de software convencional donde la mayoría son servidores

homogéneos, el ecosistema IoT se caracteriza por la diversidad y la gran cantidad de hardware existente.

Las limitaciones de memoria, almacenamiento y procesamiento de muchos dispositivos implican que el proceso de actualización debe ser más eficiente. Almacenar e instalar un nuevo firmware puede verse afectado si el dispositivo tiene poca RAM o memoria *flash* disponible. Asimismo, la memoria *flash* de estos dispositivos sufre un proceso de desgaste con escrituras repetidas, lo que puede degradar su funcionamiento a largo plazo. En el caso del ESP32, que fue el microcontrolador utilizado en las pruebas, la vida útil típica de la memoria *flash* es de aproximadamente 10.000.000 (diez millones) de operaciones de lectura/escritura, según la documentación oficial de Espressif [27]. Este límite impone una restricción práctica a la frecuencia de despliegues OTA, especialmente si no se implementan mecanismos como la nivelación de desgaste (*wear leveling*) o la rotación de particiones, que ayudan a distribuir las escrituras de manera más uniforme sobre la memoria. Considerando estas limitaciones, se evaluó el uso de un esquema de doble partición (A/B) que supone la reducción de la memoria *flash* efectiva como aceptable, dado que existen versiones del ESP32 con hasta 16MB de capacidad. Además, la posibilidad de realizar actualizaciones seguras, confiables y automáticas mediante OTA constituye un pilar fundamental para cerrar el ciclo CI/CD en entornos IoT. La fiabilidad, escalabilidad y capacidad de *rollback* ofrecidas por este mecanismo justifican su implementación, incluso ante la penalización de espacio disponible.

El consumo de energía es un aspecto importante a considerar, ya que muchos dispositivos funcionan con baterías, de modo que aplicar actualizaciones frecuentes podría agotar su energía prematuramente. El proceso OTA debe entonces optimizarse para gastar lo mínimo posible, implementando métodos que permitan la transmisión de firmware utilizando diferencias incrementales en lugar de imágenes completas o programando las actualizaciones en horarios de bajo consumo.

La heterogeneidad del hardware constituye una barrera para la estandarización del proceso continuo y complica la gestión de configuraciones y despliegues de forma uniforme. Esto obliga a que las herramientas DevOps sean livianas y altamente optimizadas; por ejemplo, instrumentar una extensa telemetría o ejecutar contenedores completos puede ser inviable en nodos de bajos recursos. Del mismo modo, garantizar la conectividad y entrega de software a dispositivos que pueden estar geográficamente dispersos o con conexión inestable es un desafío: el proceso continuo debe considerar reconexiones, colas de entrega y adaptarse a posibles pérdidas de enlace. Todo esto añade complejidad al sistema. Sin embargo, existen abordajes para mitigar esto, como el uso de protocolos de actualización OTA estandarizados y diseñados para entornos restringidos, o la incorporación de *gateways* de borde (*edge*)

que actúan como puente entre la infraestructura DevOps central y los dispositivos finales, traduciendo y distribuyendo las actualizaciones de manera compatible con cada tipo de nodo.

El tratamiento de los requerimientos no funcionales es un punto crítico a tener en cuenta. En sistemas IoT, mejorar un atributo a veces implica penalizaciones en otro. Encontrar el equilibrio óptimo entre rendimiento, seguridad, eficiencia energética y disponibilidad constituye un desafío permanente durante la adopción de DevOps en IoT y a menudo no existe una solución universal: cada caso de uso debe analizarse para parametrizar adecuadamente el proceso continuo a sus restricciones particulares.

La implementación de un flujo DevOps para IoT implica construir un proceso continuo CI/CD que integre todos los pasos y permita coordinar diversas herramientas y tecnologías. Es necesario unir repositorios de código, sistemas de integración continua capaces de compilar firmware o paquetes para diferentes arquitecturas, bancos de pruebas simulados o de hardware real y, finalmente, mecanismos de despliegue hacia los dispositivos. Conseguir que todos estos componentes trabajen en forma cohesionada no es trivial: durante la investigación se debieron superar desafíos de compatibilidad entre entornos, como adaptar GitHub Actions para manejar compilaciones de firmware o integrar Kubernetes localmente para orquestar servicios de actualización. Este esfuerzo inicial de configuración del entorno DevOps puede ser elevado, requiriendo tiempo y conocimientos especializados. Muchas organizaciones pueden carecer de infraestructuras sobre las cuales montar estas herramientas, teniendo que invertir en crear una plataforma prácticamente desde cero. Incluso cuando se logra implementar técnicamente, mantener la baja sobrecarga en los dispositivos es un punto delicado: incorporar procesos adicionales consume recursos, por lo que deben ser muy eficientes.

El dinamismo característico de aplicar prácticas DevOps plantea el problema de mantener documentación y control de cambios al día. Cada despliegue en IoT puede implicar actualizaciones en versiones de firmware, en configuraciones distribuidas, en topologías de red, etc. Mantener registros y manuales actualizados con ese ritmo resulta muy difícil si se hace de forma manual y omitirlo podría llevar a confusión o pérdida de conocimiento tácito dentro del equipo. Como indican Rong et al. [28], están surgiendo propuestas de documentación continua (DevDocOps) a la par de CI/CD, donde la documentación se genera automáticamente en cada cambio, pero es un área aún inmadura. La seguridad del conocimiento es tan importante como la del software: si parte del equipo se marcha o rota y no hay documentación fidedigna de qué se desplegó, cómo se configuraron los dispositivos o qué versión exacta está corriendo en cada nodo, la operación podría complicarse seriamente.

Por último, al ser DevOps para IoT un paradigma relativamente nuevo, no existen aún estándares universalmente aceptados ni un corpus consolidado de buenas prácticas. Las

publicaciones y casos de estudio están dispersos en la literatura y muchas organizaciones están explorando soluciones de forma aislada. Esta falta de un “camino marcado” implica que los nuevos adoptantes deban experimentar por sí mismos, con el riesgo de incurrir en errores ya vividos por otros por falta de difusión. También se evidencia una inmadurez en las herramientas especializadas: aunque abundan plataformas DevOps para aplicaciones web o móviles, las equivalentes adaptadas a IoT recién comienzan a aparecer. Esto se traduce en una curva de aprendizaje más pronunciada y en la necesidad, muchas veces, de personalizar o armar como un rompecabezas la infraestructura DevOps-IoT. Superar esta etapa de baja estandarización es un desafío comunitario: requerirá compartir más estudios de caso, desarrollar *frameworks* comunes y quizá crear foros especializados donde converjan las experiencias en DevOps para IoT.

5.3. Reflexión final

En conclusión, los resultados obtenidos muestran que integrar DevOps en el ecosistema IoT aporta mejoras sustanciales en la forma de desarrollar y operar estos sistemas. Se confirmó que principios bien establecidos en el desarrollo de software tradicional son aplicables al dominio de IoT y aportan beneficios análogos: mayor rapidez de entrega, mejor calidad del producto y una gestión más eficiente y centralizada de infraestructuras complejas. Este trabajo es importante porque acerca el mundo del IoT al estado del arte en prácticas de ingeniería de software, cerrando la brecha entre el desarrollo tradicional y las prácticas modernas de ciclo de vida continuo. Al implementar y probar un proceso continuo DevOps adaptado a IoT, se evidenció que en la práctica es posible actualizar dispositivos en producción de forma tan ágil y confiable como se actualiza una aplicación tradicional. Esto representa un cambio de paradigma: de procesos manuales y esporádicos, se transita a una práctica sistemática y repetible donde las mejoras llegan a los dispositivos con alta cadencia y de forma predecible. Consecuentemente, un dispositivo IoT deja de ser un elemento estático cuya versión de software queda obsoleta, para convertirse en un ente dinámico que evoluciona continuamente junto con las necesidades del negocio o del entorno. También es importante destacar que el enfoque adoptado, al priorizar una solución local y desacoplada de proveedores de nubes, demuestra que la independencia tecnológica en sistemas IoT es viable sin comprometer la funcionalidad. Esto otorga autonomía sobre los datos y flexibilidad para entornos que, por políticas de seguridad o falta de conectividad robusta, requieren operar al margen de grandes nubes públicas. En síntesis, el trabajo realizado no sólo valida la hipótesis de que DevOps mejora la integración y entrega de software en IoT, sino que proporciona una base práctica para que otros proyectos IoT adopten estas prácticas con mayor confianza. Se espera que

estos hallazgos contribuyan a que la comunidad IoT en general avance hacia procesos de desarrollo más ágiles, seguros y eficientes, aprovechando al máximo las ventajas de DevOps.

En cuanto a futuras líneas de investigación, este estudio deja abiertos varios frentes para profundizar y refinar la integración de DevOps en IoT. Uno de ellos es ampliar el alcance de las pruebas automatizadas: si bien se implementaron pruebas unitarias al código, sería interesante incorporar ensayos más completos, como pruebas de carga masiva o pruebas de resiliencia ante fallos de red, integradas automáticamente en el proceso continuo. Por otro lado, optimizar el mecanismo de actualizaciones es un objetivo que puede llevarse más lejos: la solución propuesta demostró la eficacia de OTA tradicional, pero futuras mejoras podrían incluir actualizaciones delta para minimizar el tráfico, o el uso de redes *peer-to-peer* entre dispositivos cercanos, con el fin de distribuir las cargas de actualización de forma más escalable. Ligado a esto, incorporar técnicas de orquestación inteligente que determinen automáticamente el momento y la estrategia de despliegue; por ejemplo, escalar actualizaciones según horarios de bajo uso o saltarse dispositivos con conexión inestable hasta que mejore la señal. Esto aumentaría la robustez del proceso en escenarios reales.

Otra línea de investigación recomendada es profundizar en el monitoreo continuo y el análisis de datos IoT dentro del ciclo DevOps. Aún persiste la necesidad de cerrar completamente el círculo de retroalimentación: se podrían desarrollar herramientas que recopilen datos de cientos de dispositivos, detecten patrones o anomalías mediante algoritmos de análisis en tiempo real y retroalimenten al equipo de desarrollo con información procesable. En esta línea, resulta pertinente mencionar el concepto emergente de AIOps (Artificial Intelligence for IT Operations). Dang et al. [29] señalan que el término fue acuñado por Gartner para abordar los desafíos operativos de DevOps mediante el uso de inteligencia artificial (IA). AIOps implica aplicar técnicas de IA y aprendizaje automático para automatizar tareas de operación, detectar eventos relevantes, anticipar fallos y facilitar la toma de decisiones en tiempo real. Por último, pero no menos importante, la seguridad integral mediante prácticas DevSecOps en IoT representa un aspecto esencial que también requiere atención. Futuros trabajos podrían centrarse en reforzar la seguridad en cada etapa del proceso continuo, incorporando mecanismos de confianza que garanticen que las actualizaciones no solo sean rápidas, sino también íntegras y fiables.

Este trabajo no solo valida la hipótesis de que DevOps mejora la integración, el despliegue y la entrega de software en IoT, sino que ofrece una base práctica para su adopción en proyectos futuros. Se espera que estos hallazgos impulsen a la comunidad a avanzar hacia procesos de desarrollo más ágiles, seguros y eficientes, aprovechando el potencial de DevOps y favoreciendo que el Internet de las Cosas alcance niveles superiores de madurez en su ciclo de vida de desarrollo y operación de software.

Referencias

- [1] Georges Bou Ghantous y Asif Qumer Gill. «DevOps Reference Architecture for Multi-cloud IOT Applications». En: *2018 IEEE 20th Conference on Business Informatics (CBI)*. Vol. 01. 2018, págs. 158-167. DOI: 10.1109/CBI.2018.00026.
- [2] Gartner. *OTA (Over-the-Air)*. Recuperado el 6 de diciembre de 2024. s.f. URL: <https://www.gartner.com/en/information-technology/glossary/ota-over-the-air>.
- [3] Patrick Debois. «Agile Infrastructure and Operations: How Infra-gile are You?» En: *Agile 2008 Conference*. 2008, págs. 202-207. DOI: 10.1109/Agile.2008.42.
- [4] Miguel Angel López Peña. «DevOps Aplicado a Sistemas IoT: Definición e Implementación de Procesos Continuos de Monitorización y Retroalimentación». No Publicado. Jul. de 2021. DOI: 10.20868/UPM.thesis.68043.
- [5] Kevin Ashton. «That 'Internet of Things' Thing». En: *RFID Journal* (2009). Recuperado el 1 de agosto de 2024. URL: <https://www.rfidjournal.com/expert-views/that-internet-of-things-thing/73881/>.
- [6] International Telecommunication Union. *Recommendation ITU-T Y.2060: Overview of the Internet of Things*. Inf. téc. Recuperado el 13 de agosto de 2024. 2012. URL: <https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060>.
- [7] Microsoft, Intel, and IoT Analytics. *IoT Signals - Manufacturing Spotlight*. Inf. téc. Recuperado el 20 de abril de 2024. 2022. URL: <https://info.microsoft.com/ww-landing-IoT-signals-manufacturing-spotlight.html>.
- [8] Katie Costello. «Gartner predice el futuro de las infraestructuras de Cloud y Edge Computing». En: *Gartner* (2021). Recuperado el 22 de agosto de 2024. URL: <https://www.gartner.es/es/articulos/gartner-predice-el-futuro-de-las-infraestructuras-de-cloud-y-edge-computing>.
- [9] Barbara Kitchenham y col. «Systematic literature reviews in software engineering – A systematic literature review». En: *Information and Software Technology* 51.1 (2009). Special Section - Most Cited Articles in 2002 and Regular Research Papers, págs. 7-15. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2008.09.009>.
- [10] Claes Wohlin. «Guidelines for snowballing in systematic literature studies and a replication in software engineering». En: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268.

- [11] Igor Muzetti Pereira, Tiago Garcia de Senna Carneiro y Eduardo Figueiredo. «Understanding the context of IoT software systems in DevOps». En: *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*. 2021, págs. 13-20. DOI: 10.1109/SERP4IoT52556.2021.00009.
- [12] Leonardo Leite y col. «A Survey of DevOps Concepts and Challenges». En: *ACM Comput. Surv.* 52.6 (nov. de 2019). ISSN: 0360-0300. DOI: 10.1145/3359981.
- [13] Ligia Georgeta Gușeală, Dragoș-Vasile Bratu y Sorin-Aurel Moraru. «DevOps Transformation for Multi-Cloud IoT Applications». En: *2019 International Conference on Sensing and Instrumentation in IoT Era (ISSI)*. 2019, págs. 1-6. DOI: 10.1109/ISSI47111.2019.9043730.
- [14] Elizabeth Suescún-Monsalve y col. «DevOps in Industry 4.0: A Systematic Mapping». En: *Revista Facultad de Ingeniería* 30.57 (jul. de 2021), e13314. DOI: 10.19053/01211129.v30.n57.2021.13314.
- [15] Muhammad Alam y col. «Orchestration of Microservices for IoT Using Docker and Edge Computing». En: *IEEE Communications Magazine* 56.9 (2018), págs. 118-123. DOI: 10.1109/MCOM.2018.1701233.
- [16] R. C. J. Brea. «HoT-Hub Of Things: Software integrador homogéneo y seguro para el Internet de las cosas». Repositorio de la Universidad Argentina John F. Kennedy. Tesis de grado. Universidad Argentina John F. Kennedy, 2017.
- [17] Konstantinos Arakadakis y col. «Firmware Over-the-air Programming Techniques for IoT Networks - A Survey». En: *ACM Comput. Surv.* 54.9 (oct. de 2021). ISSN: 0360-0300. DOI: 10.1145/3472292.
- [18] Mónica M. Villegas, Cristian Orellana y Hernán Astudillo. «A study of over-the-air (OTA) update systems for CPS and IoT operating systems». En: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. ECSA '19. Paris, France: Association for Computing Machinery, 2019, págs. 269-272. ISBN: 9781450371421. DOI: 10.1145/3344948.3344972.
- [19] R. Fielding, M. Nottingham y J. Reschke. *RFC 9110: HTTP Semantics*. 2022. DOI: 10.17487/RFC9110.
- [20] R. Fielding, M. Nottingham y J. Reschke. *RFC 9112: HTTP/1.1*. 2022. DOI: 10.17487/RFC9112.
- [21] ThingsBoard. *OTA updates*. Recuperado el 16 de enero de 2025. s.f. URL: <https://thingsboard.io/docs/user-guide/ota-updates>.
- [22] ThingsBoard. *ESP32 OTA updates sample*. Recuperado el 16 de enero de 2025. s.f. URL: <https://thingsboard.io/docs/samples/esp32/ota/>.
- [23] ThingsBoard. *ThingsBoard Client SDK Examples*. Recuperado el 3 de marzo de 2025. s.f. URL: <https://github.com/thingsboard/thingsboard-client-sdk/tree/master/examples>.
- [24] Espressif Systems. *OTA — Over-the-Air Updates*. Recuperado el 16 de enero de 2025. s.f. URL: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ota.html>.
- [25] Espressif Systems. *ESP-IDF Getting Started Guide: Linux and macOS Setup*. Recuperado el 3 de marzo de 2025. s.f. URL: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/linux-macos-setup.html>.

- [26] Espressif Systems. *Firmware Upgrades*. Recuperado el 16 de marzo de 2025. s.f. URL: <https://docs.espressif.com/projects/esp-jumpstart/en/latest/firmwareupgrade.html>.
- [27] Espressif Systems. *nvs rw*. Recuperado el 21 de marzo de 2025. s.f. URL: <https://docs.espressif.com/projects/esp-faq/en/latest/software-framework/storage/nvs.html>.
- [28] Guoping Rong y col. «DevDocOps: Enabling continuous documentation in alignment with DevOps». En: *Software: Practice and Experience* 50.3 (2020), págs. 210-226. DOI: 10.1002/spe.2770.
- [29] Yingnong Dang, Qingwei Lin y Peng Huang. «AIOps: Real-World Challenges and Research Innovations». En: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, págs. 4-5. DOI: 10.1109/ICSE-Companion.2019.00023.

Apéndice A

Gráficas de consumo de recursos durante actualizaciones OTA

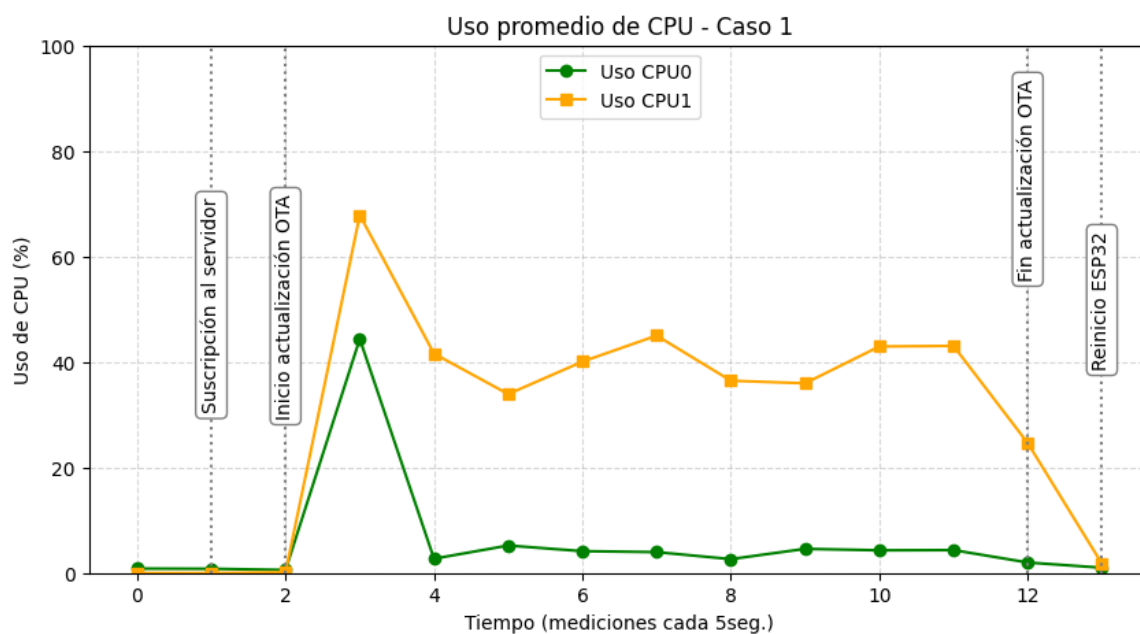


Figura A.1 Uso promedio de CPU - Caso 1

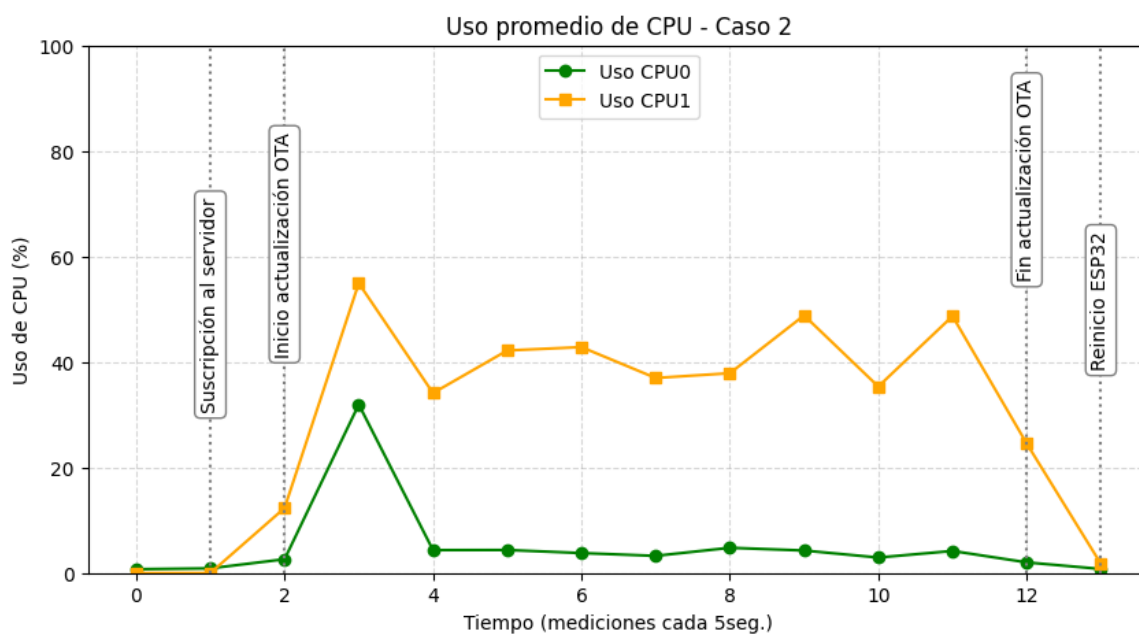


Figura A.2 Uso promedio de CPU - Caso 2

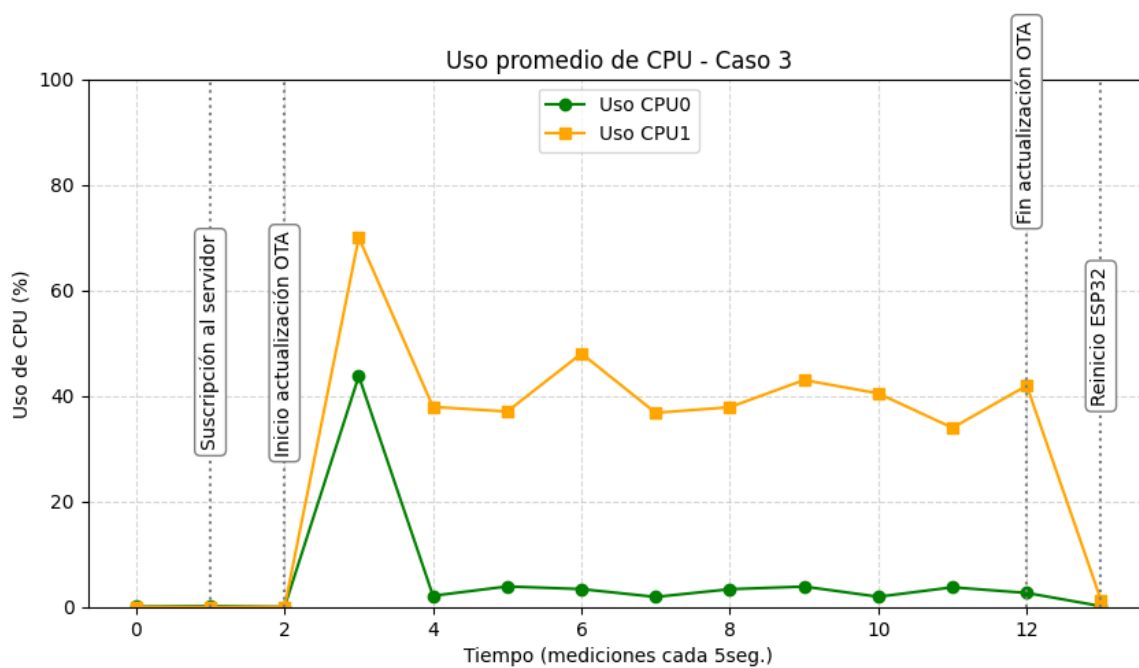


Figura A.3 Uso promedio de CPU - Caso 3

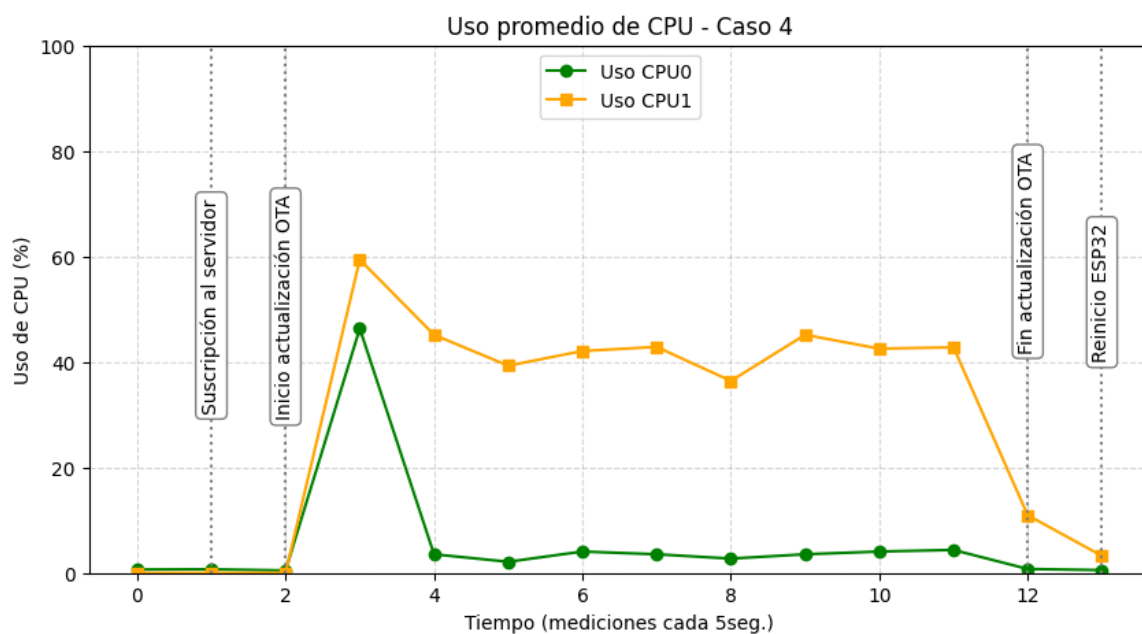


Figura A.4 Uso promedio de CPU - Caso 4

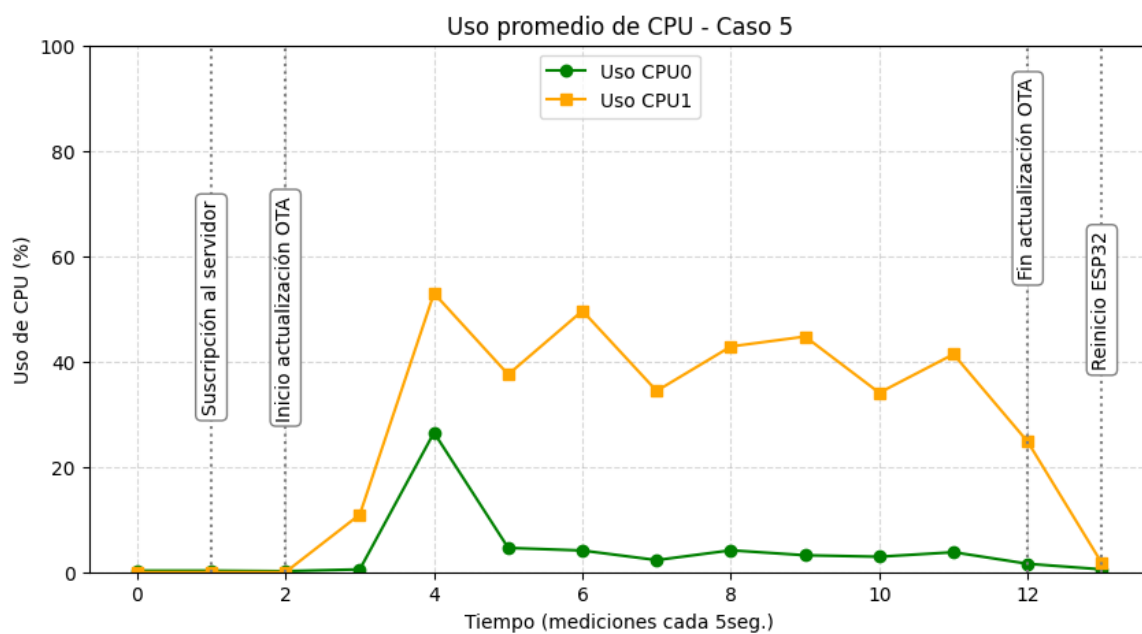


Figura A.5 Uso promedio de CPU - Caso 5

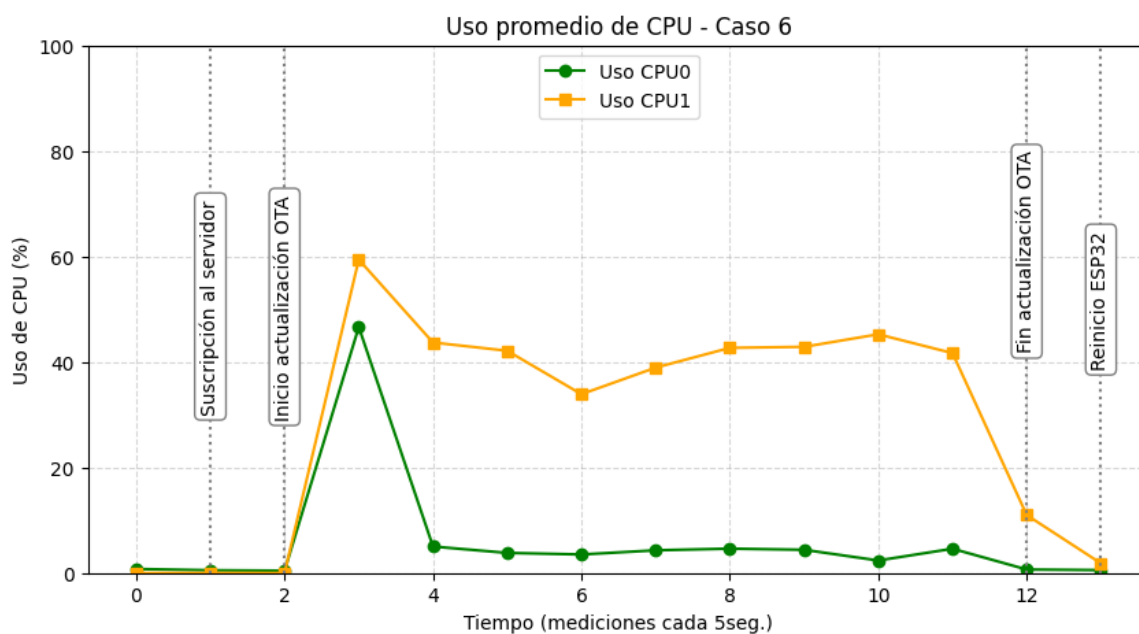


Figura A.6 Uso promedio de CPU - Caso 6

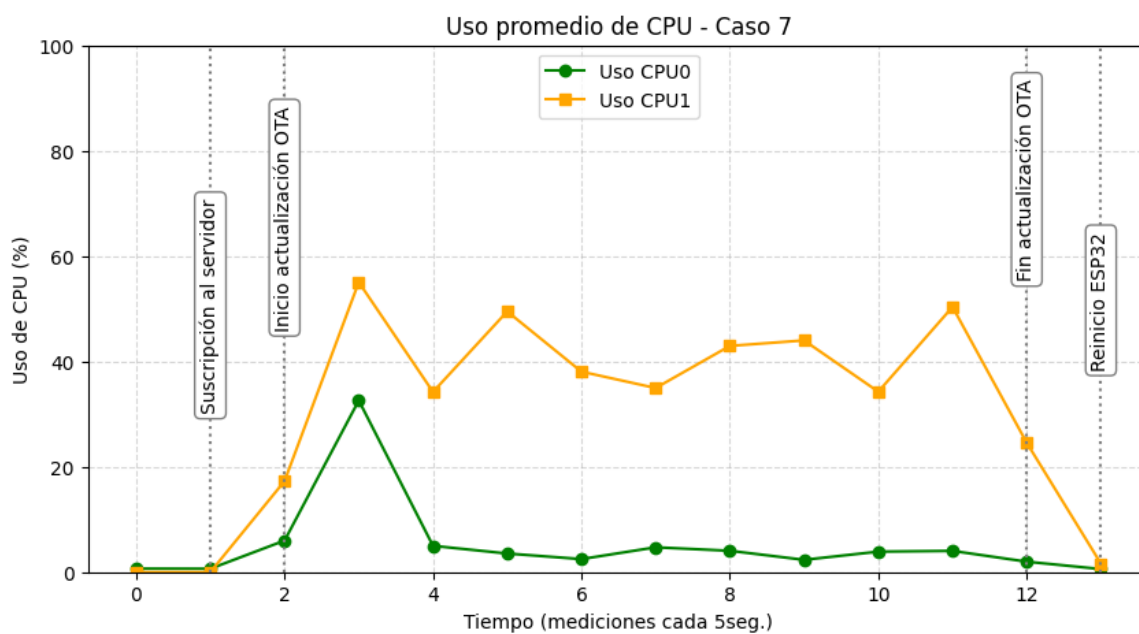


Figura A.7 Uso promedio de CPU - Caso 7

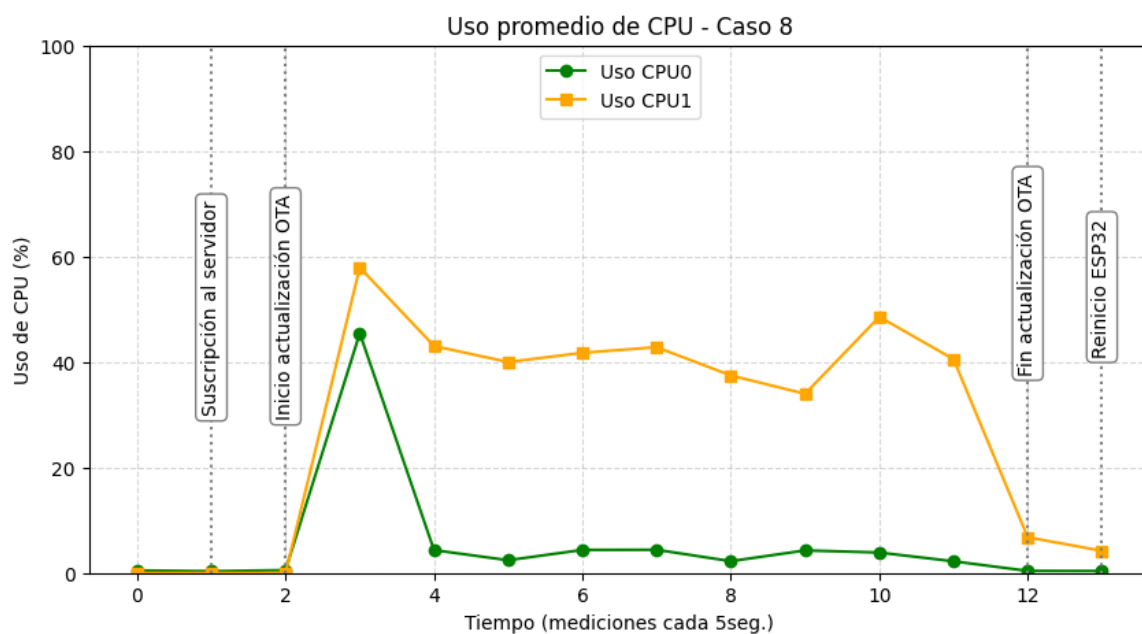


Figura A.8 Uso promedio de CPU - Caso 8

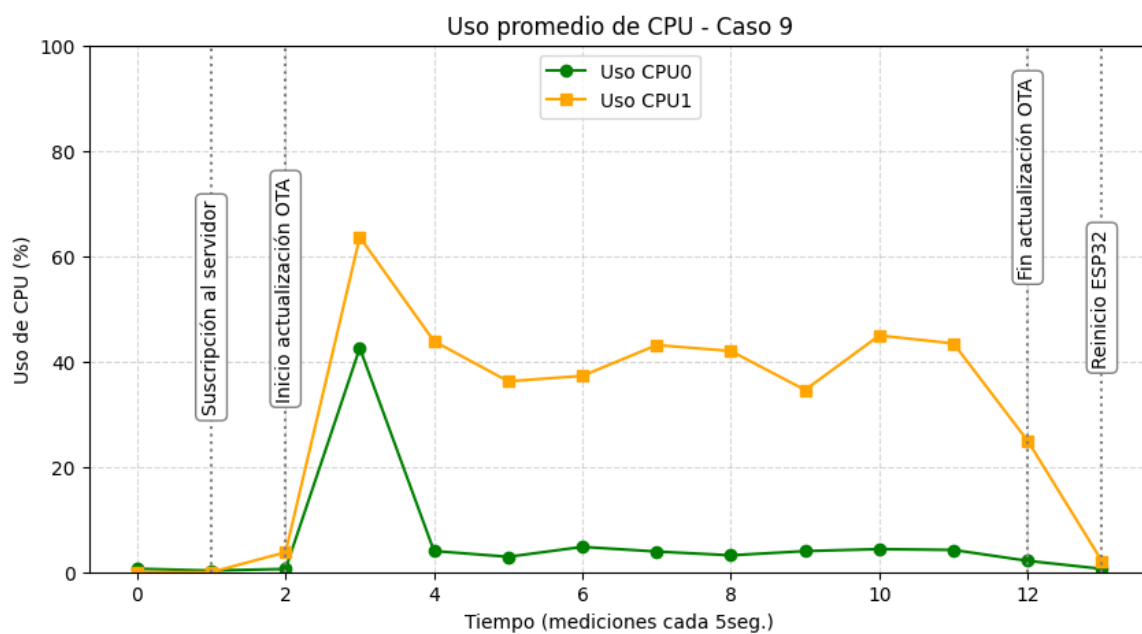


Figura A.9 Uso promedio de CPU - Caso 9

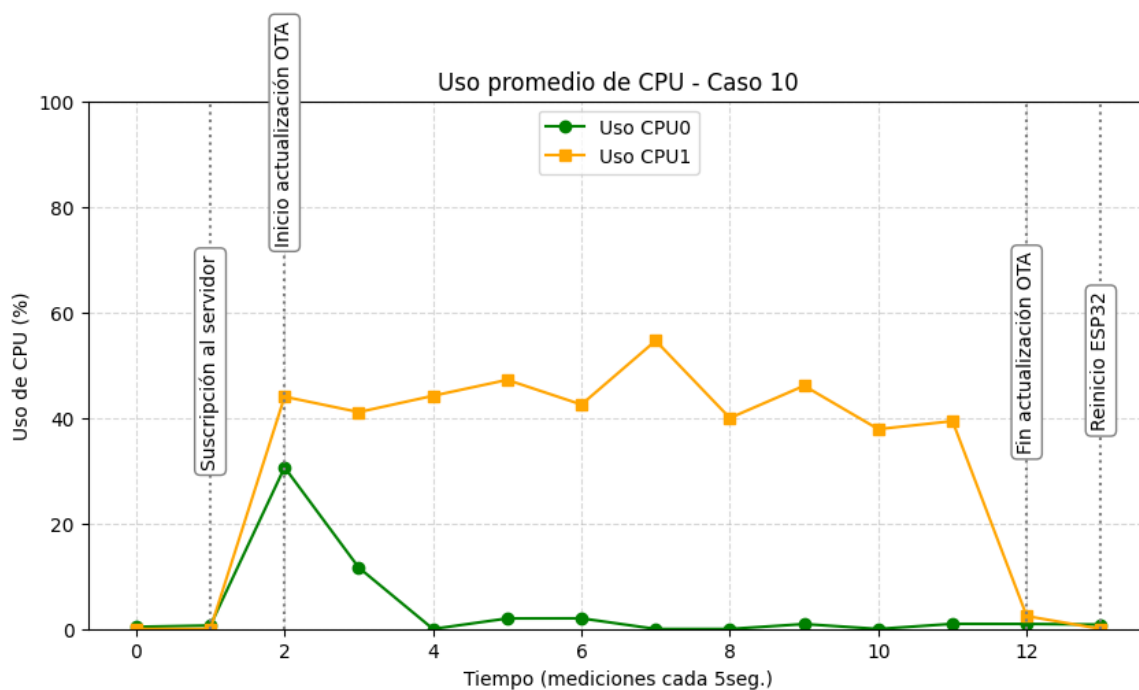


Figura A.10 Uso promedio de CPU - Caso 10

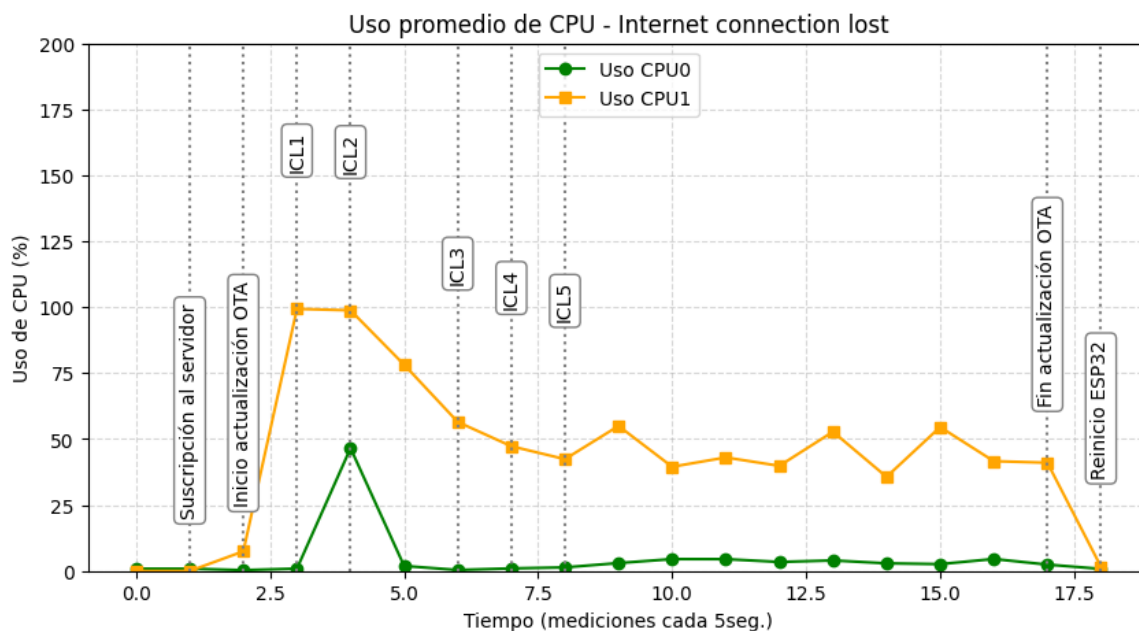


Figura A.11 Uso promedio de CPU - Internet connection lost

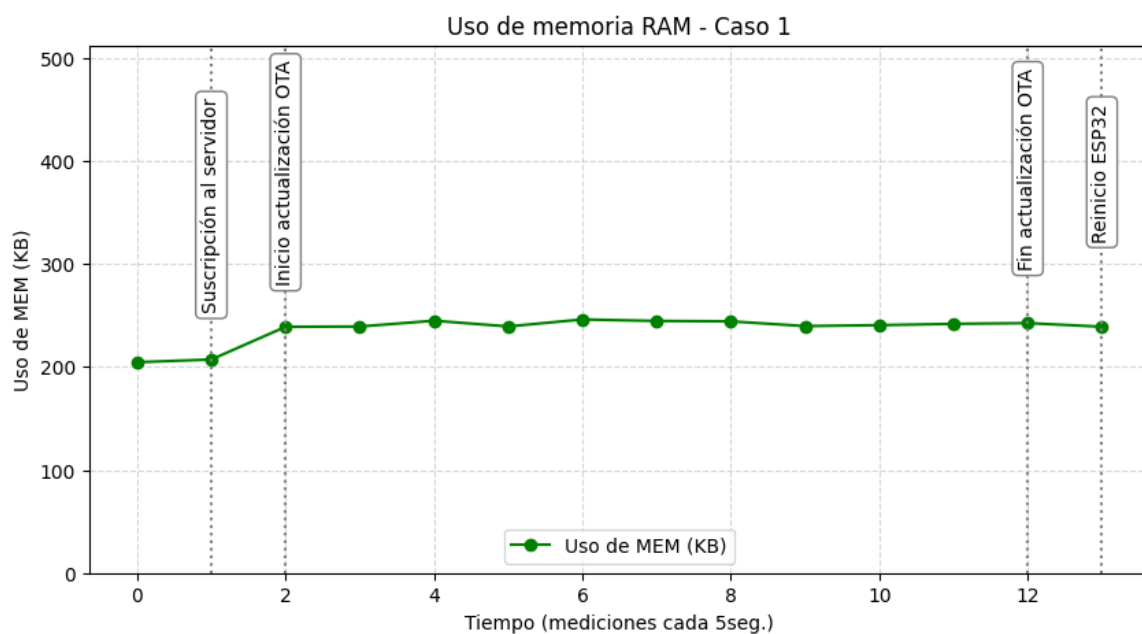


Figura A.12 Uso de memoria RAM - Caso 1

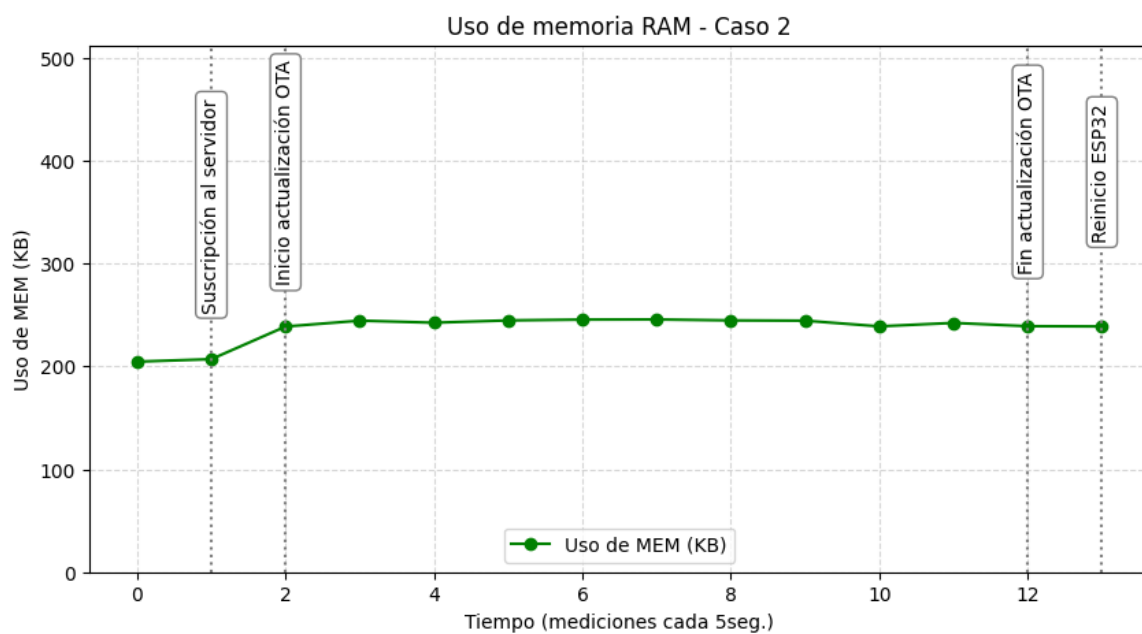


Figura A.13 Uso de memoria RAM - Caso 2

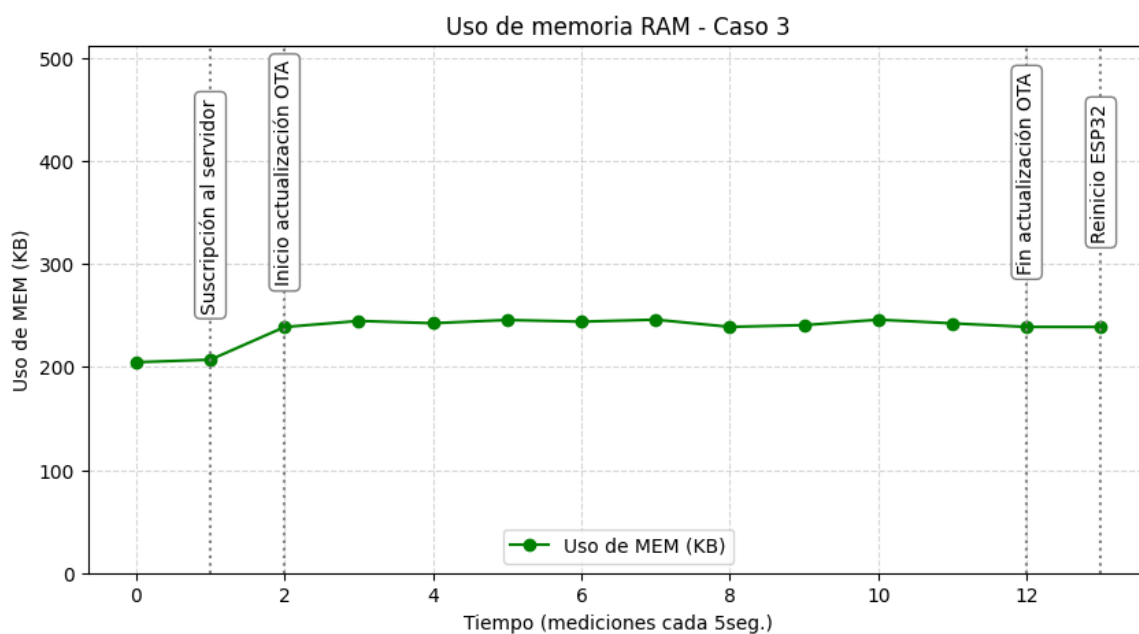


Figura A.14 Uso de memoria RAM - Caso 3

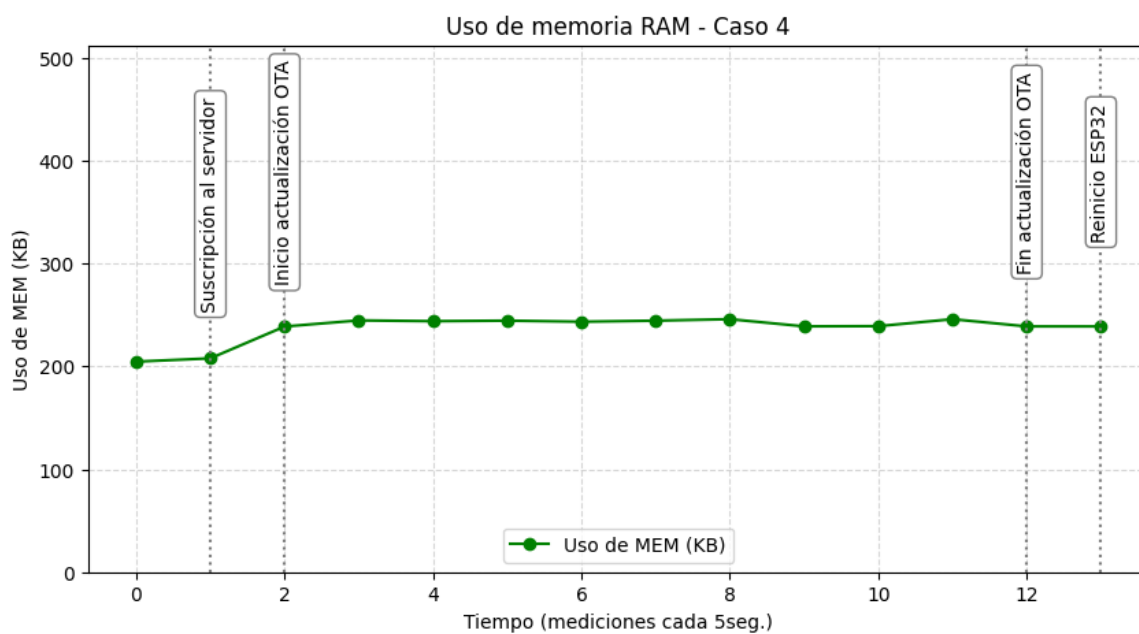


Figura A.15 Uso de memoria RAM - Caso 4

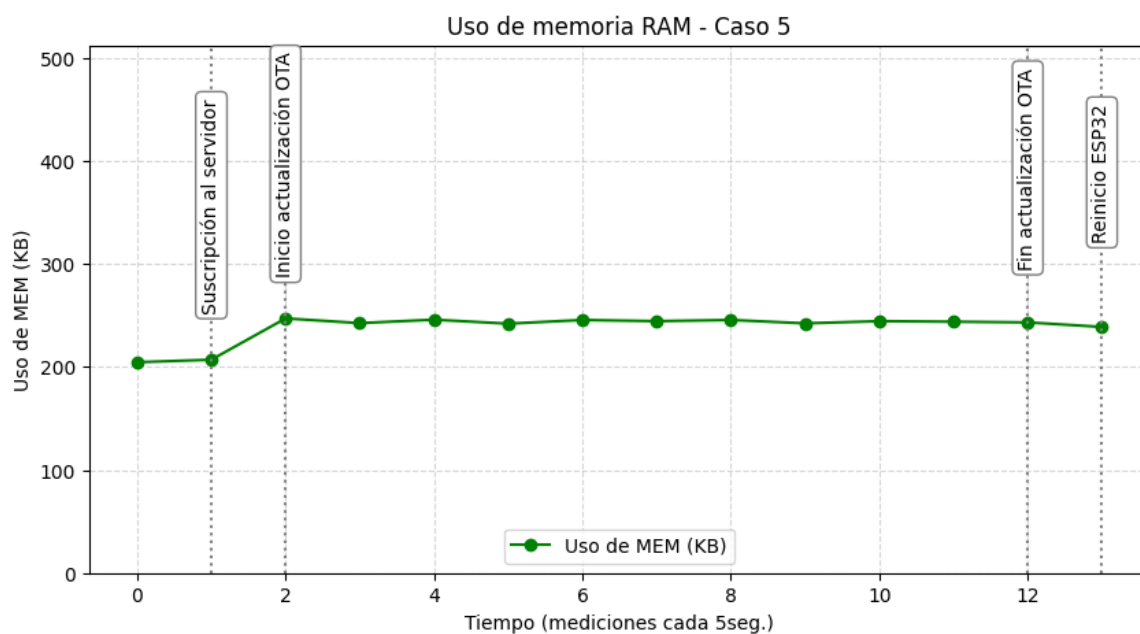


Figura A.16 Uso de memoria RAM - Caso 5

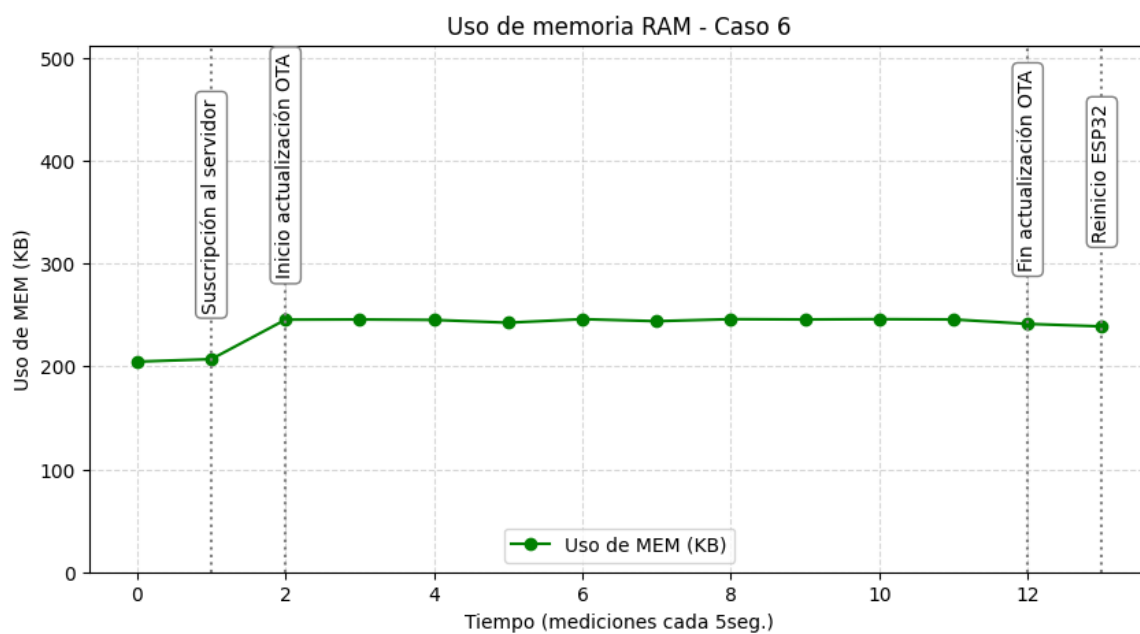


Figura A.17 Uso de memoria RAM - Caso 6

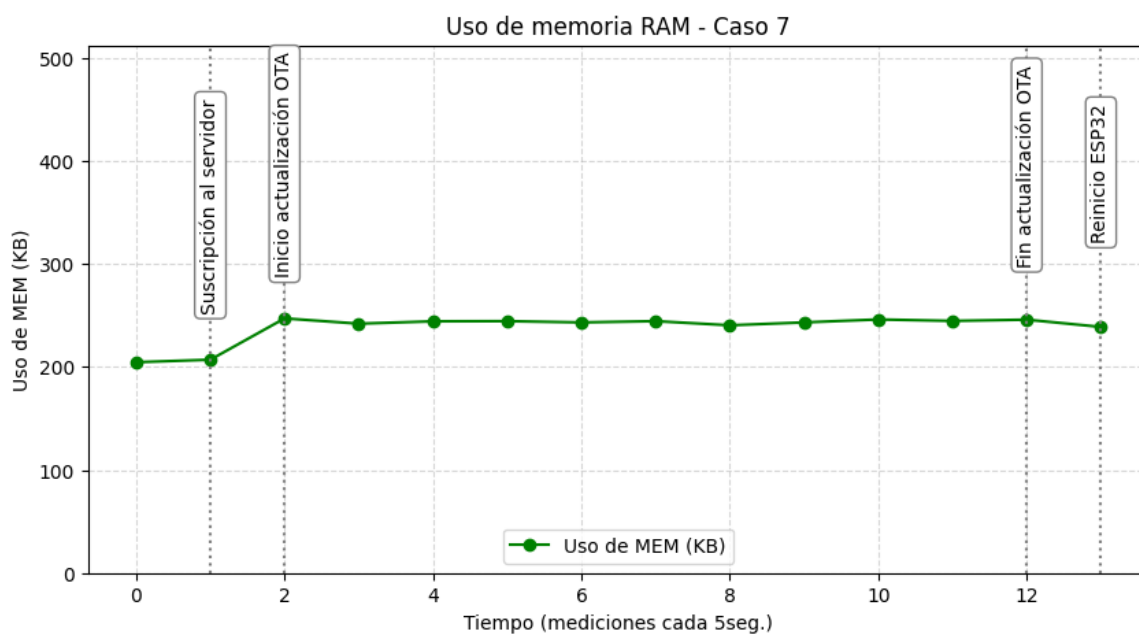


Figura A.18 Uso de memoria RAM - Caso 7

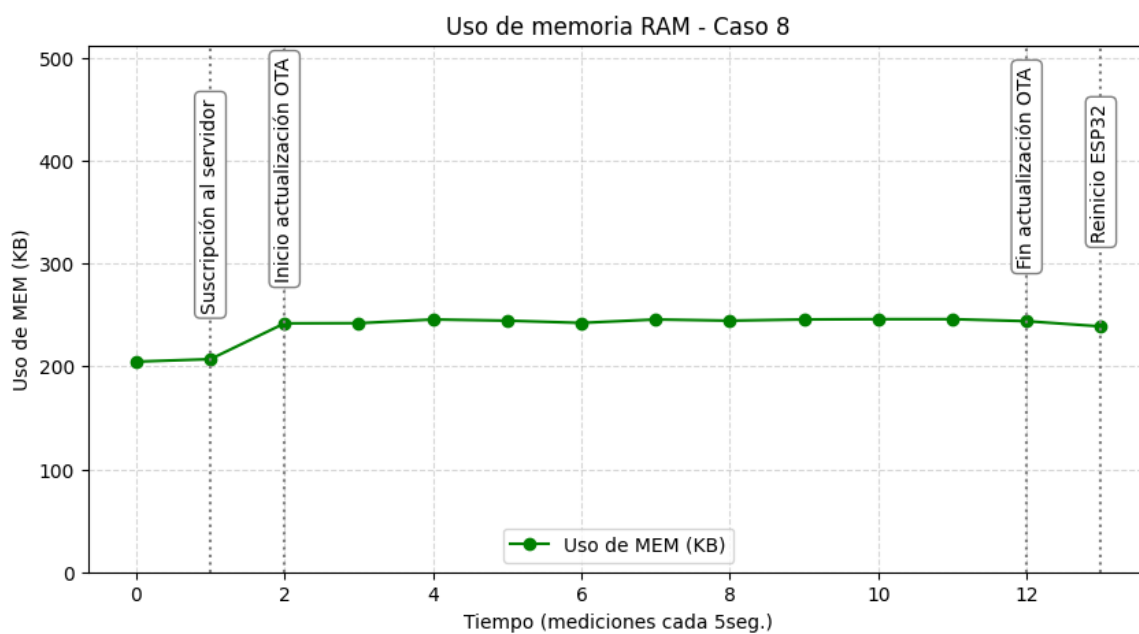


Figura A.19 Uso de memoria RAM - Caso 8

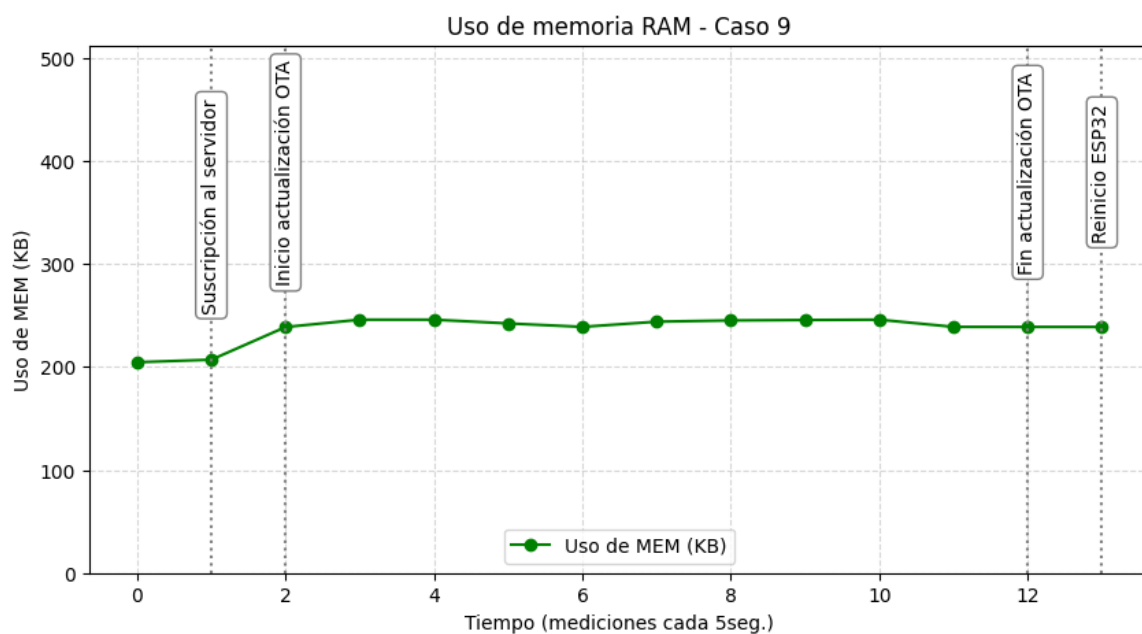


Figura A.20 Uso de memoria RAM - Caso 9

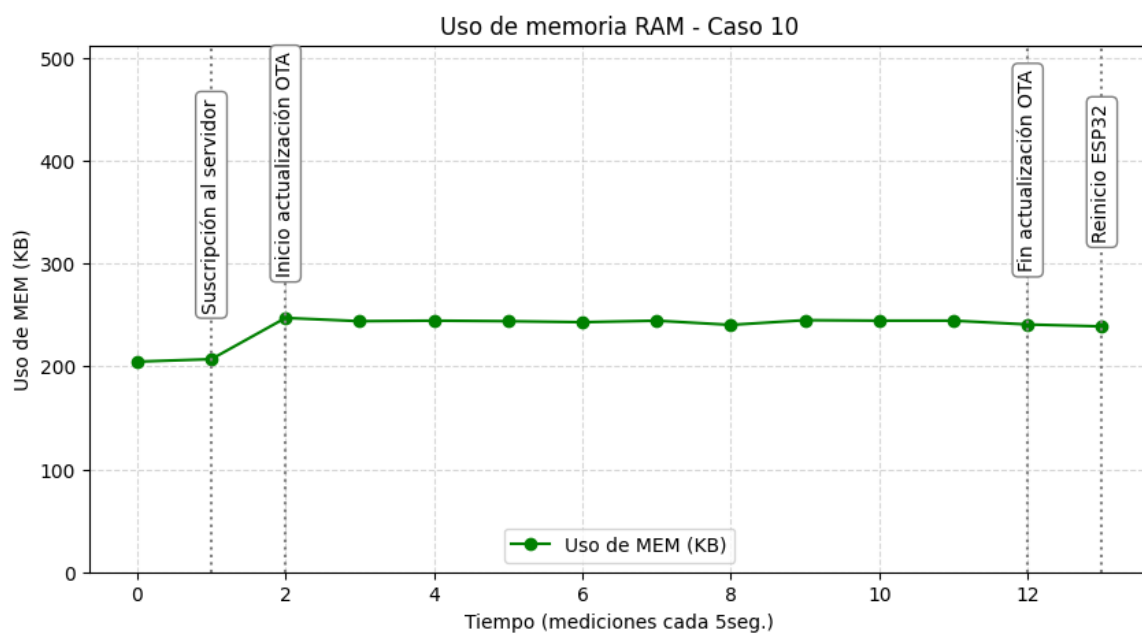


Figura A.21 Uso de memoria RAM - Caso 10

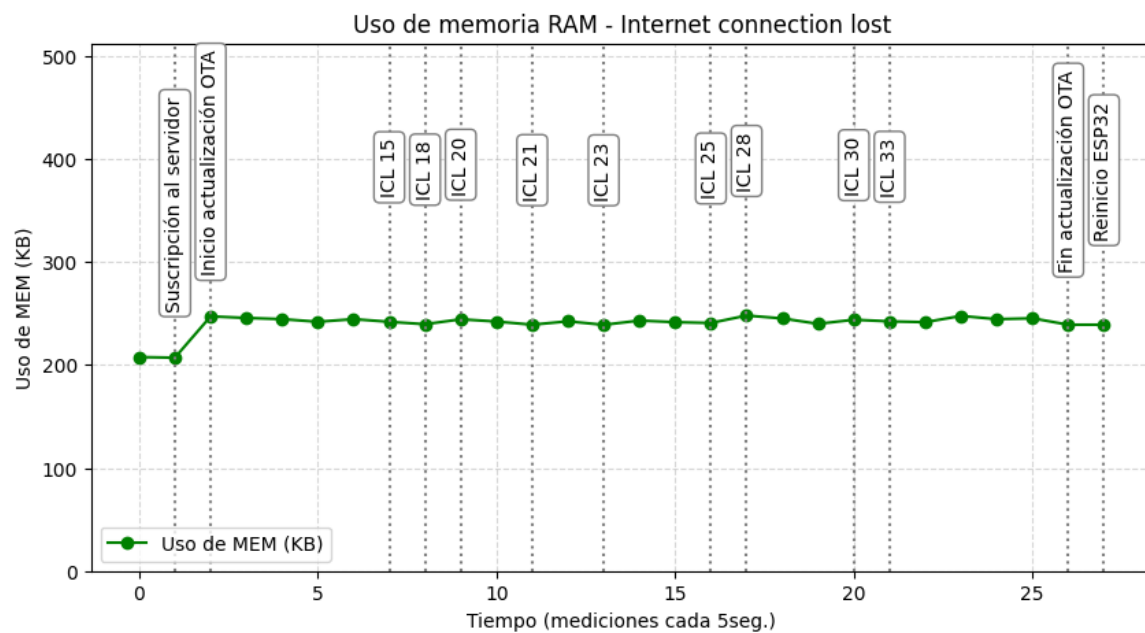


Figura A.22 Uso de memoria RAM - Internet connection lost

Apéndice B

Glosario

Bootloader: Programa ligero que se ejecuta al iniciar el dispositivo y se encarga de cargar el firmware en la memoria principal.

Boards: Placas electrónicas que integran microcontroladores, sensores y periféricos, utilizadas en proyectos de hardware embebido e IoT.

Buffers: Espacios de memoria utilizados para almacenar temporalmente datos durante operaciones de entrada/salida, evitando pérdidas o bloqueos.

Canary releases: Estrategia de despliegue que libera una nueva versión del software a un subconjunto limitado de usuarios para detectar errores antes de una liberación total.

Checksum: Valor numérico calculado a partir de un conjunto de datos que permite verificar su integridad tras una operación de transmisión o almacenamiento.

Checkout: Acción de obtener una copia de un repositorio o rama específica en un sistema de control de versiones como Git.

Cloud: Infraestructura distribuida que ofrece servicios de computación, almacenamiento y redes accesibles a través de Internet.

Contenedor: Entorno aislado para ejecutar aplicaciones, usualmente gestionado por herramientas como Docker.

Cyber-Physical Systems (CPS): Sistemas que integran componentes físicos y computacionales, como sensores y actuadores, para interactuar con el mundo real.

Dashboards: Interfaces gráficas que visualizan información clave como métricas, logs o telemetría en tiempo real para monitoreo y análisis.

Device: Dispositivo físico que recopila, procesa y transmite datos, como sensores, actuadores o microcontroladores.

Device Profile: Conjunto de configuraciones que describen las capacidades, comportamiento y parámetros de un dispositivo IoT dentro de una plataforma.

Downtime: Período durante el cual un sistema o servicio no está disponible o no funciona

correctamente.

Dual-core: Procesador con dos núcleos de procesamiento independientes, lo que permite ejecutar múltiples tareas simultáneamente.

Edge computing: Modelo de computación que realiza el procesamiento de datos cerca del lugar donde se generan (en el borde de la red).

Edge server: Servidor ubicado en el borde de la red que permite procesar datos más cerca del origen, reduciendo la latencia y el consumo de ancho de banda.

Fail-safe: Diseño que garantiza que un sistema entre en un estado seguro ante fallos o errores inesperados.

Flash (Memoria): Tipo de memoria no volátil utilizada para almacenar firmware y configuraciones persistentes en dispositivos embebidos.

Flashing: Proceso de grabar el firmware en la memoria flash de un dispositivo, habitualmente durante el despliegue o actualización.

Fog computing: Extensión del edge computing, que distribuye recursos computacionales entre el edge y la nube.

Fog server: Nodo intermedio entre el edge y la nube que permite el procesamiento distribuido, almacenamiento local y decisiones rápidas en arquitecturas IoT.

Frameworks: Conjuntos estructurados de bibliotecas, herramientas y convenciones que facilitan el desarrollo de software, especialmente en entornos complejos como IoT.

Gateway: Dispositivo intermediario que conecta diferentes redes o protocolos, permitiendo la comunicación entre sensores/dispositivos IoT y servidores.

Helm: Gestor de paquetes para Kubernetes que facilita la instalación y actualización de aplicaciones dentro del clúster.

Infrastructure as Code (IaC): Práctica que permite definir la infraestructura mediante código, facilitando la replicabilidad y el control de versiones.

Internet connection lost (ICL): Estado en el que un dispositivo pierde su conexión con la red o la nube, impidiendo la transmisión de datos o actualizaciones.

Kubernetes: Plataforma de orquestación de contenedores que automatiza el despliegue, escalado y operación de aplicaciones.

MQTT: Protocolo de mensajería ligero utilizado en IoT para comunicaciones máquina a máquina (M2M).

Microservicios: Arquitectura de software basada en servicios pequeños e independientes que se comunican entre sí.

Non-Volatile Storage (NVS): Memoria no volátil utilizada para almacenar datos persistentes como tokens de autenticación.

On-premise: Infraestructura de TI que se encuentra físicamente en las instalaciones de la

organización, en contraposición a la nube.

Out-of-memory: Error que ocurre cuando un dispositivo o proceso excede su capacidad de memoria disponible, provocando fallos o reinicios.

Overflow: Situación en la que se excede el tamaño máximo de un buffer o variable, lo que puede generar errores lógicos o de seguridad.

Peer-to-peer: Modelo de comunicación entre dispositivos en el que cada nodo actúa como cliente y servidor, sin necesidad de un intermediario central.

Release: Versión de software oficialmente publicada, habitualmente numerada y documentada, lista para su uso o despliegue.

Rollback: Mecanismo para revertir una actualización fallida y restaurar una versión anterior del firmware o software.

Rollback attacks: Ataques que explotan la posibilidad de revertir un sistema a una versión anterior y vulnerable del software o firmware.

Router: Dispositivo de red que dirige paquetes de datos entre distintas redes, determinando la mejor ruta disponible para cada uno.

Runner: Agente que ejecuta las tareas definidas en un flujo de trabajo (por ejemplo, pruebas o compilaciones en GitHub Actions).

SDK: Conjunto de herramientas y bibliotecas que facilitan el desarrollo de aplicaciones para una plataforma específica.

SHA-256: Algoritmo de hash criptográfico que genera un valor de 256 bits, utilizado para comprobar la integridad y autenticidad de datos.

Telemetría: Información que los dispositivos envían a una plataforma para su monitoreo remoto.

TI (IT): Acrónimo de Tecnología de la Información, que se refiere al uso de computadoras y software para gestionar información.

Time-to-market: Tiempo que transcurre entre el inicio del desarrollo de un producto y su disponibilidad comercial; indicador clave en ciclos de entrega continua.

Token: Código único que permite identificar y autorizar a un dispositivo o usuario frente a una plataforma o API.

Watchdog: Componente que reinicia el sistema si detecta errores o bloqueos en su funcionamiento.

Wear leveling: Técnica usada en memorias flash para distribuir equitativamente los ciclos de escritura y prolongar la vida útil del dispositivo.

YAML: Lenguaje de marcado utilizado para definir archivos de configuración. Usa sangrías para representar estructuras jerárquicas como listas y diccionarios.

